# Achieving Speculative Intermittent Computation on Commodity Energy Harvesting Hardware

Yida Zhang, Shao-Yu Huang, Byounguk Min, Andrew Inho Park, Gan Fang, Jongouk Choi, and Changhee Jung

***Abstract*—This paper presents SpecIC, a software-based speculative intermittent computing scheme for commodity batteryless energy harvesting platforms. SpecIC speculatively executes recoverable program regions without first securing sufficient energy for their completion, by predicting whether they will encounter a power failure. By dynamically adapting its degree of speculation to the the harvested energy quality, SpecIC can seamlessly adapt to the full spectrum of power conditions, unlike prior schemes designed for either strong or weak power scenarios. Also, SpecIC incorporates a lightweight reset mechanism that rapidly discards stale optimistic predictor state, thereby maintaining stability even under sudden degradations in power availability. The experimental results show that SpecIC consistently outperforms all prior schemes by up to 274% across various power conditions. Even under highly dynamic transitions between power conditions, SpecIC outperforms all prior schemes by up to 228%.**

***Index Terms*—Intermittent computing, speculation, reliability.**

## I. Introduction

Recent advances in energy harvesting offer new opportunities for IoT devices [1], wearables [2], and health and wellness monitors [3]. With energy harvesting technology, these devices can operate without the restriction of a tethered power supply or a battery, which is generally expensive, bulky, and requires periodic maintenance. Instead, an energy harvesting system collects and buffers ambient energy, e.g., from solar, radio frequency (RF), and motion sources [4], [5]. However, they are weak and unstable, causing frequent power failure on which all volatile data vanishes. The energy harvesting system thus runs intermittently, operating only when the energy buffer (a tiny capacitor) is sufficiently charged and otherwise experiencing a long outage until the buffer is recharged enough to reboot. This is characterized as *intermittent computing* [6], [7], [8].

Given the frequent outages, intermittent computing architectures often leverage non-volatile memory (NVM) as main memory, with only registers serving as volatile storage [9], [10], [11], [12]. To resume a power-interrupted program correctly, researchers have proposed software-based crash-consistency mechanisms [13], [14] on top of the intermittent architecture [15], [16], [17], [18]. They typically partition a program into multiple regions and checkpoint volatile registers to NVM within the regions or at region boundaries [19]. In the wake of a power failure, they restore the registers and resume from the beginning of the interrupted region.

The primary challenge with intermittent computing lies in its power condition bias. All existing software systems are narrowly optimized for specific parts of the power spectrum, e.g., strong or weak power scenarios, leading to significant performance degradation when energy harvesting conditions shift. For example, WARio [9] performs well under relatively strong power conditions but cannot handle *stagnation* that occurs when some long region is power-interrupted repeatedly, *ad infinitum*. Chinchilla [15] is also designed for strong power scenarios, where it skips checkpointing and thus reduces the amount of undo logging—its basis for crash consistency. While re-execution overhead quickly dominates as power conditions worsen—due to skipped regions—Chinchilla still maintains a preference for strong power conditions.

Conversely, RockClimb targets weak power scenarios. Its compiler divides a program into stagnation-free regions with so-called *power failure immunity* (PFI) [16], ensuring the energy required for each region never exceeds the energy buffer capacity. At the start of every region, RockClimb [16] waits for the capacitor to fully charge, thus eliminating in-region failures entirely. Although this achieves reexecution-free intermittent computing, it causes significant performance degradation when power conditions improve. In short, strong-power-biased frameworks struggle when transitioning to weaker power conditions, whereas the weak-power-biased one suffers a substantial performance hit when the conditions get strong.

Motivated by the limitations, this paper introduces SpecIC, the first intermittent computing framework that maintains high performance across varying power conditions—unlike Chinchilla and RockClimb tailored to only one side of the spectrum. SpecIC integrates the strengths of the prior works while addressing their shortcomings. To guarantee stagnation freedom and maintain performance under weak power conditions, SpecIC adopts RockClimb's PFI region formation. As RockClimb's poor performance results from waiting for charging at every region boundary, SpecIC bypasses the waiting if unnecessary. This implies that SpecIC loses the advantage of no reexecution and needs to provide crash consistency.

Instead of using costly undo logging, SpecIC exploits idempotent regions [20] used by WARio—and other work [18], [21], [22]. The SpecIC compiler unifies their region formation algorithms while preserving both idempotence and PFI, resulting in a program partitioned into PFI-enforced idempotent regions. To further improve performance under good power conditions, SpecIC performs checkpoint pruning [21], [23], [22] that greatly reduces the number of register checkpoints. Nevertheless, if SpecIC were to rely solely on PFI regions without charging at each region boundary, many regions would still be executed twice under weak power conditions—though stagnation freedom is still ensured by PFI.

To address this, we propose a novel approach, Speculative Intermittent Computing (SpecIC), that employs the speculative execution of PFI-enforced idempotent regions. This is governed by a simple yet effective power failure predictor whose trained state is conditionally retained across power outages

based on the duration of the power-off interval. A long outage triggers the predictor to retrain from scratch, whereas a short outage allows the inheritance of the previously trained state. If the next region is predicted not to be power-interrupted, SpecIC speculatively executes the next region without waiting at the region boundary. Otherwise, SpecIC waits at the region boundary until the energy buffer fully charges to eliminate in-region outages. The beauty of such speculative execution is that it can adapt to dynamically varying power conditions. That is, SpecIC can maximize forward execution progress under good conditions, while minimizing power interruptions and the necessary reexecutions under bad conditions.

We evaluated SpecIC against 4 state-of-the-art software intermittent computing frameworks under good, moderate, and bad power conditions. Notably, prior work evaluates systems by fixing a power condition for the entire program execution—though they report results across different conditions—despite the reality that intermittent systems execute continuously and experience fluctuating energy availability. To address this gap, we introduce, for the first time, an emulation infrastructure that dynamically varies power conditions at run time using Markov chains [24] that probabilistically model the transition among the power conditions. To reflect the growing trend toward soft real-time intermittent systems [25], [26], we also conduct throughput tests for SpecIC and its competing schemes with their real-time task schedulability in mind.

According to our experimentation, SpecIC consistently delivers the best performance across all three power conditions. Under good power conditions, SpecIC achieves 1.6% average speedup over WARio and is 117%, 247% and 274% faster than Chinchilla, RockClimb and EarlyBird, respectively. Under moderate power conditions, WARio suffers from excessive reexecutions and stagnation for certain benchmarks, and Chinchilla's performance degrades due to the checkpointing and undo logging. Under bad conditions, SpecIC outperforms Chinchilla, RockClimb and EarlyBird by 181%, 91% and 103%, respectively, whereas WARio often suffers stagnation.

When power conditions transition under the Markov chain model, SpecIC performs the best with a higher margin, yielding speedups of 139%, 214% and 228% over Chinchilla, RockClimb, and EarlyBird, respectively. Finally, for throughput evaluation in a soft real-time system with an EDF (earliest deadline first) scheduler [27], SpecIC increases the number of completed real-time tasks by up to 371% compared to RockClimb. Our contributions are summarized below:

- It is the first to realize speculative intermittent computing, that adapts to varying power conditions, on a real device.
- It outperforms all prior software schemes. Unlike their evaluation, ours dynamically varies power conditions.
- It devises a simple yet effective power failure predictor to enable adaptive and performant speculative execution.

## II. Background and Motivation

### A. Energy Harvesting System Basics

Energy harvesting systems collect and buffer ambient energy in a tiny capacitor and manage to execute programs only while the capacitor offers sufficient energy. Otherwise, they are power-interrupted and hibernate for recharging. Here, since the input power is generally weak, energy harvesting systems suffer frequent power failures during which all volatile data are lost, characterizing them as intermittent computing systems. To preserve critical program states across power failures, commodity intermittent computing architectures use NVM as main memory and avoid caches due to the complexity they introduce for crash consistency support [13], [28], [29]. As a result, the CPU registers are the only volatile storage.

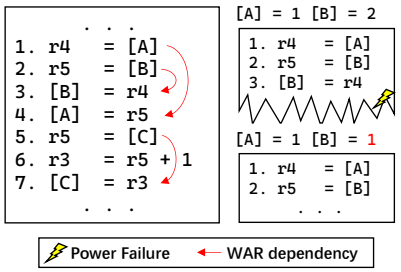



Fig. 1: Memory inconsistency; the region is supposed to swap the values of [A] and [B], initialized as 1 and 2 respectively, and update the value of [C].

On top of this architecture, software-based intermittent computing frameworks rely on rollback recovery [18], [30], [31], [32], [33], also used in resilience and persistence work [34], [35], [22], [36], [37], [38], [39], [40], [41]. Typically, program is divided into recoverable regions so that any power-failed region rolls back to the beginning of the region and restarts in the wake of the failure. This rollback recovery leads to two issues that must be resolved to correctly resume a power-interrupted program: *memory inconsistency* and *stagnation*.

*1) Issue 1: Memory Inconsistency:* Figure 1 shows memory inconsistency using a program that swaps the memory values at addresses A and B. In the example region, memory values $[A]$ and $[B]$ have Write-After-Read (WAR) dependencies [42]. As shown on the right of Figure 1, suppose a power failure occurs after line 3 during the first run, in which case the value originally from [A] (loaded into r4) has been stored into [B] and persisted in NVM. Upon re-execution in the wake of the power failure, $[B]$ and $[A]$ both have 1. Thus, the program will fail to swap their original values, with r5 reading the wrong value 1 not 2—at line 2 in the second run. This memory inconsistency never occurs under a continuous power supply.

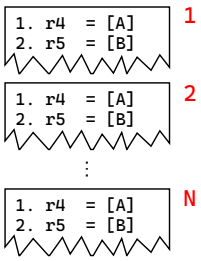



Fig. 2: Stagnation.

*2) Issue 2: Stagnation:* Consider a scenario where a region requires $t$ units of time to complete, and outages occur so frequently that power cycle (power-on period) is shorter than $t$. The system can only re-execute the same region repeatedly, thereby wasting harvested energy without making any progress [43]. Figure 2 shows the stagnation effect with the same region as in Figure 1. During each power cycle, the available energy permits only 2 instructions to execute, while the region contains 7 instructions. The region thus never reaches its end despite endless reexecutions, causing stagnation.

TABLE I: Comparison of Intermittent Computing Systems

| Intermittent System | Suitable Power Conditions | Stagnation freedom | Checkpoint & Logging Overhead | Re-execution Penalty | Region Boundary Overhead | No User Intervention | Adaptive Execution |
|---|---|---|---|---|---|---|---|
| Ratchet [18] (2016) | Good | ✗† | High | Medium | Register Checkpoint, Medium | ✓ | No |
| Chinchilla [15] (2018) | Good | ✓‡ | High | High | Register Checkpoint, Medium | ✗ | Fragile |
| WARio [9] (2022) | Good to moderate | ✗ | Medium | Medium | Register Checkpoint, Medium | ✓ | No |
| RockClimb [16] (2022) | Bad | ✓ | Low | Low | Recharge, High | ✓ | No |
| EarlyBird [17] (2024) | Bad to moderate | ✓ | Medium | Low | Reg. Checkpoint & Recharge, High | ✓ | No |
| **SpecIC (Ours)** | **All** | ✓ | **Very Low** | **Low** | **Speculative Recharge, Low** | ✓ | **Robust** |

† While Ratchet claims that its timer-based checkpoint can avoid stagnation, it causes wrong recovery for regions that have WARAW dependencies [16].
‡ As Chinchilla relies on manual energy debugging, it is not sound [44] as with other dynamic analysis, failing to avoid stagnation for all program paths/inputs.

### *B. Related Work*

To address those two issues, many intermittent computing frameworks have been proposed. However, they all fall short in one or more aspects, resulting in lower efficiency under certain power conditions or even stagnation. Ratchet [18] and WARio [9] focus on eliminating memory inconsistency. They adopt idempotent regions [45], each of which has no WAR dependency for side-effect-free reexecution. However, they leave stagnation unaddressed; many regions can be excessively long, consuming more energy than the energy buffer can offer, which leads to frequent reexecutions and potential stagnation. As a result, while their performance is high under favorable power conditions, it degrades significantly when conditions worsen and even fails to terminate due to stagnation, as confirmed in our evaluation (Section VI-B).

Chinchilla [15], RockClimb [16], and EarlyBird [17], on the other hand, target stagnation. To avoid it, Chinchilla requires users to measure the energy consumption of regions and split them if they exceed the energy buffer capacitance, whereas both RockClimb and EarlyBird let their compilers automatically form PFI-enforced regions for the input program. While these frameworks avoid memory inconsistency, they incur high overhead under certain power conditions. Chinchilla leaves undo logs—each involving multiple NVM writes—before every store instruction. Its adaptive execution mechanism maintains a timer, and upon expiration, performs checkpointing and clears the undo logs. As power conditions improve, it extends the timer duration and skips checkpointing for successive regions—avoiding redundant undo logging along the way—until the timer expires. However, this incurs potentially huge reexecution overhead, e.g., if checkpointing is skipped for 100 regions and a power failure follows before the timer expires, Chinchilla must resume execution from 100 regions earlier. Also, its adaptation logic is easily fooled by long outages—common in intermittent systems—as shown in Section III-A. Besides, when power conditions worsen, the logging and checkpointing overhead quickly becomes dominant.

RockClimb and EarlyBird resolve the memory consistency issue by waiting at each region boundary until the energy buffer fully charges, preventing any in-region power failures. However, this results in poor performance under good power conditions, where the waiting is often unnecessary and becomes detrimental due to the significant overhead of transitioning between execution and waiting modes at region boundaries. Section VI-B shows that Chinchilla performs reasonably well under good power conditions, but its performance degrades significantly as power conditions worsen. In contrast, RockClimb and EarlyBird perform much better than Chinchilla under poor power conditions, but incur significant overhead under good conditions. These contrasting behaviors highlight the need for predicting upcoming power conditions and adapting the intermittent computation accordingly. Without such prediction-based adaptation, systems risk either over-provisioning for potential outages—incurring unnecessary overhead when power is stable—or under-provisioning, leading to excessive logging or wasted energy under poor power conditions.

### *C. Challenge of Varying Power Conditions*

Unfortunately, all prior software-based intermittent computing frameworks evaluate their systems at a fixed distance between the device and its energy harvester. This explains why Chinchilla appears to perform well—despite the fragile adaptation—but degrades significantly when power conditions fluctuate, as shown in Section VI-B. In real-world energy-harvesting systems, however, power conditions vary over time. For example, consider people wearing a battery-free smart wristband [46] while moving around a gym with sporadically installed energy harvesters. Depending on its proximity to a harvester, the wristband may receive more or less energy, causing its power condition to vary continuously. To address this challenge, we propose Speculative Intermittent Computing (SpecIC), the first software-based intermittent computing framework designed to maintain high performance under varying power conditions. Table I summarizes related work and highlights that only SpecIC simultaneously achieves stagnation freedom, low overhead, and robust adaptation across the full spectrum of power conditions. No prior work maintains performance in the face of power fluctuations.

## III. SpecIC Approach

The goal of SpecIC is to enable performant intermittent computing under varying power conditions. To prevent stagnation, SpecIC adopts the PFI-enforced region formation of RockClimb [16]. Observing its unnecessary charging, SpecIC devises a lightweight power failure predictor to speculatively proceed at region boundaries without waiting for the capacitor to fully charge if an outage is unlikely, as shown in Figure 3. To address memory inconsistency caused by reexecutions when handling *misspeculation* (an outage occurs despite being

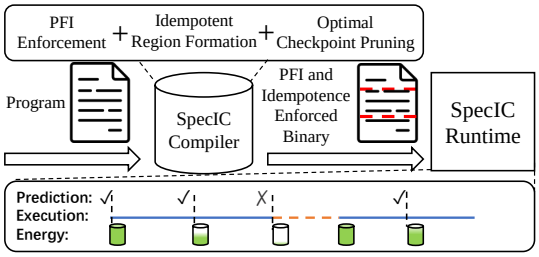



Fig. 3: Overview of SpecIC; vertical dotted lines denote region boundaries; orange dotted lines denote waiting at a region boundary if prediction dictates.

predicted otherwise), SpecIC employs idempotent recovery as in Ratchet/WARio and others [20], [21], [22]. Finally, SpecIC incorporates *checkpoint pruning* to enhance performance. The rest of this section describes these techniques and explains how they collectively integrate into SpecIC.

*1) Power Failure Immunity (PFI):* SpecIC adopts PFI to effectively prevent stagnation. The root cause of stagnation in intermittent computing is the repeated failure of long code regions whose energy requirements exceed the system's energy buffer capacity. PFI eliminates this issue by ensuring that every region is short enough to be executed completely with a single full charge. The key insight is that after a power failure, the system always restarts (reboots) with its energy buffer fully charged, guaranteeing a minimum duration of uninterrupted execution time, termed the Safe Active Time (SAT) [16]. For a microcontroller unit (MCU) $\mu$, a region $r$ is considered SAT-safe—or a PFI-enforced region—if the Worst-Case Execution Time (WCET) of $r$ satisfies:

$$WCET(r) \leq SAT(\mu) \tag{1}$$

If every region in the program satisfies this PFI constraint, stagnation becomes theoretically impossible because the system is always guaranteed sufficient energy to complete any region from a fresh start (reboot). RockClimb [16] leverages this constraint by using a compiler to automatically partition the input program into a sequence of SAT-safe regions, ensuring at most a single in-region outage. However, to eliminate in-region outages entirely, RockClimb conservatively waits for the energy buffer to fully recharge at every region boundary. EarlyBird [17] later optimizes this by waiting only for the estimated energy required for the next region.

While both schemes prevent stagnation, these "always-wait" strategies incur significant performance penalties under good power conditions, where continuous energy is available and waiting becomes unnecessary. Since SpecIC aims to maximize performance across all power conditions, simply adopting PFI's conservative execution model is insufficient. Instead, SpecIC uses PFI solely as a safety net to guarantee stagnation freedom, while employing a lightweight runtime for predicting upcoming power failures to bypass unnecessary waiting at region boundaries when power conditions are good.

### A. Featherweight Power Failure Predictor

To maintain performance across varying power conditions, SpecIC conducts speculative execution of PFI-enforced regions. Under good power conditions, SpecIC speculatively proceeds past region boundaries without waiting, maximizing performance by predicting that imminent power failure is unlikely (*misspeculation* handling is deferred to Section III-B). Under poor power conditions, however, SpecIC disables speculation to minimize costly reexecutions, i.e., it waits for the energy buffer to fully recharge in anticipation of an upcoming failure. To decide when to speculate and to control the degree of speculation (i.e., how many region boundaries may safely bypass charging), SpecIC introduces a simple yet effective power failure predictor tailored for intermittent systems.

Designing such a predictor requires meeting strict constraints, most notably ultra-low energy. Because the predictor is invoked at every region boundary, any heavyweight operation would incur prohibitive cost, outweighing the benefits of speculation. Prior systems such as Elastin [14] and Chinchilla [15] employ adaptation schemes that rely on hardware timers and interrupts to measure execution time and energy consumption. However, managing timers, configuring interrupts, and handling interrupt service routines all incur non-negligible overhead—cost that is particularly harmful in energy-constrained intermittent systems.

In response, SpecIC introduces a featherweight power failure predictor designed explicitly for minimal overhead. Instead of relying on costly hardware timers, SpecIC maintains only two integers: **region counter** and **predicted-safe window**. The window represents how many regions SpecIC expects to execute successfully without power failure, while the counter tracks how many regions have actually completed since the last recovery or recharge. At each region boundary, SpecIC performs a single integer comparison between the counter and the window. If the counter is within the window, SpecIC speculatively proceeds. Once the counter reaches the window limit, SpecIC pauses to recharge—ensuring the next region executes failure-atomically—then resets the counter and adjusts the window using a heuristic. This simple integer comparison introduces negligible overhead, making the predictor suitable for frequent checks at region boundaries. We evaluated multiple heuristics (Section IV-B2) and selected one that balances aggressive speculation under good power conditions with rapid stabilization when conditions deteriorate.

Beyond minimizing overhead, SpecIC's predictor is designed to handle sudden power-degradation scenarios more robustly than prior heuristics, which can be easily misled by abrupt changes. Figure 4 illustrates how Chinchilla and SpecIC behave differently under such a scenario, highlighting their contrasting recovery and adaptation mechanisms. Initially, power conditions are good, and thus both systems adapt accordingly. Chinchilla skips checkpointing at region boundaries, while SpecIC skips waiting at region boundaries. When a short power outage occurs at region $r15$, Chinchilla must roll back to the last checkpoint (region $r1$), losing all execution progress made since then. In contrast, SpecIC only reexecutes the interrupted region $r15$ thanks to the idempotent recovery mechanism.

The difference in adaptation becomes critical when power conditions suddenly degrade, e.g., when a person wearing a solar-powered device steps into a shaded area of the gym. In Figure 4, this degradation occurs at the end of region $r60$ for Chinchilla and $r75$ for SpecIC, resulting in a prolonged power

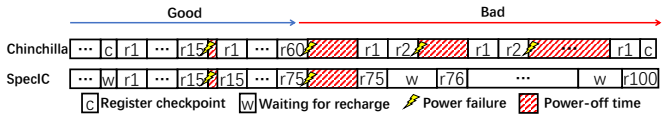



Fig. 4: Conceptual comparison of adaptation and recovery under sudden power degradation. At boundaries, Chinchilla only has register checkpoints and SpecIC only has waiting. Both are skipped during $r1 - r15$.

outage (red hatched area). Chinchilla's adaptation heuristic proves fragile in this scenario: it fails to react quickly to the sudden drop in power quality. First, Chinchilla loses all progress by rolling back to region $r1$ again. Then, due to its slow adaptation logic, it continues to skip checkpointing for subsequent regions (e.g., $r1, r2$), causing repeated power outages and reexecution penalties. As shown in the figure, Chinchilla struggles through multiple failure cycles before finally managing to checkpoint region $r1$, which becomes the new recovery point for subsequent outages.

In contrast, SpecIC employs a fast-drop policy to rapidly detect deteriorating power conditions and halt speculation. The key insight is that a long power-off interval reliably indicates weak input power. SpecIC calibrates the power-off interval threshold through application profiling on the target platform—a practical approach commonly adopted in embedded systems. Specifically, we run applications under the weakest energy-harvesting environment, e.g., the bad power condition in Section VI-A, while measuring the power-off duration per outage. SpecIC then uses the average outage duration observed under the weakest condition as the long-outage threshold. However, directly measuring the power-off duration at run time would require persistent timer or peripheral support, causing significant energy overhead in intermittent systems. To address this, we use SRAM retention time as a near-zero-overhead proxy for the threshold. Accordingly, SpecIC monitors the duration of each outage by leveraging intrinsic SRAM retention time—a method far cheaper than timer interrupts. When an outage occurs, SpecIC writes a canary value to a specific SRAM location. At reboot time, if the read-back canary value is detected as modified or corrupted, signifying a long outage, then SpecIC immediately discards its trained prediction state and resets the **predicted-safe window** to a conservative value. This prevents the system from incorrectly carrying over predictions trained under earlier strong power conditions.

Again, under good power conditions, the short outage following region $r15$ does not affect SpecIC's speculation, as shown in Figure 4. However, when the sudden degradation occurs after region $r75$, the long power-off interval triggers the fast-drop policy, causing SpecIC to immediately stop aggressive speculation and thereby minimize further misspeculations. Combined with its lightweight execution model, SpecIC makes substantially more forward progress than Chinchilla in this scenario. By the time Chinchilla finally completes its first checkpoint after multiple outages, SpecIC has already advanced far ahead—reaching region $r100$.

As quantified using real-board measurements in Section VI-C, sudden power degradation poses a significant challenge to Chinchilla, leading to notable performance drops. In contrast, the fast-drop policy allows SpecIC to maintain robust adaptation, even under such fluctuating power conditions.

### B. Idempotent Recovery

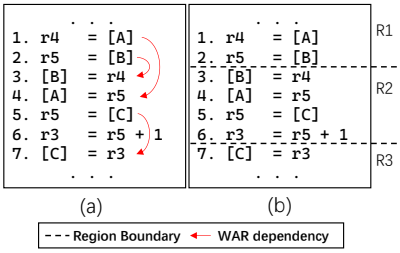



Fig. 5: (a) Code with WAR dependencies; (b) its idempotent regions

Unlike the "always-wait" strategy for ensuring no in-region outages (and thus no reexecutions), SpecIC can speculatively start the execution of a region without waiting—when guided by its predictor—to improve performance. However, this introduces the risk of *misspeculation*, i.e., the region is actually power-interrupted. In such cases, SpecIC must reexecute the interrupted region, which may lead to memory inconsistency. To address this, SpecIC adopts idempotent recovery, a technique shown to be significantly more lightweight than Chinchilla's undo logging [18], [9]. Idempotent recovery requires that every region contain no WAR dependencies [42], [45], ensuring that its reexecution has no side effects, assuming volatile registers are checkpointed/restored across power failures. SpecIC enforces idempotence by inserting region boundaries to break all WAR dependencies in the program. Figure 5(b) shows this process: by inserting a boundary between a read and a subsequent write, the resulting regions have no WAR dependencies, yielding safe reexecution.

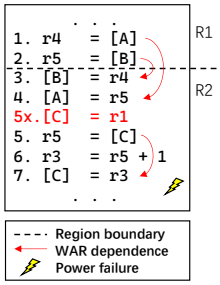



Fig. 6: WARAW dependencies need not be cut.

Care must be taken with WARAW (Write-After-Read-After-Write), as the WAR dependency is hidden by a preceding write to the same memory location within a region. As shown in Figure 6, while a read at line 5 and a write at line 7 form a WAR dependency on address $[C]$, they are preceded by a write to $[C]$ at line 5x. If a power failure occurs and region R2 is reexecuted later, the read at line 5 will observe the value written by line 5x anyway—rather than the value written at line 7 before the failure. Thus, the memory state remains consistent. SpecIC identifies such WARAW dependencies and avoids inserting unnecessary boundaries, preserving longer regions and thus resulting in fewer register checkpoints.

### C. PFI-enforced Idempotent Regions

To simultaneously guarantee stagnation freedom and efficient recovery, the SpecIC compiler partitions the input program into a series of regions that satisfy two constraints: (1) **SAT-safety** (for PFI) and (2) **Idempotence** (for recovery). Since these constraints are largely orthogonal, SpecIC can technically apply them in any order. However, the order of

partitioning affects the final number of resulting regions, which impacts performance. To minimize region count, SpecIC implements two partitioning strategies—**PFI First** and **Idempotence First**—and selects the optimal one for each program.

**PFI First** begins by partitioning the program into SAT-safe regions using RockClimb's algorithm [16]. It then scans these PFI regions for WAR dependencies. If a WAR dependency is found, SpecIC inserts an additional boundary to cut it. Since splitting a region only reduces its execution time, the resulting idempotent regions remain SAT-safe. Thus, this strategy guarantees that all final regions satisfy both constraints.

**Idempotence First** initially partitions the program to remove all WAR dependencies, forming a set of idempotent regions. It then checks the WCET of these regions. If a region exceeds the SAT limit, violating Equation 1, SpecIC further partitions it into smaller SAT-safe regions. While generally effective, this re-partitioning can end up breaking idempotence. Notably, if a cut is made between a write and a subsequent WAR dependency, thereby splitting a WARAW chain as shown in Figure 6, then the latter region may lose its idempotence because the preceding write that previously masked the WAR dependency is no longer in the same region. Although such cases are rare, SpecIC performs a final verification pass to detect and correct any resulting violations.

### D. Checkpoint Pruning

While optimal region formation minimizes the number of regions, the dual constraints of PFI and idempotence inevitably result in a higher region count compared to systems that enforce only one property. More regions imply more frequent register checkpoints, which can become a performance bottleneck. To mitigate this, SpecIC adopts *distributed checkpointing* [16] as a baseline. Unlike traditional approaches such as Ratchet and WARio that snapshot the entire register file at region boundaries (thus called *boundary checkpointing*), distributed checkpointing only saves a register when it is updated and live-out to some following region(s), significantly reducing NVM write traffic/energy.

However, SpecIC takes a further step by integrating *checkpoint pruning* [21]—originally designed for GPU soft error recovery—on top of distributed checkpointing. The core insight is that even live-out registers often do not need to be saved to NVM. Instead of checkpointing a volatile register value at run time, SpecIC recomputes it during recovery by reexecuting the instruction sequence generating it—akin to its backward slice [47] and thus called a *recovery slice*.

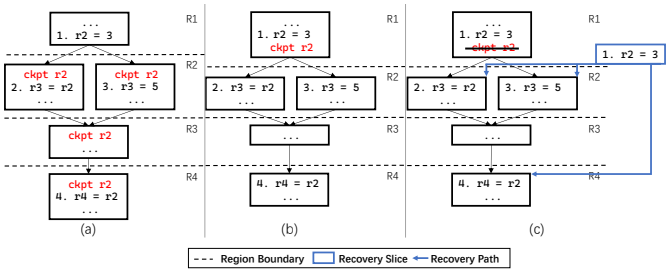



Fig. 7: (a) Boundary checkpointing; (b) Distributed checkpointing; (c) Checkpoint pruning and recovery slice.

Figure 7 illustrates the evolution of these schemes. Boundary checkpointing (Figure 7(a)) checkpoints all live-in registers ($r1, r2$) while distributed checkpointing (Figure 7(b)) optimizes this by only checkpointing $r2$ right after its definition. In contrast, checkpoint pruning (Figure 7(c)) eliminates $r2$'s checkpoints entirely. If any region R1, R2, or R3 is power-interrupted in the figure, SpecIC simply executes a write at line 1—as $r2$'s recovery slice for recomputing its value—before reexecuting the interrupted region for recovery. To realize this, SpecIC adapts an optimal checkpoint pruning compiler backend [21]—built for Nvidia GPUs—to Texas Instruments' MSP430 for SpecIC to efficiently identify removable checkpoints in its PFI-enforced idempotent regions and construct their recovery slices.

Note that checkpoint pruning introduces a fundamental trade-off: it reduces run-time overhead (fewer NVM writes) but increases the computational cost of recovery (due to recomputation). In systems with frequent outages, the increased recovery cost could negate the benefits. Crucially, SpecIC's power failure predictor (Section III-A) resolves this tension. By accurately predicting failure and avoiding misspeculation under bad power conditions, SpecIC ensures that recovery—and thus the cost of recomputation—is rare. This synergy allows SpecIC to aggressively prune checkpoints for maximum efficiency without suffering from high recovery penalties. Details are deferred to Section IV, while Section IV-A3 offers our improved recovery slice mapping over the original one.

## IV. Implementation

SpecIC consists of compiler and runtime support. The compiler forms PFI-enforced idempotent regions with register checkpoints inserted and generates per-region recovery slices. The runtime system handles power failure prediction and the recovery process. SpecIC is transparent to programmers; they only need to compile their code—as is—with SpecIC's compiler and feed the resulting binary to the runtime system.

### A. The SpecIC Compiler

SpecIC's compiler is built on top of LLVM [48]. With LLVM frontend support, the SpecIC compiler supports many high-level programming languages. Five major backend passes are added. First, the region formation pass partitions the program into PFI-enforced idempotent regions. Second, the distributed checkpointing pass identifies the definition points of all live registers and inserts checkpoint stores to save their values in NVM. Third, the checkpoint pruning pass utilizes an optimal checkpoint pruning algorithm to remove all checkpoints that can be safely recomputed at reboot time. Fourth, the recovery slice generation pass takes the results of all preceding passes and generates recovery slices for all regions. Finally, a verifier pass ensures that every region still meets all the constraints. Since PFI region formation requires energy information for each instruction to calculate WCET, our compiler passes are placed just before the emission of assembly code.

*1) Region Formation, I/O, Distributed Checkpointing:* Our region formation integrates both idempotent processing [45] and PFI enforcement [16]. Initial region boundaries are inserted at all call sites and loop headers [45], [16]. To cut all WAR dependencies, minimal additional boundaries are inserted using the Hitting-Set algorithm [45]. For PFI, we traverse each region's control-flow graph (CFG) while accumulating the WCET of instructions over each CFG path. If the accumulated WCET exceeds the SAT (Equation 1), the compiler inserts a region boundary to ensure the preceding portion remains SAT-safe and then continues processing the newly formed region. In particular, SpecIC treats each I/O and peripheral operation as a single region. SpecIC's compiler already inserts region boundaries around function calls, which covers most I/O operations. At run time, SpecIC ensures a full energy buffer for I/O regions to prevent outages. Once each region is formed, SpecIC identifies the last-update points of its live values—via DFS (depth-first search) of the CFG—and inserts their checkpoints. Further details are provided in [16].

*2) Checkpoint Pruning and Recovery Slice Building:* SpecIC's compiler implements the optimal checkpoint pruning algorithm [21], which operates in 2 phases. In the first phase, the compiler identifies checkpoints with dependency chains by traversing the use-def chain of each register checkpoint. If there is a self-loop, the checkpoint is marked INVALID for pruning, while a chain ending at a constant is marked VALID. For interdependent checkpoints, the second phase constructs a decision dependency graph and applies Tarjan's algorithm [49] to identify strongly connected components, that are then traversed to determine the final pruning decisions.

After pruning, the compiler builds a recovery slice for each region to regenerate pruned live-in checkpoint values. For pruned checkpoints, the slice contains their dependent instructions already identified during pruning; it does load instructions for committed checkpoints. Each recovery slice ends with a jump to the region entry for idempotent recovery.

Because a recovery slice contains only the instructions needed to reconstruct pruned live-in values, it is typically much smaller than the full region. During region formation (Section IV-A1), SpecIC already reserves space for the recovery slice and ensures that the combined size of a region and its slice still satisfies the PFI constraint. If a slice exceeds the reserved budget, the compiler selectively retains checkpoints until the resulting slice fits within the constraint.

*3) Lightweight Recovery Slice Mapping:* Prior checkpoint pruning [21], [23], [22]—devised for soft error recovery—typically stores the program counter at each region boundary and consults a map to locate the corresponding recovery slice during recovery [21], [22]. While simple, this original mapping scheme incurs extra memory accesses and computation during recovery. Unfortunately, this is particularly problematic for intermittent systems due to their frequent (power) failures—unlike soft errors, whose recovery overhead is negligible given their rarity. Also, the NVM accesses involved in the map operations are costly due to their energy overhead [50].

To eliminate this recovery-time overhead, SpecIC assigns each recovery slice a unique label and checkpoints the label of the next region's recovery slice into a dedicated NVM location at every region boundary. The label address is resolved during linking [51]. After a power failure, SpecIC's recovery runtime simply loads this address from NVM into the program counter register and jumps directly to the correct recovery slice.

### B. Runtime System

We implemented SpecIC's runtime for TI MSP430 and Arm Cortex-M23. The assembly code generated by SpecIC's compiler is integrated into the runtime. It deals with execution control, the recovery procedure, and power failure prediction.

**Program Execution Controller:** The program execution controller has two tasks. First, it uses a flag stored in NVM to indicate whether the program has started and is responsible for starting the recovery procedure in the wake of a power failure. Second, at each region boundary, the controller needs to determine whether to proceed or wait for capacitor recharging based on the predictor result.

*1) Recovery Procedure:* Because of SpecIC's lightweight recovery slice mapping, the recovery procedure is simple and fast. In the wake of a power failure, SpecIC jumps to the recovery slice directly by loading the checkpointed label address. The slice reconstructs the values of pruned live-in checkpoints and resumes execution from the beginning of the interrupted region for recovery. Recall that SpecIC's WCET analysis already considers the recovery slice, and SpecIC guarantees a full energy buffer before a recovery. As a result, the execution of both the recovery slice and the corresponding region can never be power-interrupted at reboot time, guaranteeing correct power failure recovery.

*2) Power Failure Predictor:* Without any hardware support, intermittent systems can usually gather very little information. Moreover, their predictor must be extremely simple, low-power, and fast—as it is triggered across frequent outages—which would otherwise eat up hard-won energy. To this end, SpecIC adjusts the **predicted-safe window** at each region boundary based on simple heuristics: (1) Addition: at a region boundary, if the value of the region counter exceeds the predicted-safe window, SpecIC expands the window by 2. (2) Doubling: when the counter value exceeds the window size, SpecIC doubles the window size. (3) Aggressive Addition: it is Addition, but each time the window size is increased by 50. For all heuristics, the window size is halved after an outage. Section VI-E shows that Aggressive Addition performs the best in most scenarios and is chosen for SpecIC in evaluation.

## V. Discussion

### A. Predictor Convergence Property and Error Bounds

This section analyzes the convergence behavior and error bounds of SpecIC's predicted-safe window under the Aggressive Addition heuristic. Suppose the power condition changes and the predictor should adapt to the new condition. Let $W_{oracle}$ denote the oracle safe window under the new power condition, i.e., the maximum number of consecutive regions that can be executed without waiting before a power failure would occur; it is unknown to SpecIC and used only as an analytical reference. Let $W_{old}$ be the predictor state carried over

from before the condition change. Let $W_t$ be the predicted-safe window after the $t$-th update, with $W_0 = W_{old}$, and let $A$ be the additive step size used by Aggressive Addition.

The update rule of the heuristic follows the additive-increase/multiplicative-decrease (AIMD) [52] pattern below:

$$W_{t+1} = \begin{cases} W_t + A, & \text{if } W_t \leq W_{oracle}, \\ \lceil W_t/2 \rceil, & \text{if } W_t > W_{oracle}. \end{cases}$$

The first case corresponds to a conservative prediction (SpecIC reaches the predicted-safe window without failure and thus increases the window to probe for more speculative execution), while the second corresponds to an over-optimistic prediction (SpecIC skips too long, suffers misspeculation, and halves the window).

If the power condition improves, $W_{old}$ can be smaller than the new $W_{oracle}$. In this case, SpecIC is conservative and increases the window linearly by $A$. Starting from $W_{old}$, the predictor reaches or exceeds $W_{oracle}$ after at most

$$\left\lceil \frac{W_{oracle} - W_{old}}{A} \right\rceil$$

successful window expirations. The transient cost in this direction is unnecessary waiting, but no misspeculation is introduced here by being conservative.

Conversely, if the condition degrades, $W_{old}$ can be larger than the new $W_{oracle}$. In this case, misspeculation triggers multiplicative decrease. Without fast-drop, after at most

$$\left\lceil \log_2 \frac{W_{old}}{W_{oracle}} \right\rceil$$

misspeculation-triggered shrinks, the predicted-safe window gets no larger than $W_{oracle}$. Notably, SpecIC's fast-drop shortens this degradation: when a long outage implies an abrupt power degradation, SpecIC discards the stale optimistic window and resets it to a conservative value, reducing the shrink phase from logarithmic time to $O(1)$ reset steps.

Therefore, after a finite adaptation period under the new power condition, the predicted-safe window repeatedly operates within the bounded band:

$$\frac{W_{oracle}}{2} < W_t \leq W_{oracle} + A.$$

This band also bounds the predictor-state error. The optimistic error is at most $A$, because the predictor can overshoot $W_{oracle}$ by only one additive step before misspeculation is observed. The conservative error is bounded by $W_{oracle}/2$, because the final multiplicative decrease that crosses below $W_{oracle}$ leaves the window larger than half of $W_{oracle}$.

Empirical results show that $A$=50 offers the best tradeoff: it limits prediction errors, maintains sufficiently fast expansion, and keeps the predictor within a well-bounded operating range.

### *B. System-Level Worst-Case Efficiency Bound*

The bounded predictor error above characterizes the predictor state; we next show that, because every region is PFI-enforced and idempotent, even worst-case prediction outcomes incur bounded system-level overhead. We first define the total work the program should complete. For program $P$ and its region execution sequence $S$, let $w(r)$ be the work of region $r$. Then, the total work $W_{total}$ is defined as

$$W_{total} = \sum_{r \in S} w(r)$$

This $W_{total}$ represents the lowest amount of work required to complete the program, i.e., executing each region in the execution sequence $S$ exactly once. However, SpecIC has its own overhead, so its work is inherently greater than $W_{total}$. With $W_{SpecIC}$ denoting the work required by SpecIC, we define SpecIC's efficiency as

$$E_{SpecIC} = \frac{W_{total}}{W_{SpecIC}} < 1 \tag{2}$$

We now dissect SpecIC's per-region overhead. For region $r$, let $o^{pred}$ be the predictor overhead for predicted-window size comparison, $o_r^{wait}$ be the overhead of waiting for recharging, and $o_r^{mis}$ be the overhead of misspeculation. Although the predictor overhead $o^{pred}$ is paid for every region, it is fixed and very low due to SpecIC's simplistic design. The waiting overhead $o_r^{wait}$ covers both waiting and wait-state transition, while the misspeculation overhead $o_r^{mis}$ includes wasted partial execution, power-off time, and recovery-slice execution.

At each region $r$, SpecIC can either choose to wait or skip. For waiting, the overhead is $o^{pred} + o_r^{wait}$. For skipping, there are two possible outcomes:

- If no power failure occurs, the overhead is only $o^{pred}$
- If a power failure occurs, the overhead is $o^{pred} + o_r^{mis}$

The worst case occurs when SpecIC pays the larger of the two avoidable costs: unnecessary waiting or misspeculation. This per-region bound is sufficient because SpecIC's PFI region formation ensures at most a single in-region outage; if region $r$'s speculative execution is interrupted, SpecIC reboots with a full capacitor, executes the recovery slice, and reexecutes $r$. Since the compiler accounts for the recovery slice in the PFI constraint, this recovery path is guaranteed to complete with no outage. Thus, each dynamic region execution can incur at most one waiting or misspeculation cost before making progress.

For each region $r$, we thus define the worst-case overhead besides fixed $o^{pred}$ as

$$o_r^{max} = \max(o_r^{wait}, o_r^{mis}).$$

Therefore, the total work required by SpecIC is bounded by

$$W_{SpecIC} \leq W_{total} + \sum_{r \in S} \left( o^{pred} + o_r^{max} \right). \tag{3}$$

Combining this bound with Equation 2, SpecIC's efficiency is lower-bounded by

$$E_{SpecIC} \geq \frac{W_{total}}{W_{total} + \sum_{r \in S} \left( o^{pred} + o_r^{max} \right)}.$$

### *C. Expected Efficiency Under Prediction Errors*

While the worst-case bound above assumes that every region incurs the larger of the waiting and misspeculation costs, SpecIC's predictor often allows for avoiding both unnecessary costs. We therefore refine the analysis by modeling the expected overhead induced by prediction outcomes.

At each region boundary, the next region is either *safe*, i.e., SpecIC can skip waiting and execute it without failure, or *unsafe*, i.e., skipping waiting would cause misspeculation. Let $P_r^{unsafe}$ be the probability that region $r$ is unsafe. The predictor can make two types of mistakes:

- **false-wait**: the region is safe, but SpecIC chooses to wait;
- **false-skip**: the region is unsafe, but SpecIC chooses to skip and suffers misspeculation.

Let $F_r^{wait}$ be the probability of false-wait conditioned on the region being safe, and let $F_r^{skip}$ be the probability of false-skip conditioned on the region being unsafe. Since the predictor overhead $o^{pred}$ is paid at every region boundary, the expected total work of SpecIC is defined as

$$\mathbb{E}[W_{SpecIC}] = W_{total} + \sum_{r\in S} o^{pred} + \sum_{r\in S}\Big[P_r^{unsafe}\big(F_r^{skip}o_r^{mis} + (1-F_r^{skip})o_r^{wait}\big) + (1-P_r^{unsafe})F_r^{wait}o_r^{wait}\Big]. \tag{4}$$

The corresponding efficiency based on expected work is

$$\bar{E}_{SpecIC} = \frac{W_{total}}{\mathbb{E}[W_{SpecIC}]}.$$

Equation 4 captures the tradeoff among predictor heuristics. A conservative heuristic reduces false-skips lowering the $P_r^{unsafe}F_r^{skip}o_r^{mis}$ term, but it increases false-waits raising the $(1-P_r^{unsafe})F_r^{wait}o_r^{wait}$ term. An aggressive heuristic has the opposite effect. Aggressive Addition balances this tradeoff by increasing the window fast enough to avoid repeated unnecessary waiting, while limiting the optimistic overshoot as shown in the convergence bound earlier. This analytical tradeoff is consistent with the empirical comparison in Section VI-E.

## VI. Evaluation

### A. Experimental Setting

We conducted our experiments using a TI MSP430FR5994 evaluation board featuring a $10\mu F$ capacitor as the energy buffer, with a frequency of 16MHz—a configuration commonly used in prior studies [18], [15], [14], [16]. We also varied the capacitor setting for sensitivity analysis (Section VI-I). A Powercast P2110-EVB RF board was used with a patch antenna connected to our test platform for energy harvesting, and the platform was wirelessly powered using a Powercast TX91503 915MHz RF transmitter. To collect execution time more accurately without interfering with program execution, we used a Raspberry Pi to externally control the test board and measure execution time through GPIO signals. This approach avoids relying on an on-board internal timer module—that consumes significant power and interferes with execution. We evaluated SpecIC using 10 compute-intensive benchmarks that have been widely used in prior studies [16], [14], [18]. Additionally, we included 3 Deep Neural Network (DNN) kernels—specifically convolution, pooling, and ReLU—to reflect the increasing deployment of inference tasks on intermittent systems. They encompass most computational tasks that an intermittent system targets.

We tested SpecIC against four prior schemes: Chinchilla [15], RockClimb [16], WARio [9] and EarlyBird [17]. We ported WARio to MSP430. For Chinchilla, since our target platform uses NVM as main memory, it does not need to checkpoint the volatile stack at region boundaries. To evaluate the performance of these schemes across a wide dynamic range of power conditions, we varied the distance between the energy harvester and the RF transmitter. We selected three distances: 20cm, 50cm, and 80cm to represent good, moderate, and bad power conditions, respectively. To reflect weaker signal strengths within a compact experimental setup, we oriented the transmitter sideways. Under good conditions, power outages are negligible. In contrast, the bad condition (80cm) represents an extreme scenario where the system suffers from frequent power failures, potentially reaching hundreds per second depending on the tested workload.

Crucially, this distance pushes the system to its operational limit; further distance renders the energy harvesting board unusable, as it fails to execute the firmware bootstrap code to reach user code. We did not vary the distance during the execution, as it is practically impossible to ensure the same movement across experiments. Nonetheless, to assess SpecIC's performance in a more realistic and dynamic environment, we modeled dynamic transitions among power conditions using Markov chains (details are deferred to Section VI-C).

### B. Performance

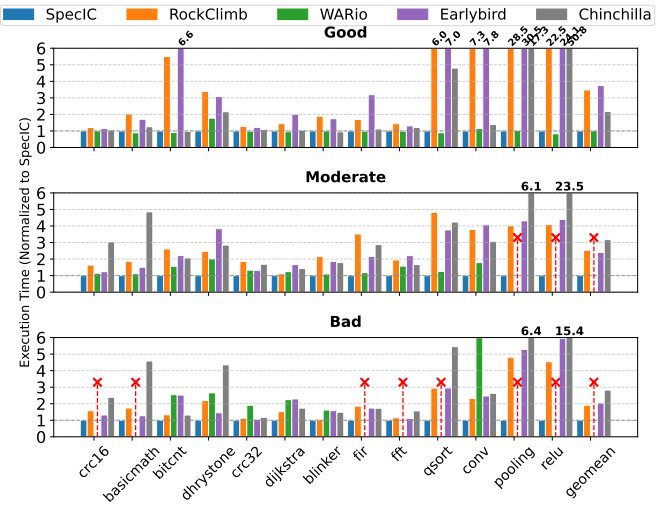



Fig. 8: Normalized execution time under various power conditions

Figure 8 presents the evaluation results for all 5 schemes under different power conditions. SpecIC performs the best under all power conditions. Under good power conditions, RockClimb unnecessarily waits at every region boundary for charging. As a result, RockClimb is 3.47x slower than SpecIC. Especially for *pooling* and *relu*, which have many regions, the performance degradation is significant, up to 28x slower than SpecIC. As power conditions deteriorate, the benefit of waiting at boundaries shines thanks to the complete removal of in-region power outages. Under moderate power conditions, RockClimb is 2.53x slower than SpecIC, while under bad conditions, RockClimb is 1.92x slower than SpecIC. It turns

out that even under bad conditions, waiting at all boundaries is still overkill, and SpecIC shows noticeable speedup.

While EarlyBird can reduce the waiting time as it does not always fully charge a capacitor as RockClimb does, it shows similar performance to RockClimb That is because its all register checkpointing at region boundaries incurs much more overhead compared to RockClimb's distributed checkpointing. Overall, EarlyBird is 3.74x, 2.4x and 2x slower than SpecIC under good, moderate and bad conditions, respectively.

WARio performs well under good power conditions—only 0.016x slower than SpecIC. For some benchmarks, WARio is even faster than SpecIC owing to its much fewer regions. However, this comes at the cost of stagnation when power conditions worsen. Under moderate power conditions, *pooling* and *relu* start to stagnate, as WARio takes no checkpoints for them; they must be completed in one power cycle. Under bad power conditions, WARio stagnates in many applications. For the rest, it also performs worse than all other schemes, since it reexecutes each region many times across frequent outages.

Chinchilla adaptively skips checkpointing under good conditions, while avoiding redundant undo logs for the same addresses, so it performs much better than RockClimb. However, for *basicmath*, it has 4.5x more regions than SpecIC; for *dhrystone* and *qsort*, the unavoidable undo logs are still many, leading to performance degradation. Particularly, *relu* demonstrates the extreme case of undo loggings. Since all store instructions write to different addresses, there is no redundant undo log to be avoided. On the other hand, as power conditions deteriorate, more outages occur, which results in unskippable checkpoints—avoiding less undo logs—and more reexecutions. Thus, the performance degrades significantly in that both checkpointing and undo logging are expensive. Overall, Chinchilla is 3.17x slower than SpecIC under moderate power conditions and is 2.82x slower under bad conditions.

### C. Dynamic Power Conditions Emulation

As stated in Section II-C, real-world intermittent systems operate under continuously fluctuating energy harvesting conditions. Returning to the smart wristband example, as a user moves through a gym with sporadically placed RF sources, the device experiences a highly dynamic power spectrum. Nonetheless, prior evaluations rely on static, fixed-distance setups, masking the performance cliffs that arise under mobility: schemes optimized for strong power, e.g., WARio and Chinchilla, can stagnate if the user enters low-RF-signal areas, while conservative schemes, e.g., RockClimb and EarlyBird, underutilize harvested energy—by waiting unnecessarily—once the user returns to energy-rich zones. To accurately reflect this variability and demonstrate SpecIC's ability to sustain high performance across the full power spectrum, particularly during condition transitions, we developed an emulation infrastructure based on Markov chains [24].

We model the environment as a three-state Markov process representing GOOD (20cm), MODERATE (50cm), and BAD (80cm) power conditions. Transitions among the three states are probabilistic, capturing the temporal dynamics of physical mobility, e.g., a user moving in and out of RF coverage. State changes are governed by a predefined transition probability matrix, reflecting realistic movement patterns.

*1) Task Execution Model:* We drive the emulation using performance profiles from real-board measurements. For each benchmark and scheme, we model task completion time as a Gaussian distribution based on the mean and standard deviation measured on our hardware setup (Section VI-A) under steady-state conditions. When a task executes entirely within a single power state, its duration is sampled directly from the corresponding distribution. However, when a state transition occurs during task execution, we adopt a *virtual work progress* model to maintain continuity. Specifically, if a task is $p\%$ complete in the initial state, e.g., GOOD, at the time of transition, the remaining $(1-p)\%$ of the workload is projected onto the performance profile of the new state, e.g., BAD. This ensures that the simulated execution time scales proportionally with the varying energy availability.

*2) Modeling Adaptation Overhead:* While the virtual work progress model captures the change in intrinsic execution speed, simple rescaling alone cannot capture the *adaptation overhead* incurred when an intermittent computing scheme must react to an abrupt change in power conditions.

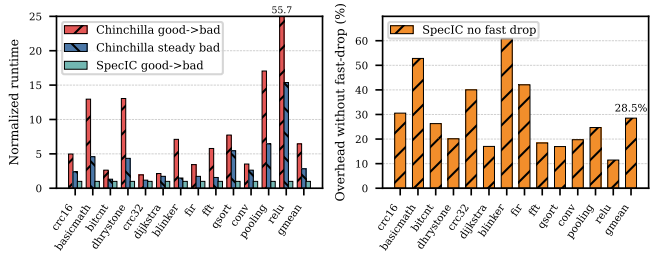



Fig. 9: (a) Chinchilla transition penalty (b) SpecIC without fast-drop.

As shown in Figure 4, sudden GOOD→BAD power degradation is particularly harmful to adaptive schemes that carry stale optimistic state across a prolonged outage. Figure 9 quantifies this effect on the real board. The left subplot shows that under a GOOD→BAD transition, Chinchilla suffers a large adaptation penalty, incurring a 2.28× slowdown over a steady-BAD power condition. In contrast, SpecIC maintains high performance, yielding more than a 6x speedup over Chinchilla—mainly due to its lightweight yet efficient intermittent computation. The right subplot then isolates the effect of fast-drop: disabling it increases run time by 28.5% under the same transition, showing its necessity and effectiveness.

To incorporate this effect into the Markov emulation, we profile transition-specific overhead on the board. For each adaptive scheme, we execute every benchmark under GOOD power until its internal adaptation state stabilizes in its most optimistic configuration and then switch to BAD power while retaining that internal state. The transition overhead, performance penalty relative to a steady-state BAD execution, is injected at the corresponding Markov state transition.

*3) Emulation Results:* To stress-test each intermittent computing scheme, we defined three distinct transition matrices:

- **Skew to GOOD**: The scheme spends 80% of its time in the GOOD state, reflecting a stable, energy-rich environment with occasional interruptions.

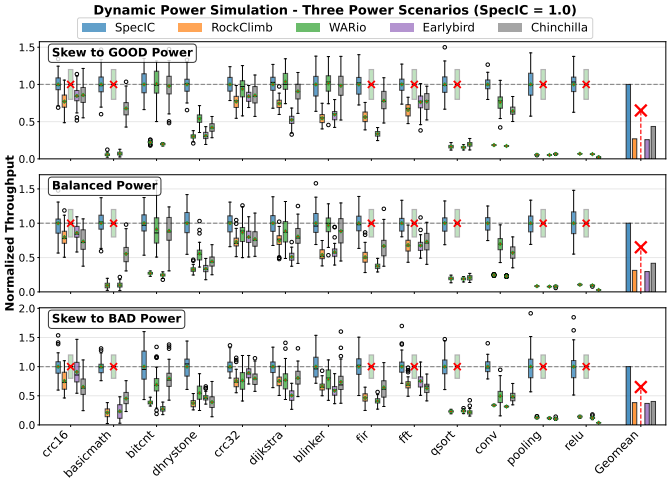



Fig. 10: Normalized throughput comparison under dynamic power conditions modeled by Markov chains (higher is better). Results are normalized to SpecIC. Red crosses indicate benchmarks that suffered from stagnation.

- **Skew to BAD**: The scheme spends 80% of its time in the BAD state, testing resilience in energy-scarce scenarios.
- **Balanced**: The scheme has uniform state probabilities.

Figure 10 presents the throughput comparison of SpecIC against Chinchilla, RockClimb, WARio, and EarlyBird under these dynamic power scenarios. Overall, SpecIC demonstrates superior robustness and efficiency, consistently achieving the highest throughput across the full power spectrum.

WARio stagnates in all three scenarios despite being competitive under steady GOOD power, confirming that performance without forward-progress guarantee is unreliable in dynamic environments. For the affected benchmarks, these stagnation events reduce throughput to zero regardless of WARio's efficiency during favorable intervals. RockClimb and EarlyBird avoid stagnation, but their always-wait behavior prevents them from exploiting the intervals in which tasks can make the fastest progress. GOOD intervals offer the greatest opportunity to complete work quickly, so waiting conservatively throughout these intervals substantially suppresses overall throughput. Even in the *Skew to BAD* scenario, which favors conservative execution, they achieve only ∼37% of SpecIC's throughput and trail Chinchilla by ∼40%. While Chinchilla performs better than these non-adaptive schemes, it significantly underperforms SpecIC. This overall gap arises mainly from SpecIC's lightweight execution and recovery model that avoids Chinchilla's expensive undo logging, checkpointing and large rollback penalty; fast-drop offers an additional benefit by discarding stale optimistic predictor state after a long outage, thereby reducing transition-induced reexecution penalties.

### D. Performance Breakdown

To better understand the performance of SpecIC, we collected the performance breakdown of SpecIC and RockClimb under all power conditions as shown in Figure 12. The runtime is segmented into three distinct stages: Run (green), Wait (yellow), and Failure (red). To collect the data, we let the runtime system signal the controller each time the system changes its stage, i.e., starts waiting or resumes execution. The runtime system also maintains an active low signal for power failure detection. Note that all these operations incur overhead, so the execution time in this breakdown will deviate from the runtime without these probing signals.

The analysis demonstrates that the predominant source of overhead in RockClimb is its waiting time. Under good power conditions, SpecIC experiences virtually no waiting time, resulting in a substantial reduction in overall runtime overhead. As power conditions worsen, both SpecIC and RockClimb encounter more power failures. It is obvious that the failure time is much shorter for RockClimb than for SpecIC, demonstrating the effectiveness of RockClimb's waiting strategy in preventing in-region power failures.

However, the drawback is the much longer waiting time. For SpecIC, failure time becomes a significant portion of the total execution time under moderate and bad power conditions, but this does not indicate poor performance. On the contrary, SpecIC effectively leverages Power Failure Immunity (PFI), minimizing frequent reexecutions even in bad conditions. With the help of SpecIC's power failure predictor, power failures, though prolonged when they occur, are infrequent. Combined with the fact that after a power failure, SpecIC resumes execution with a fully charged energy buffer, ensuring successful reexecution, SpecIC avoids repetitive reexecutions. The absence of a significant increase in run time further supports this. For SpecIC, the failure time is the inevitable charging time when the energy buffer is near empty, and the only wasted time is the several interrupted regions.

One interesting case is *dijkstra*. Under good power conditions, SpecIC shows a noticeably longer run time than RockClimb. Given that SpecIC does not have any power failures and reexecutions, this is due to the overhead of the power failure predictor, which is included in the run time. SpecIC's region count for *dijkstra* is 21x higher than RockClimb's. While the predictor is designed to be lightweight, the overhead of the predictor is still noticeable when the region count is so high. This highlights the importance of the simple predictor design.

### E. Predictor Heuristics Comparison

Figure 13 compares the three predictor heuristics using runtime overhead relative to Aggressive Addition. Addition is the most conservative heuristic and performs the worst, incurring 34.5% geomean overhead. Because it expands the predicted-safe window too slowly, the system spends substantial time waiting before reaching a useful speculation level. Doubling performs better, but still incurs 8.6% geomean overhead. Although it expands faster than Addition, its growth is unstable: it is too cautious at small window sizes yet becomes overly optimistic once the window grows. Aggressive Addition performs the best because it ramps up speculation quickly without the instability of Doubling.

Moreover, due to the lack of a HW multiplier and the low-performance nature of the MSP430, heuristics with more computation would incur excessive overhead. A more sophisticated heuristic may become attractive on a more powerful platform.

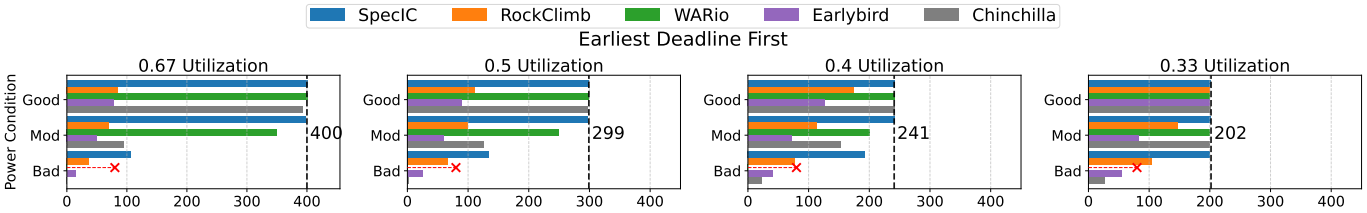



Fig. 11: Schedulability evaluation results with the EDF scheduling algorithm


Fig. 12: Performance breakdown

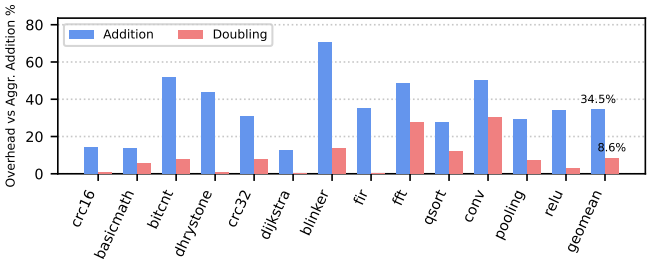



Fig. 13: Predictor heuristic comparison by overhead over Aggressive Addition.

### F. Checkpoint Pruning

To analyze the impact of checkpoint pruning, we compare SpecIC with/without pruning on top of distributed checkpointing. Figure 14(a) shows that checkpoint pruning eliminates 45% of checkpoint stores statically on average relative to distributed checkpointing. Figure 14(b) shows the end-to-end cost of disabling pruning. Without pruning, run time increases by 5.8% geomean, with overheads exceeding 10% for several benchmarks such as *bitcnt*, *dhrystone*, *fir* and *qsort*. This shows the checkpoint reduction largely translates to end-to-end benefit, though the magnitude varies across benchmarks.

### G. Schedulability Analysis

To evaluate SpecIC's compatibility with a soft real-time system and its task throughput, we test all 5 schemes with the EDF (earliest deadline first) scheduler [27]. The detailed settings are as follows. We utilize benchmark applications as tasks with each task set comprising four tasks—constrained

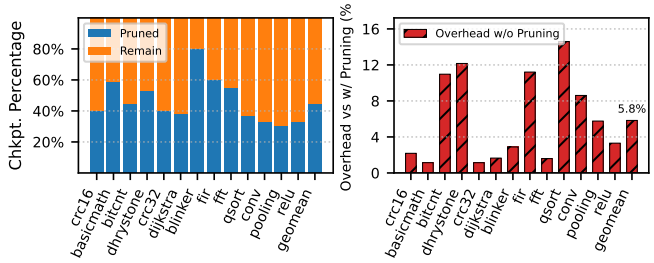



Fig. 14: (a) Static pruning result (b) Performance impact

by the memory capacity of the evaluation board. Also, the period for each task is set to its baseline no-failure WCET, scaled by a factor to simulate varying scheduling pressures, yielding utilization rates ranging from 0.67 to 0.33. Finally, task lengths are adjusted to ensure each task has a comparable WCET, promoting uniform job generation across the task set.

Figure 11 presents the schedulability results for all 5 schemes under varying power conditions and scheduling pressures, with the total number of jobs indicated by vertical gray dashed lines. Chinchilla and WARio demonstrate strong schedulability in good power conditions but experience a rapid decline as power conditions worsen. In particular, WARio exhibits no schedulability under bad power conditions due to its stagnation. RockClimb performs adequately in bad power conditions but exhibits significantly reduced schedulability in good conditions, while EarlyBird shows a similar trend to RockClimb. SpecIC, with its enhanced execution efficiency across all power conditions, consistently achieves the highest number of scheduled jobs in all scenarios.

### H. Recovery Slice Size Statistics

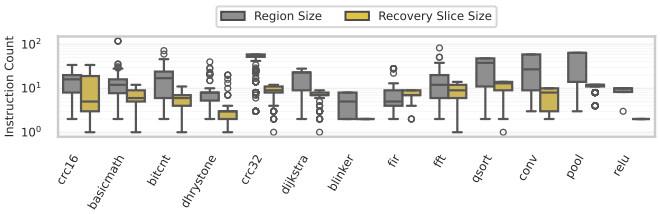



Fig. 15: Region size and recovery slice size comparison

Figure 15 shows the recovery slice size in comparison to the region size on logarithmic scale. Notably, the recovery slices of most applications are over 10 times smaller than their regions.

### I. Sensitivity Analysis of Capacitor Size

Figure 16 demonstrates that SpecIC operates effectively and efficiently across various capacitor sizes, maintaining consistently high performance under all power conditions. While

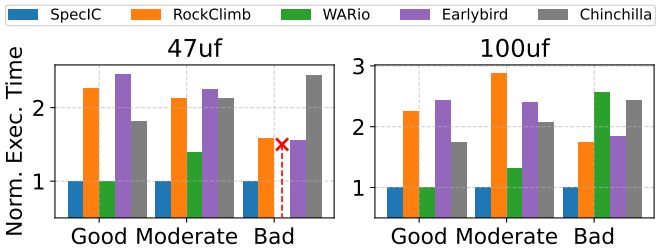



Fig. 16: Normalized execution time (Geomean) in different capacitor sizes

increasing capacitor size impacts each scheme differently—reducing the number of power failures, enlarging region sizes for RockClimb and EarlyBird, increasing recharging time, or creating more favorable working conditions for WARio and Chinchilla—these effects are beyond the scope of this paper.

### J. Sensitivity Analysis with Arm Architecture

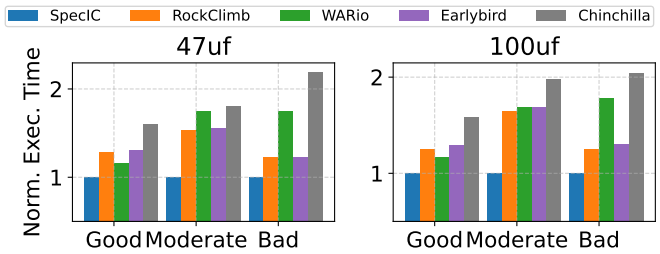



Fig. 17: Normalized execution time (geomean) on M2L31

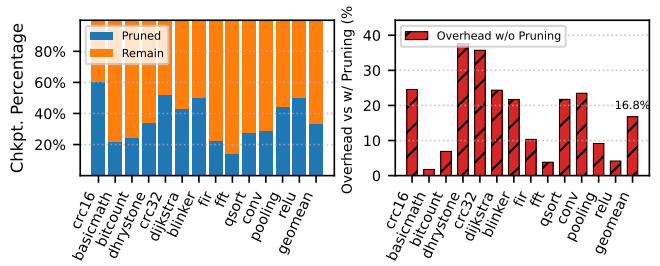



Fig. 18: (a) Static pruning result (b) Performance impact

We also conducted an architecture sensitivity analysis on an Arm platform to show the generality of SpecIC's approach. The experiments were performed using a Nuvoton NuMicro M2L31KIDAE evaluation board with a frequency of 72MHz. M2L31 is based on Arm Cortex-M23 core at Armv8-M microarchitecture, equipped with ReRAM. The other experiment configurations remain identical; the same energy harvester and RF transmitter were used and execution time was collected by a Raspberry Pi through GPIO signals. Due to the higher power consumption of the board than MSP430, we were only able to run with $47\mu F$ and $100\mu F$ reliably.

Figure 17 demonstrates that SpecIC remains effective on the Arm architecture as well. There are two noticeable differences. First, faster clock speed makes the transition from execution mode to waiting mode faster than on MSP430 (on top of the architecture difference), which reduces the overhead of SpecIC, RockClimb, and EarlyBird. RockClimb and EarlyBird show better performance under good conditions than when running on the MSP430. The benefit for SpecIC is smaller, though, as it already skips the majority of the waiting. Second, ReRAM is slower than FRAM in terms of memory write. Together with faster clock speed, each register checkpoint requires more cycles than on MSP430. As a result, WARio and Chinchilla both perform worse here because they incur many register checkpoints at every region boundary.

Figure 18 shows the impact of SpecIC's checkpoint pruning on M2L31. Because of the architectural differences including calling convention and register usage/allocation, the static pruning result is less effective than on MSP430. Nevertheless, the performance impact is much stronger due to the slow ReRAM writes and high clock speed. Without pruning, the total execution time increases by 16.8% on average, with 7 benchmarks experiencing over 20% overhead. These results highlight that SpecIC's checkpoint pruning can be even more beneficial on faster platforms—with slower, denser NVM.

## VII. Conclusion

This paper presents SpecIC, a software-only speculative intermittent computing scheme. Its lightweight yet effective power failure predictor enables high-performance speculative execution, while its fast-drop policy enhances robustness against sudden power degradation. SpecIC also leverages checkpoint pruning to reduce checkpoint overhead. As a result, SpecIC significantly outperforms prior work across varying power conditions, achieving up to 274% higher performance.

## VIII. Acknowledgments

This work was supported by NSF grants 2153749, 2314680, and 2314681. We used GPT for limited editing and refinement.

## References

[1] S. Ahmed, B. Islam, K. S. Yildirim, M. Zimmerling, P. Pawełczak, M. H. Alizai, B. Lucia, L. Mottola, J. Sorber, and J. Hester, "The internet of batteryless things," *Communications of the ACM*, 2024.

[2] Y.-W. Chong, W. Ismail, K. Ko, and C.-Y. Lee, "Energy harvesting for wearable devices: A review," in *IEEE Sensors Journal 19, 20*, 2019.

[3] V. Dsouza, J. Pronk, C. Peppelman, V. I. Madariaga, T. Pereira-Cenci, B. Loomans, and P. Pawełczak, "Densor: An intraoral battery-free sensing platform," *Proceedings of IMWUT*, 2024.

[4] S. Beeby and N. White, "Energy harvesting for autonomous systems," in *Artech House, Incorporated*, 2014.

[5] Y. Wu, B. Min, M. Ismail, W. Xiong, C. Jung, and D. Lee, "{IntOS}: Persistent embedded operating system and language support for multi-threaded intermittent computing," in *OSDI*, 2024.

[6] G. Fang and C. Jung, "Speculation under intermittence: A guide for computer architects," *Computer*, vol. 59, no. 8, 2026.

[7] G. Fang, J. Zeng, A. Gupta, and C. Jung, "Rethinking prefetching for intermittent computing," in *ISCA*, 2025.

[8] G. Fang and C. Jung, "Rethinking dead block prediction for intermittent computing," in *HPCA*, 2025.

[9] V. Kortbeek, J. Hester, S. Campanoni, and P. Pawełczak, "Wario: Efficient code generation for intermittent computing," in *PLDI*, 2022.

[10] 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, 2022.

[11] J. Choi, J. Choi, H. Joe, and C. Jung, "Caphammer: Exploiting capacitor vulnerability of energy harvesting systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2024.

[12] J. Choi, H. Joe, C. Jung, and J. Choi, "Defending against emi attacks on just-in-time checkpoint for resilient intermittent systems," in *IEEE/ACM International Symposium on Microarchitecture*, 2024.

[13] T. S. Pillai, V. Chidambaram, R. Alagappan, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Crash consistency," *Communications of the ACM*, vol. 58, no. 10, pp. 46–51, 2015.

[14] J. Choi, H. Joe, Y. Kim, and C. Jung, "Achieving stagnation-free intermittent computation with boundary-free adaptive execution," in *Real-Time & Embedded Technology and Applications Symposium*, 2019.

[15] K. Maeng and B. Lucia, "Adaptive dynamic checkpointing for safe efficient intermittent computing," in *OSDI 18*, 2018.

[16] J. Choi, L. Kittinger, Q. Liu, and C. Jung, "Compiler-directed high-performance intermittent computation with power failure immunity," in *Real-Time & Embedded Technology and Applications Symposium*, 2022.

[17] H. Reymond, M. Briday, S. Faucou, I. Puaut, and E. Rohou, "Earlybird: Energy belongs to those who wake up early," in *RTCSA*, 2024.

[18] J. Van Der Woude and M. Hicks, "Intermittent computation without hardware support or programmer intervention," in *OSDI*, 2016.

[19] W. S. Lim, C.-H. Tu, C.-F. Wu, and Y.-H. Chang, "icheck: Progressive checkpointing for intermittent systems," *IEEE TCAD*, 2020.

[20] 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 *IEEE/ACM International Symposium on Microarchitecture*, 2018.

[21] 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 *PLDI*, 2020.

[22] Y. Zhang and C. Jung, "Featherweight soft error resilience for gpus," in *2022 55th IEEE/ACM International Symposium on Microarchitecture*.

[23] Q. Liu, C. Jung, D. Lee, and D. Tiwari, "Compiler-directed lightweight checkpointing for fine-grained guaranteed soft error recovery," in *SC'16*.

[24] J. R. Norris, *Markov chains*. Cambridge university press, 1998.

[25] Z. Zhang, C. Liu, and H. Kim, "Mii: A multifaceted framework for intermittence-aware inference and scheduling," *IEEE TCAD*, 2024.

[26] M. Karimi, H. Choi, Y. Wang, Y. Xiang, and H. Kim, "Real-time task scheduling on intermittently powered batteryless devices," *IEEE Internet of Things Journal*, vol. 8, no. 17, pp. 13 328–13 342, 2021.

[27] S. K. Baruah, A. K. Mok, and L. E. Rosier, "Preemptively scheduling hard-real-time sporadic tasks on one processor," in *RTSS*, 1990.

[28] V. Chidambaram, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Optimistic crash consistency," in *SOSP*, 2013.

[29] S. Kalbfleisch, L. Werling, and F. Bellosa, "Vinter: Automatic {Non-Volatile} memory crash consistency testing for full systems," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022.

[30] G. Fang, J. Choi, and C. Jung, "Hybrid power failure recovery for intermittent computing," in *ICCAD*, 2024.

[31] N. Hassan, B. Min, C. Jung, Y. Solihin, and J. Choi, "Warmcache: Exploiting stt-ram cache for low-power intermittent systems," in *International Symposium on Computer Architecture*, 2025.

[32] Y. Zhou, J. Zeng, J. Jeong, J. Choi, and C. Jung, "Sweepcache: Intermittence-aware cache on the cheap," in *MICRO*, 2023.

[33] J. Zeng, J. Choi, X. Fu, A. P. Shreepathi, D. Lee, C. Min, and C. Jung, "Replaycache: Enabling volatile cachesfor energy harvesting systems," in *International Symposium on Microarchitecture*, 2021.

[34] M. Didehban, S. R. D. Lokam, and A. Shrivastava, "Incheck: An in-application recovery scheme for soft errors," in *DAC*, 2017.

[35] S.-Y. Huang, J. Zeng, X. Deng, S. Wang, A. Sifat, B. Bharmal, J.-B. Huang, R. Williams, H. Zeng, and C. Jung, "Rtailor: Parameterizing soft error resilience for mixed-criticality real-time systems," in *RTSS*, 2023.

[36] J. Zeng, S.-Y. Huang, J. Liu, and C. Jung, "Soft error resilience at near-zero cost," in *ICS*, 2024.

[37] J. Zeng, H. Kim, J. Lee, and C. Jung, "Turnpike: Lightweight soft error resilience for in-order cores," in *MICRO*, 2021.

[38] J. Jeong and C. Jung, "Pmem-spec: persistent memory speculation," in *ASPLOS*, 2021.

[39] J. Zeng, T. Zhang, and C. Jung, "Compiler-directed whole-system persistence," in *ISCA*, 2024.

[40] Y. Zhou, J. Zeng, and C. Jung, "Lightwsp: Whole-system persistence on the cheap," in *MICRO*, 2024.

[41] J. Zeng, J. Jeong, and C. Jung, "Persistent processor architecture," in *IEEE/ACM International Symposium on Microarchitecture*, 2023.

[42] S. Muchnick, *Advanced compiler design implementation*, 1997.

[43] A. Colin and B. Lucia, "Termination checking and task decomposition for task-based intermittent programs," in *CC*, 2018.

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

[45] M. A. De Kruijf, K. Sankaralingam, and S. Jha, "Static analysis and compiler design for idempotent processing," in *PLDI*, 2012.

[46] S. Chen and E. Rodriguez-Villegas, "An energy-aware, self-adaptive, battery-free smart wristband for long-term health monitoring," 2025.

[47] M. Weiser, "Program slicing," *IEEE Transactions on software engineering*, no. 4, pp. 352–357, 2009.

[48] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *CGO*, 2004.

[49] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM journal on computing*, vol. 1, no. 2, pp. 146–160, 1972.

[50] X. Zhang, C. Patterson, Y. Liu, C. Yang, C. J. Xue, and J. Hu, "Low overhead online data flow tracking for intermittently powered non-volatile fpgas," *ACM JETC*, vol. 16, no. 3, pp. 1–20, 2020.

[51] L. Presser and J. R. White, "Linkers and loaders," *ACM Computing Surveys (CSUR)*, vol. 4, no. 3, pp. 149–167, 1972.

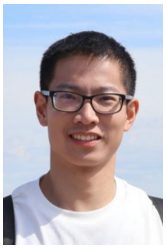
[52] Y. R. Yang and S. S. Lam, "General aimd congestion control," in *Proceedings 2000 International Conference on Network Protocols*, 2000.


**Yida Zhang** Yida Zhang is a Ph.D. candidate in the Computer Science Department at Purdue University. He specializes in compiler-hardware co-design. His research lies on reliability and performance of GPU, CPU, and memory systems. Contact him at zhan3339@purdue.edu.

**Shao-Yu Huang** Shao-Yu Huang is a Ph.D. student in the Department of Computer Science at Purdue University. He specializes in software-hardware co-design at the intersection of compilers and microarchitecture, focusing on improving CPU reliability and mitigating soft errors. Contact him at huan1464@purdue.edu.

**Byounguk Min** Byounguk Min is a Ph.D. candidate in the Computer Science Department at Purdue University. His research interests are in computer architecture and simulator design, with an emphasis on GPU, heterogeneous systems, and intermittent systems. Contact him at min87@purdue.edu.

**Andrew Inho Park** Andrew Park is a PhD student in Elmore Family School of Electrical and Computer Engineering at Purdue University. His research lies on computer architecture and compiler, focusing on developing machine learning accelerator. Contact him: park2041@purdue.edu.



**Gan Fang** Gan Fang is a Ph.D. candidate in the Computer Science Department at Purdue University. His research lies at the intersection of computer architecture and systems, with an emphasis on CPU and GPU microarchitecture, cache and memory optimization, heterogeneous multicore systems, and intermittent computing. Contact him at fang301@purdue.edu.

**Jongouk Choi** Jongouk Choi is Assistant Professor in the Department of Computer Science at the University of Central Florida. He develops architecture/compiler co-design solutions to improve performance, reduce hardware complexity, and address reliability/security problems. Contact him: jongouk.choi@ucf.edu.

**Changhee Jung** Changhee Jung is Samuel D. Conte Associate Professor in the Computer Science Department at Purdue University. His research interests are in compilers and computer architecture with an emphasis on performance, reliability, and security. Contact him at chjung@purdue.edu.