

Compiler-Directed High-Performance Intermittent Computation with Power Failure Immunity

Jongouk Choi
Purdue University
choi658@purdue.edu

Larry Kittinger
Block.one
larry.kittinger@block.one

Qingrui Liu
Annapurna Labs
qingrui@amazon.com

Changhee Jung
Purdue University
chjung@purdue.edu

Abstract—This paper introduces power failure immunity (PFI), an essential program execution property for energy harvesting systems to achieve efficient intermittent computation. PFI ensures program code regions never fail more than once i.e., at most single in-region outage, during intermittent computation as if they are immunized after the first power outage. To enforce PFI automatically for such batteryless systems that use a tiny energy buffer instead, we present its compiler-directed enforcement. The compiler leverages a precise static analysis to partition the program into recoverable regions with the energy buffer size in mind so that their execution can be completed—using the full energy buffered in a single charge cycle—regardless of program execution paths. In this way, no matter how unstable the energy harvesting source is, no region fails more than once.

In the virtue of PFI, this paper presents ROCKCLIMB, a high-performance and rollback-free intermittent computation scheme. It guarantees that PFI-enforced regions never fail, i.e., there is no in-region outage at all. To achieve this, ROCKCLIMB checks if the fully buffered energy is secured at each region boundary. If it is not secured, ROCKCLIMB waits until the energy buffer is fully charged, before executing the following region. In particular, the rollback-free nature of ROCKCLIMB obviates the need to log memory writes—required for rollback recovery—since no region is power-interrupted. As a result, PFI+ROCKCLIMB achieves rollback-free and memory-log-free intermittent computation, ensuring forward execution progress and maximizing it even in the presence of frequent power outages. Our real board experiments demonstrate that PFI+ROCKCLIMB outperforms the state-of-the-art work by 5%—550% on average in various energy harvesting conditions.

I. INTRODUCTION

The ARM roadmap predicts that the number of devices connected to the Internet of Things (IoT) will reach ≈ 50 billion in the next few years [1], and wearables devices are expected to generate the most mass consumer adoption [2]. Powering these IoT devices is a pressing challenge; it is not feasible to change the batteries of billions of the devices. Apart from that, a battery is heavy and bulky for small IoT devices such as wearables. This has brought a tremendous interest in energy harvesting to self-power the devices using ambient energy sources. Owing to the self-sustaining, maintenance-free, and environmentally-friendly nature, energy harvesting is the logical next step in the evolution of IoT [3], [4].

However, energy harvesting systems are prone to power failure due to the unstable nature of harvested energy and the absence of a battery. Since the systems use a tiny capacitor as an energy buffer, they intermittently compute when it provides sufficient energy, which would otherwise die, thus being called

intermittent computation. This implies that frequent power interruptions become the norm of program execution, forcing it to restart from the beginning. Hence, an intermittently-powered microcontroller (MCU) uses nonvolatile memory (NVM) as main memory without caches—due to their power demand—and has some form of recovery support to backup and restore necessary data across a power outage.¹

Existing software-based recovery schemes partition program into a series of recoverable regions (tasks) with checkpointing/logging their input register/memory data in NVM. If any region is interrupted due to power failure, the recovery schemes, in the wake of the failure, first restore the checkpointed/logged data by loading them from NVM and then resume the program at the beginning of the interrupted region [10]–[13]; this is so-called rollback recovery.

Unfortunately, the existing recovery schemes are not systematic, forming their regions sometimes too conservatively or aggressively. If regions are too short (i.e., unnecessarily making frequent checkpoints at each region boundary), the schemes consume more energy for checkpointing but use less energy for computing; that is because checkpoints are the most energy-consuming in that they are essentially NVM stores. While one could take an aggressive approach by forming long regions for fewer checkpoints, expensive re-execution penalty has to be paid by restarting such a long region possibly many times across power outages. Either way, the forward execution progress is limited leading to significant performance degradation. Even worse, the existing recovery schemes could suffer from a *stagnation* problem [14], [15]—livelock-like situation where power failure repeatedly occurs before some long region finishes—making no forward progress in spite of continuous energy consumption (also called a non-terminating bug [16]).

To overcome these challenges, this paper introduces power failure immunity (PFI), a novel program execution property for achieving energy-efficient intermittent computation. PFI ensures that each code region can fail at most once, i.e., a single in-region outage, regardless of power failure frequency. If a region ever encounters power failure, it never fails again during the re-execution—as if it was immunized after the first failure. The *never-fail-again* nature makes it possible

¹We target traditional single-core energy harvesting systems and leave multi-core [5] and accelerator systems [6]–[9] for our future work.

for intermittent computation schemes to take the aggressive region formation (i.e., long regions) without the expensive re-execution penalty.

Thus, enforcing PFI technically solves the stagnation problem unlike dynamic testing (measurement-based) approaches, e.g., auto-tuning [12] and manual energy-debugging [16]–[18] that try to avoid stagnation in a best-effort manner; they both require a huge amount of testing time (up to more than a hundred hours to cover various program inputs and paths). In contrast to the dynamic testing approaches, this paper takes a static analysis approach and presents compiler-directed PFI enforcement that can automatically form stagnation-free regions within a few seconds.

The key insight is energy harvesting systems do not boot until their energy buffer (capacitor) is fully charged as with commodity systems like WISP [19]. In the wake of power outages, it is thus assured that program can make as much progress as the full energy buffer allows, even if no additional energy is harvested. This insight serves as a basis for enforcing the PFI at compile time without paying the high cost of dynamic testing approaches and their false negatives; unlike static analysis, testing might miss stagnations due to the inherent *unsoundness* [20].

To enforce PFI statically, our compiler partitions program into a series of code regions considering the energy buffer size, so that each region can be completed using the energy buffered in a single charge cycle. Given the fully-buffered energy, the compiler analyzes how long an energy harvesting MCU can sustain its execution under the maximum power consumption mode of the MCU which drains the energy from the capacitor at the highest rate; we refer to such a minimum sustainable time bound as *safe active time* (SAT). More precisely, our compiler carefully partitions program into a series of SAT-aware regions such that their worst-case execution time in any given path is never greater than SAT.

In particular, this paper leverages PFI to achieve *rollback-free* intermittent computation without expensive hardware support, e.g., just-in-time (JIT) checkpointing of nonvolatile processors which preserves their volatile states when power is about to be cut off [21]–[25]. To achieve the rollback freedom, we propose ROCKCLIMB guarantees that PFI-enforced regions never fail, i.e., there is no in-region outage at all. In particular, ROCKCLIMB checks if a fully buffered energy is secured at each region boundary to ensure the completion of the next region without power failure. If it is not secured, ROCKCLIMB waits at the boundary until the energy buffer is fully charged before executing the following region. The upshot is that the rollback-free nature of ROCKCLIMB obviates the need to perform logging for each memory write—required for prior work [10], [12], [13], [16], [26], [27] to achieve rollback recovery of power failure.

However, although no region is power-interrupted, a power outage can still occur while ROCKCLIMB waits for the energy buffer to be fully charged at a region boundary; volatile states, i.e., registers, can still be lost upon an outage. To address the issue, our compiler leverages a novel optimization called

distributed checkpointing that saves only essential registers without compromising the recovery guarantee. Unlike prior rollback recovery schemes [10]–[13] that insert checkpoints (i.e., store instructions saving registers to NVM) at the beginning of each region/task boundary, *distributed checkpointing* spreads them out where each register is defined, thereby eliminating unnecessary checkpoints and their energy consumption.

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

- We define PFI as a basic program execution property for achieving energy-efficient intermittent computation and implement its compiler-directed enforcement.
- Our compiler automatically enforces PFI forming stagnation-free regions in 1.4 seconds on average, unlike dynamic approaches that take many hours of testing but can only remove the stagnation found on tested paths.
- We propose ROCKCLIMB that can ensure PFI-enforced regions never fail, i.e., there is no in-region outage at all.
- We propose a new compiler optimization called distributed checkpointing that can remove unnecessary checkpoints thus extending forward progress with their saved energy.
- To the best of our knowledge, PFI+ROCKCLIMB is the most performant software-based intermittent computation scheme that outperforms the state-of-the-art work (up to a 5.5x speedup on average).

II. BACKGROUND AND MOTIVATION

A. Energy Harvesting System Architecture

Due to frequent power outages, researchers equip energy harvesting systems with byte-addressable nonvolatile memory (NVM) as main memory for efficient checkpoint/recovery across the outages. For example, TI’s MSP430FR series of microcontrollers (MCU) have integrated FRAM [29]. This paper targets such MCUs with NVM and a simple in-order processor, that has no cache, as in prior works [11], [14], [15], [29]–[37]. Thus, only volatile data in the processor—i.e., registers—will be lost on a power outage and therefore should be checkpointed for recovery.

B. Crash Consistent Power Failure Recovery

Due to unreliable ambient energy sources, energy harvesting systems suffer frequent power failure that must be recovered in a crash consistent manner [8], [11], [38]–[58]. To achieve the crash consistency, (software-based) prior works partition program into a series of recoverable regions/tasks (and back up necessary data therein to NVM) so that their re-executions result in the same and correct output across power failure; hereafter, we use the term region(s) as the same meaning as task(s). To a large extent, there are two crash

	Partitioning Time	Analysis	Memory Log	HW Support	User Intervention	Checkpoint Type	Re-execution
Auto-tuning [12]	Very long	Dynamic	Yes	No	No	Centralized	Yes
Chinchilla [13]	Very long	Dynamic	Yes	Yes [16], [18], [28]	Yes (energy debugging)	Centralized	Yes
PFI+ROCKCLIMB	Short	Static	No	No	No	Distributed	No

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

consistency approaches that both require handling memory antidependence [59] also known as Write-After-Read (WAR) dependence—since it overwrites an input to be read for re-execution.

First, automatic idempotent region formation such as Ratchet [11], places a region boundary to cut the antidependence(s) and checkpoints live [59] registers. By checkpointing registers, this paper means storing them in NVM, i.e., checkpoints are essentially such store instructions. Second, users can alternatively partition program into a series of regions on their own, preserve the memory locations being overwritten by antidependent stores with logging the original value to NVM, and checkpoint registers [10], [12], [13]. In the wake of power failure, the prior works restart from the beginning of the interrupted region after restoring the checkpointed registers and the logs from NVM.

C. Expensive Centralized Checkpointing

No matter how antidependence is addressed, the prior works [10]–[13] checkpoint every volatile input at the beginning of each region, thus being called *centralized checkpointing*. More precisely, they checkpoint all live-in [59] registers—that hold live [59] values at the beginning of a region—for every region at its entry in case it is interrupted due to power failure; in the wake of the failure, the interrupted region must be restarted from the entry for recovery. Unfortunately, many of live-in register checkpoints tend to be unnecessary wasting harvested energy that could otherwise be used to make further execution progress; it is important to note that because of the NVM write latency/energy, checkpoints are the most expensive instruction in energy harvesting MCUs.

We observe that the live-in registers are often not used in the current region, but read in the later regions; the more regions the live-range [59] of registers spans, the more redundant checkpoint stores the centralized checkpointing generates. As an extreme example, if registers are defined at the beginning of program and read at the end of it, they must be checkpointed at every single region entry. The takeaway is that although the registers are not used in the current region, the centralized checkpointing has no choice but to save them, otherwise their values are lost upon power failure; *this is why all the prior works end up checkpointing all live-in registers every time a region starts*. With that in mind, we propose a new compiler optimization called *distributed checkpointing*. It spreads out checkpoint stores to where volatile registers become live-

out [59], i.e., at their last-update point in each region.² In this way, they do not have to be checkpointed repeatedly in the following regions unless they are updated and live-out again. Section V-B details the distributed checkpointing.

D. Stagnation in Energy Harvesting Systems

Suppose a region whose execution time is greater than the power failure period, i.e., the time between the failures. If they periodically occur with the same frequency, the program ends up rolling back to the beginning of the region indefinitely. That is because the failures keep occurring before the end of the region is reached, in which case the program just wastes harvested energy in vain making no forward execution progress. Researchers call this livelock-like phenomenon *stagnation* [12]–[16], [18], [28].

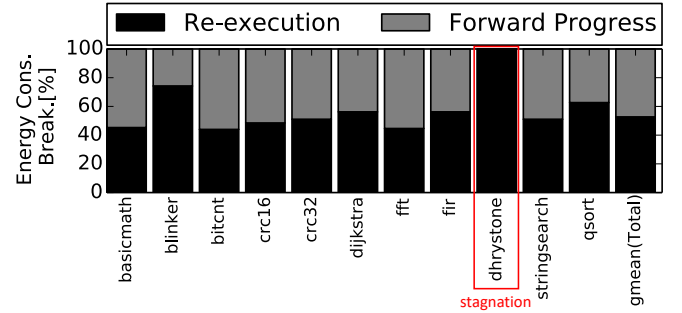


Fig. 1: Energy breakdown of Ratchet for a real energy harvesting condition. For *dhrystone*, 100% re-execution means stagnation. A geometric mean (gmean) is calculated only for those non-stagnated.

It turns out that prior works can suffer the stagnation problem [7], [10], [11], [34], [62]–[64]. To investigate the phenomenon, we conducted experiments by using one of the prior works, the idempotence-based power failure recovery scheme called Ratchet [11]. We ran 11 benchmark applications on a real energy harvesting board (the evaluation setting is described in Section VI-A) and analyzed the cost of re-execution across power failure by breaking down the total energy consumption of each application into two parts: re-execution and forward progress as shown in Figure 1. Note that

²Distributed checkpointing is akin to incremental checkpointing that can be achieved with either hardware [60] or runtime support [61], because they checkpoint only updated registers. However, all updated registers do not require checkpointing, e.g., some registers could be re-defined before their use, in which case the incremental checkpointing wastes harvested energy by persisting such dead registers unnecessarily. In contrast, our distributed checkpointing preserves only essential (live-out) registers by taking into account their liveness at compile time.

we disabled Ratchet’s timer based checkpointing, since it could result in wrong recovery for those regions that have Write-After-Read-After-Write (WARAW) dependence [59], [65].

We discover that Ratchet can be trapped in some long regions leading to stagnation, thus never finishing the program as in the case of *dhrystone* in Figure 1; region size analysis is deferred to Section VI-A. Even for non-stagnating applications, Ratchet ends up wasting 47-75% of hard-won energy by repeatedly checkpointing/restarting the same interrupted region across power failure—before getting out of the region. Overall, Ratchet spends 52% of the total energy consumption for re-executions, leading to significant performance degradation. On the contrary, our proposal can always make forward progress across power failure without re-execution and stagnation at all, thereby achieving energy-efficient and rollback-free intermittent computation (Section III-B).

E. Prior works are Not a Solution for Stagnation

To address the stagnation problem, the state-of-the-art works use dynamic testing approaches by using an energy debugger [13], [16], [18], [28] or an auto-tuning framework [12] as summarized in Table I. However, such dynamic testing approaches have several limitations.

The energy debugger based approach requires multi-step *expert-level* user interventions for precise diagnosis [13], [16], [18], [28]. First, users should manually measure basic blocks’ energy consumption with randomized inputs. Second, users need to compare the energy consumption of a given basic block to the total energy availability obtained by estimating the storage capacity of the energy buffer. Third, if the basic block consumes more energy than the available amount thus being vulnerable to stagnation, users should rewrite the code or let the prior work [16] split the block to smaller pieces with inserting checkpoint stores and memory logs therein.

The crux of the problem with the debugger based schemes is that the energy profiling [16] takes about 30 minutes on average even for toy applications, which makes the schemes impractical. Furthermore, the resulting program can still suffer stagnation provided some of untested program paths is taken; this is technically possible since users cannot cover all possible paths with dynamic testing due to its inherent *unsoundness* [20]. Due to the issue, the state-of-the-art work Chinchilla [13] ends up inserting its region boundary at each basic block, rendering the regions too small—though it can adaptively skip register checkpointing when energy harvesting condition is good.

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

Benchmark Application [12], [13], [66], [67]	Auto-tuning [12] (in hours)	Energy-debugging [13], [16], [18] (in hours)	Compiler-directed PFI enforcement (in seconds)
basicmath	124.2	9.3	2
blinker	11.2	0.5	1
bitcnt	9.6	9.7	4
crc16	2.3	1.7	1
crc32	18.1	3.7	1
dijkstra	7.2	5	1
fft	8.9	19.3	2
fir	6.0	2	1
dhrystone	1.0	5	1
stringsearch	6.2	22	2
qsort	53.5	4	1
geomean	9.6 hours	4.7 hours	1.4 seconds

TABLE II: Comparison of stagnation-aware region formation schemes in terms of the time taken to complete the region formation

To evaluate the usability of the prior dynamic testing approaches, we measured the total elapsed time for completion of their recoverable region formation; two different test inputs were used for each of 11 applications. For the auto-tuning approach, we tested 200 different region size variants as suggested in the original work [12] with two different power failure traces. As shown in Table II, the auto-tuning (2nd column) and the energy-debugging (3rd column) approaches take a considerable amount of time for each application (up to more than 5 days as in *basicmath*). Note that although both dynamic approaches finally form regions after many hours, such a high cost has to be paid anew for different program/input combinations and various power failure behaviors. Unfortunately, this is a serious problem in that energy harvesting systems inevitably encounter the significant change of the failure behavior—because the underlying harvesting condition often unpredictably varies over time. In contrast, our proposal only requires a few seconds of compilation time as shown in the 4th column in the table. On average, PFI compiler is five orders of magnitude faster than the both dynamic testing approaches. More importantly, unlike the approaches, our static analysis can guarantee stagnation-free execution regardless of program paths and inputs.

III. OUR APPROACH: PFI+ROCKCLIMB

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

The second step is leveraging PFI to achieve rollback-free and high-performance intermittent computation—that we call ROCKCLIMB—where no region is power-interrupted (Section III-B). That is, ROCKCLIMB not only ensures that hard-won energy is never wasted for region re-execution, but also maximizes the forward execution progress fully utilizing the energy only for computation. Furthermore, since all regions

are sure to finish thanks to ROCKCLIMB, it can eliminate expensive per-region memory logging—that is required by prior work for crash consistent rollback recovery—without compromising the correctness guarantee. The rest of this section details the PFI and its compiler-directed enforcement, the workflow of which is shown in Figure 2, and shows how it guarantees that no region ever fails.

A. PFI: Power Failure Immunity

To prevent repetitive in-region outages, this paper leverages two important observations. First, energy harvesting systems do not start to operate their microcontroller (MCU) until the energy buffer (capacitor) is fully charged as with virtually all commodity systems, e.g., WISP [19]. That is, when the MCU is ready to resume the execution in the wake of power outage, the capacitor is always sure to have the fully buffered (charged) energy at the starting point of the resumption. The implication is that the power-interrupted region can make as much progress as the full energy buffer allows, even if there is no additional energy is harvested. We refer to the minimum progress time, for which the MCU can be sustained under the fully buffered energy, as safe active time (SAT).

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

$$\text{PFI enforcement constraint: } WCET(r) < SAT(\mu) \quad (1)$$

With that in mind, for a given region (r) and the underlying MCU (μ), we formulate the problem of ensuring the forward execution progress as Eq.1 above.

Thus, PFI can achieve stagnation freedom by partitioning the original program into *SAT-safe regions*, each of which satisfies the PFI constraint Eq.1; even if the SAT-safe regions may encounter power failure, they **never fail again** upon recovery from the failure. In other words, when power comes back, the previously interrupted region never retreats before reaching the end of the region, which ensures forward progress to the next region without exception.

1) *SAT Calculation under Worst-Case Execution Scenario:* To calculate the SAT for a given MCU's full energy buffer in a *sound* way (where all regions satisfy the PFI constraint Eq.1), we must consider the worst-case scenario of MCU operation, which would otherwise fail to achieve PFI for those regions power-interrupted under the scenario. Hence, our PFI compiler considers the most harsh environmental setting where there is no harvested energy, and the MCU consumes the maximum amount of energy all the time. That is, the compiler calculates the SAT by analyzing how long program can sustain its execution, under the maximum power consumption mode of the MCU—which drains the energy from the capacitor at the highest rate—to take into account the worst-case scenario.

2) *PFI Region Formation:* To form PFI-enforced regions that satisfy the above constraint Eq.1, the compiler takes a 2-step approach. First, it forms initial regions at function call boundaries and loop headers. As shown in Figure 2 (b),

the input program (a) gets to have a region boundary at a `Print()` callsite and the entry of a `for` loop.

Second, after finishing the initial region formation, the compiler performs per-region WCET analysis to check if the initial regions satisfy the PFI constraint Eq.1. If so, the regions remain the same—until they are instrumented later for rollback-free intermittent computation; otherwise, the compiler partitions the SAT-unsafe region, that violates the PFI constraint Eq.1, into a series of SAT-safe regions; the SAT-driven partitioning might need to be repeated if the remainder of the cut is still too long to satisfy the constraint. For example, as shown in Figure 2 (c), the first two initial regions in (b) are both cut at the point where their WCET hits SAT. As a result, every program point belongs to one of SAT-safe regions where PFI is enforced.

B. ROCKCLIMB: Never Fail Whatsoever!

Once SAT-safe regions are formed, our compiler enables ROCKCLIMB that leverages the PFI as a basis for achieving rollback-free intermittent computation, i.e., extending the PFI to much stronger guarantee that **no region ever fails**. In fact, the name ROCKCLIMB is inspired by rock climbing; climbers divide their route into multiple sections, and at the entry of each section they usually rest eating energy bars until they get powered up enough to pass the section. Similarly, to complete each PFI-enforced region with no power failure, ROCKCLIMB checks the energy buffer at each region boundary. If the buffer is not fully charged, ROCKCLIMB waits for the buffer to secure the full energy before starting the next region; otherwise, it is immediately started with the guarantee of failure-free completion—because a PFI-enforced region can always finish with a fully buffered capacitor.³

In particular, ROCKCLIMB's guarantee of no in-region failure simplifies achieving crash consistency obviating the memory logs in each region. As discussed in Section II-B, a root cause of the memory inconsistency is that in the wake of power failure, program control rolls back across antidependence, reading values updated by stores left behind the failure. That is why prior work logs memory inputs of each region to make a copy of the original values before they are overwritten by antidependent store instructions.

On the contrary, since ROCKCLIMB's regions are never power-interrupted (thus no rollback), they do not have to log memory inputs at all; the absence of rollback recovery means no need to handle the restoration of memory inputs. The upshot is that ROCKCLIMB's rollback-freedom saves the high energy/latency of the NVM logging stores, thereby achieving an energy-efficient and high-performance energy harvesting system. Figure 3 highlights ROCKCLIMB compared to prior works that partition program into several regions with memory logs for recovery. While the prior works keep spending their energy for logging, restoring, and re-executing as shown in the figure, ROCKCLIMB here makes a further forward

³This paper assumes that the energy harvesting system does not suffer accidental reliability issues such as energetic particle striking or capacitor malfunctioning in the circuit.

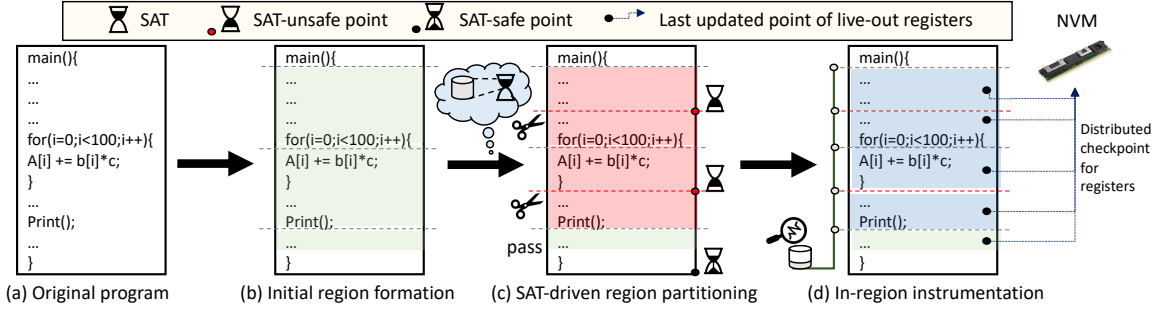


Fig. 2: Workflow of PFI compiler: it partitions program into a series of PFI-enforced regions and instruments them to achieve rollback-free and high-performance intermittent computation.

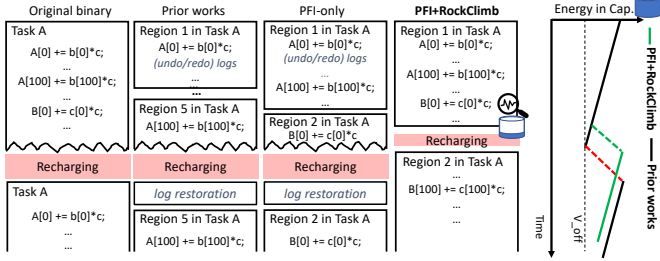


Fig. 3: Comparison of intermittent computation schemes: Each scheme runs the same program (Task A). While prior works form many regions, e.g., Region 1~5 in Task A, PFI generates a few regions, and PFI+ROCKCLIMB further lengthens the region size and eliminates the re-execution.

progress due to its log-free and re-execution-free intermittent computation.

a) *Region Instrumentation*: When a program control reaches the end of a region, ROCKCLIMB checks the energy availability (full capacitance) before starting the next region. For this purpose, as shown in left side of Figure 2 (d), the compiler thus inserts—at each region boundary—the voltage-level checking code offered by commodity energy harvesting systems, e.g., TI-MSP430’s power management library [68], [69] supports the checking through a voltage comparator interrupt; Section V-A offers more details.

Finally, our compiler, if necessary, inserts checkpointing stores to save volatile registers in some regions. Although PFI-enforced regions are never interrupted by power failure, it can still occur at region boundaries while ROCKCLIMB waits the capacitor to be fully charged. In this case, unlike NVM resident data, volatile registers are lost on power failure. To this end, as shown in the right side of Figure 2 (d), the compiler checkpoints registers using a novel compiler optimization called *distributed checkpointing* that saves only essential registers without compromising the recovery guarantee (Section V-B).

IV. IMPLEMENTATION

A. SAT Calculation

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

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

where C_{buf} , V_{max} , V_{min} are capacitance, MCU power-on voltage level, and MCU power-off voltage level, respectively. Then, the compiler measures MCU’s energy consumption during operation as [70]:

$$E_{tot} = P_{tot}t = V_{dd}I_{leak}t + C_{msp}V_{dd}^2, \quad (3)$$

where V_{dd} , I_{leak} , C_{msp} are input voltage to MCU, leakage current, and the MCU capacitance, respectively. However, the input voltage (V_{dd}) is not constant; it decreases as the energy buffer is discharged (Section III-A). With that in mind, we consider the MCU as a resistance-capacitor (RC) circuit, i.e., our PFI compiler substitutes the input voltage with $v_o(t) = V_o e^{-t/CR}$, where the capacitance (C) is the same as C_{buf} , and the resistance (R) can be estimated as $R = V/I$. Here, we can readily get the V and I as the operating voltage and the maximum current, respectively, from the MCU manual. If a certain MCU’s manual does not specify them, our compiler can adapt the typical leakage current and capacitance model [70].

The key insight of PFI is that, to guarantee the forward progress, the available energy input obtained by Eq. 2 should be always greater than the energy consumption of the underlying MCU given by Eq. 3. In light of this, ROCKCLIMB obtains the SAT by calculating a threshold time t in the following formula (Eq. 4).

$$\frac{1}{2}C_{buf} * (V_{max}^2 - V_{min}^2) > V_o e^{-t/CR} (I_{leak}t + C_{msp}(V_o e^{-t/CR})^2) \quad (4)$$

In particular, SAT should also cover the system recovery cost to safely restore the checkpointed registers at the system reboot time in the wake of power failure. We use the simple energy profiling model—which can be further improved by using a recent advanced model [69], [71]. For simplicity, our PFI compiler conservatively updates SAT (Eq.2) as $SAT = SAT - \text{Recovery_Cost}$ by assuming all registers are restored upon recovery as with prior work [11].

B. WCET Analysis

First of all, our WCET calculation is straightforward for two reasons: (1) the energy-harvesting MCU architecture is simple, i.e., a cache-free single in-order core, unlike multi-core systems backed with out-of-order execution and deep cache hierarchy [72], [73], and (2) unlike traditional WCET

calculation [74], [75], ours does not require whole program analysis.

To analyze WCET for each initial region shown in Figure 2(a), our PFI compiler navigates all possible paths in region-based control flow subgraph; while whole-program-analysis based WCET calculation is challenging, our **region-based** (intra-region) analysis makes it possible to run the WCET analysis for all the benchmarks we tested.

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

For instruction-level WCET calculation, we build a cost model by referring to the MCU manual which gives the execution cycles of each instruction [76]⁴; if executing some instructions takes a range of cycles, our compiler takes the worst latency. Since the worst-case execution cycles of instructions are fixed, the timing cost model is simple and safe unlike the profile-based energy consumption model [16] that can be inaccurate in different execution environments.

C. SAT-Driven Region Formation

Once the SAT is obtained, our PFI compiler statically converts it to the MCU cycles and splits those initial regions, that are SAT-unsafe, into SAT-safe regions. As shown in Figure 4(a), the compiler keeps accumulating the execution time (cycles) of instructions on every path in a given initial region, i.e., the accumulated sum is the WCET of the path between the region entry and the current instruction just visited there.

During the instruction time accumulation on each path, if the sum becomes greater than or equal to the SAT (i.e., the current instruction and its successors are susceptible to power failure), then the compiler cuts the susceptible path by placing a new region boundary before the current instruction with zeroing the sum for a further partitioning.⁵ This happens recursively until the last instruction of the path is reached. Figure 4(b) shows the final shape after partitioning the original region with SAT of 200 cycles.

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

⁴For example, a simple register update instruction takes 1 cycle while a load instruction takes 5 cycles at least.

⁵Technically, the region boundary instruction is a checkpoint store for saving a program counter so that it serves as a recovery point in case the following region is power-interrupted.

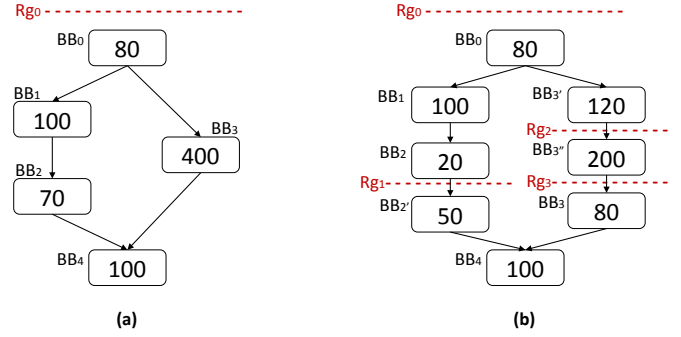


Fig. 4: Region partitioning with the SAT threshold of 200; the number in each basic block (box) represents its total execution cycles, and dashed lines represent region boundaries.

Algorithm 1 Region Formation Algorithm

```

1: for each basic block  $bb_i$  in CFG do
2:    $Cycle_{bb_i} \leftarrow Cycle_{ori_{bb_i}} + CkptCycles_{bb_i}$ 
3:    $IncomeCycle_{bb_i} \leftarrow 0$ 
4: end for
5: for each basic block  $bb_i$  in program topological order do
6:   if  $bb_i$  starts with region boundary then
7:      $accum\_cycle \leftarrow Cycle_{bb_i}$ 
8:   else
9:      $accum\_cycle \leftarrow Cycle_{bb_i} + IncomeCycle_{bb_i}$ 
10:  end if
11:  while  $accum\_cycle > threshold_{time}$  do
12:    place boundary and split  $bb_i$  into  $bb_i'$  and  $bb_i$ 
13:    recalculate  $Cycle_{ori_{bb_i}}$  and  $CkptCycles_{bb_i}$ 
14:     $accum\_cycle \leftarrow Cycle_{ori_{bb_i}} + CkptCycles_{bb_i}$ 
15:  end while
16: end for

```

a) *Loops*: To handle loops, our PFI compiler inserts a region boundary in the loop header. The compiler splits the loop body if its WCET is greater than SAT. Otherwise, the compiler tries to extend such a short loop body since it can cause more live-out registers and checkpoint stores across the region boundary in the loop header. That is, to address this issue, our compiler repeatedly unrolls such a loop as long as the WCET of the loop body is smaller than SAT. By maximizing the loop body in this way, the compiler can minimize the number of live-out registers in the loop, thereby reducing the number of checkpoint stores to be inserted. Currently, for those loops whose iteration count is statically unknown, the compiler just inserts a region boundary in the loop header without performing the unrolling.

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

Although individual IO operations are failure-atomic, their combination can violate the PFI constraint (Eq.1), provided they are performed in parallel. To address the issue, we

should conservatively estimate the SAT by covering the total current consumption of the IO operations. For this purpose, our compiler updates the resistance (R) of Equation 4 with the sum of each IO's maximum current, i.e., $R = \frac{V}{I_{MCU} + I_{IO_1} + I_{IO_2} + \dots + I_{IO_n}}$; this paper assumes that the required values can be found from the IO manuals.

D. Discussion

As discussed in Section IV-B, to meet the PFI enforcement constraint (Eq.1) for region formation, our compiler simply adds up the worst-case instruction latencies and places a region boundary at the point where the accumulated sum becomes greater than the SAT—without considering the instruction pipeline. Thus, such a conservative WCET analysis tends to generate smaller regions than traditional WCET analysis. The rationale behind is that the smaller region leaves residual energy in the capacitor when the region ends. In particular, the residual energy can serve as a safe margin⁶ to guarantee the soundness of PFI-enforcement even when the full capacitance is not secured due to reliability issues. For example, even if a capacitor malfunctions holding a smaller amount of energy than normal due to its physical properties, e.g., abnormal leakage or aging-induced damage, PFI-enforced regions do not exceed the SAT.

V. OPTIMIZATION

A. Securing Full Capacitance for Rollback-Free Computation

At each region boundary of SAT-safe regions that satisfy the PFI constraint Eq.1, our compiler enables ROCKCLIMB that dynamically checks if the full capacitance is secured, which would otherwise wait for the energy buffer to be fully charged, before starting the next region. For this purpose, ROCKCLIMB leverages a voltage emergency interrupt of commodity energy harvesting systems, e.g., TI-MSP430 [68], which can compare a current voltage level to a certain voltage threshold that can be controllable by software. That is, at each region boundary, ROCKCLIMB enables the voltage interrupt by controlling the interrupt vector and immediately starts the next region unless the interrupt is generated; otherwise, ROCKCLIMB puts the microcontroller (MCU) to a power-down mode so that it can be rebooted when the buffer is fully charged [69]. The implication is two-fold. First, no-interruption here means that the MCU is directly powered by the energy harvesting source with the energy buffer fully charged, and therefore the next region is guaranteed to finish without power failure. Second, the voltage interrupt should be disabled before executing any instruction of the next region. That is, at the beginning of a region, the compiler inserts the code that disables the interrupt.

B. Compiler Optimization: Distributed Checkpointing

Unlike centralized checkpointing, our compiler does not checkpoint all live-in registers at each region boundary. Instead, it spreads the checkpoints (store instructions saving

⁶The safe margin does not harm the performance because the residual energy is going to be picked up by ROCKCLIMB, i.e., the charging time before executing the next region becomes shorter.

registers in NVM) out where each register is defined; they are often distributed to many regions, thus being called *distributed checkpointing*. More precisely, the compiler checkpoints the updated register of each region, which is used in following regions and therefore called *live-out* [59], right after the register update. In data flow terms, we define the output of region r as the live-out registers defined in the region—the values that are written and downward-exposed [59]: $ckpt_r = Def_r \cap LiveOut_r$, where Def_r is the set of registers defined in r and $LiveOut_r$ is the set of live-out registers of r . For each region r , the compiler checkpoints only the registers belonging to $ckpt_r$ as soon as they are defined in the region.

The distributed checkpointing has several implications. First, if the register is defined multiple times in a region, the compiler only checkpoints the last-updated value used by later regions, i.e., the live-out value.

Second, since the compiler checkpoints such a live-out register right after its update, each checkpoint shares the same register and the value with the preceding instruction that updates the register. Here, the benefit is two-fold: (1) minimal dynamic switching activity in the circuit; in contrast, two consecutive instructions with different operands/values increase the power consumption by 20% due to the switching activity [77], and (2) rare chance to increase the peak power consumption—typically made by consecutive checkpoints (NVM stores)—since no two checkpoints are consecutively placed in our distributed checkpointing; even if registers are updated in a row, their checkpoints are interleaved with the instructions that update the registers, i.e., checkpoints are always separated.

Third, at the beginning of each region, all input registers to the region are already sure to have been checkpointed; thus, no action is required when each region entry is reached—unlike traditional JIT checkpointing that must save all registers before impending power failure [22], [51], [67], [78].

Fourth, once a register is checkpointed, no further checkpoint is necessary across regions unless the register is redefined and becomes live-out again. That is, unlike centralized checkpointing, the live-in registers of each region which are not written in the region, do not have to be checkpointed; again they are known to have already been checkpointed somewhere before the region entry. Finally, the compiler can minimize the number of necessary checkpoints.

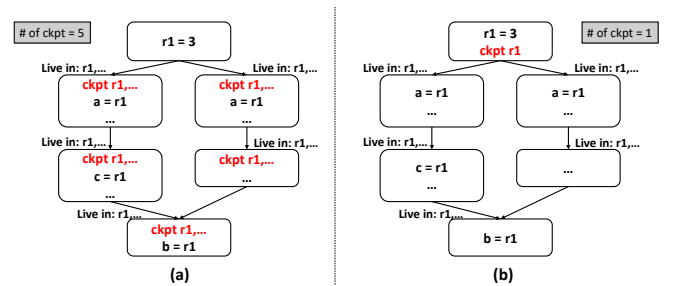


Fig. 5: Checkpoint reduction by distributed checkpointing: (a) centralized checkpointing and (b) distributed checkpointing. Each box represents a program region.

Figure 5 shows how the compiler removes a majority of

the checkpoints introduced by the centralized checkpointing of prior works [10]–[13]. Here, $r1$ is defined on the top region (box) and used in other regions. In particular, $r1$ is live at the entry of all regions except for the top one. In Figure 5(a), the centralized checkpointing inserts five checkpoints, each of which saves the live register $r1$ (and other live-ins if exist) at the entry of the bottom five regions. Note that checkpointing $r1$ at the entry of the 2nd region on the left branch is redundant since the predecessor region has already checkpointed $r1$. Even worse, in the 2nd region on the right branch, the centralized checkpointing stores $r1$ though it is not even used there. In contrast, our distributed checkpointing stores $r1$ only once right after it is defined on the top region as shown in Figure 5(b). Here, our SAT-driven region formation is aware of the additional code for both voltage interrupt controlling and distributed checking already. Thus, the compiler strictly maintains the PFI-enforcement for all regions.

VI. EVALUATION

A. Experimental Setting

We implemented novel compiler techniques described in Section IV in the LLVM compiler infrastructure [79] and conducted experiments by running compute-intensive 11 benchmarks that are used in prior works [66], [67], [80], [81]; sensing applications are ruled out on purpose, because they mostly consist of I/O or sensor tasks that must be power-failure-atomic. This implies that the tasks must be formed that way by the system designer in the first place; once they are formed, it is technically impossible to partition them [12].

To evaluate the effectiveness of PFI+ROCKCLIMB in preventing stagnation, we conducted experiments using TI’s MSP430FR5994 [29] evaluation board with Powercast P2110-EVB RF energy harvester [82] as our energy harvesting system testbed, following the same convention used by prior works [10], [13], [17], [51], [83]–[85]; we equipped the board with a $10\mu\text{F}$ capacitor which is used as energy storage of commodity systems such as WISP [19]. To power the energy harvesting system, we used Powercast TX91501-3W transmitter emitting RF signal at 915 MHz center frequency to the system supplying 6.1 dBi patch antenna. We placed the RF transmitter as real energy source 50cm away from the energy harvesting system by default; we also varied the harvesting condition for sensitivity analysis (Section VI-C).

B. Stagnation Analysis

To analyze the stagnation problem, we conducted experiments by running the benchmark applications with 4 different schemes as shown in Figure 6: (1) Ratchet [11], (2) Chinchilla [13] the state-of-the-art software solution for stagnation freedom, (3) PFI-only scheme that partitions program to SAT-safe regions but leave memory logs and checkpoints therein for recovery, and (4) PFI+ROCKCLIMB as shown in the figure. Here, we assumed that the stagnation occurred if program had not finished within an hour; all benchmarks should have been finished within a couple of minutes. We avoid testing

auto-tuning [12] since it requires too much tuning cost in real harvesting situation (Section II-E).

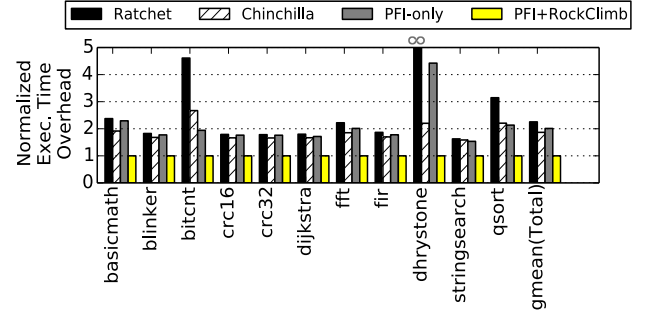


Fig. 6: Performance results in real energy harvesting situation. We compare PFI+ROCKCLIMB with Ratchet and Chinchilla. Y-axis shows the normalized execution time to PFI+ROCKCLIMB. ∞ represents the stagnation problem.

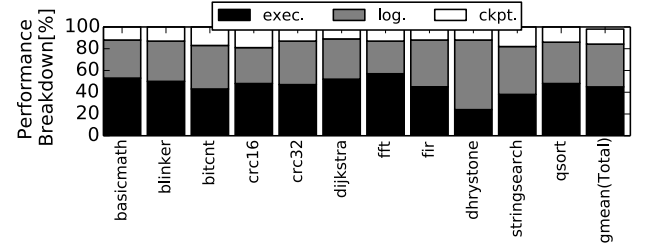


Fig. 7: Performance Breakdown of PFI-only.

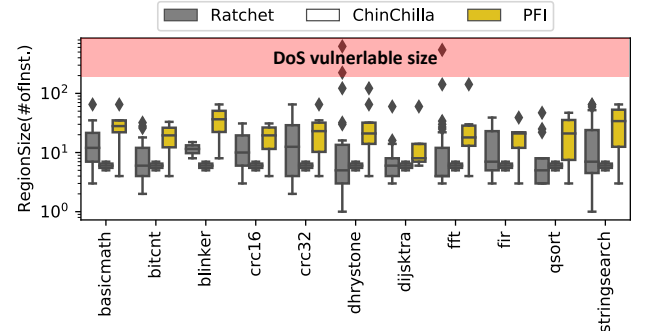


Fig. 8: Region Size Comparison of Ratchet/Chinchilla/PFI.

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

On the other hand, Chinchilla, PFI-only, and PFI+ROCKCLIMB completed all applications. However, PFI-only and Chinchilla are 2x and 1.85x slower than PFI+ROCKCLIMB on average, respectively; Ratchet is 2.2x slower than PFI+ROCKCLIMB on average. This is mainly because they cause the overheads of re-execution, logging, and checkpointing—though Chinchilla was faster than PFI-only thanks to its adaptive execution that can

skip checkpointing sometimes by considering energy source condition. Overall, PFI+ROCKCLIMB outperforms Ratchet, PFI-only, and Chinchilla thanks to the re-execution-free and memory-log-free nature.

In particular, when ROCKCLIMB is enabled for PFI-only, it becomes 2x faster. That is because ROCKCLIMB eliminates the logging (and their restoration) overhead. To see the benefit of ROCKCLIMB in more detail, we analyzed the performance breakdown of PFI-only into three parts: execution, checkpointing, and logging. As shown in Figure 7, the logging overhead is more than 40% on average, i.e., ROCKCLIMB can avoid the expensive cost and make a further forward progress.

1) *Region Size Characteristics*: To figure out the reason for the stagnation and characterize the regions of different schemes, we measured the region size of each application for Ratchet, Chinchilla, and PFI. Here, we counted the number of instructions executed during the execution of each region. As shown in Figure 8, there are two outliers, i.e., excessively long regions, in *dhystone* for Ratchet, leading to stagnation. For other applications, Ratchet forms shorter regions than PFI on average. This is because Ratchet requires each region to be idempotent by cutting all antidependent store-load pairs, which makes the region size small [65], [86].

Unlike Ratchet, two schemes Chinchilla and PFI do not generate stagnating regions. In particular, PFI forms relatively longer regions than others on average; this trend demonstrates that PFI can aggressively increase the region size as long as it does not violate the constraint Eq.1. On the other hand, Chinchilla generates very short regions because it considers each basic block as a region, the entry of which—if not skipped by the adaptive execution—checkpoints all registers, to address the stagnation problem. Although the region size is short, Chinchilla outperforms Ratchet. That is because Chinchilla's adaptive execution can skip the register checkpointing according to the underlying power outage behavior.

2) *Performance Modeling and Analysis*: To analyze the performance benefit of ROCKCLIMB, we set the cost model of PFI+ROCKCLIMB and PFI-only schemes as following: $PFI_only = orig.exec + checkpoint + logging + reexecution + \sum_0^m T_{recharging}$, whereas $PFI + RockClimb = orig.exec + checkpoint + \sum_0^n T_{wait}$. That is, PFI-only execution time (PFI_only) consists of original execution time ($orig.exec$), checkpoint time ($checkpoint$), logging time ($logging$), reexecution time ($reexecution$), and the sum of recharging time across the m number of power outages ($\sum_0^m T_{recharging}$). On the other hand, PFI+ROCKCLIMB execution time ($PFI + RockClimb$) is comprised of original execution time, checkpoint time, and the sum of waiting time across the n number of waits at region boundaries ($\sum_0^n T_{wait}$).

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

Moreover, even if the number of waits n could be greater than the recharging count m (i.e., the number of power outages), the total waiting time can be easily paid off by avoiding logging and reexecution time overheads. This is confirmed by our experiments; it turns out that PFI+ROCKCLIMB waited 10~15 times while PFI-only had 3~5 power outages on average, but PFI+ROCKCLIMB is almost 1.7x faster than PFI-only as shown in Figure 6. The results with various energy harvesting settings show the same trend (Section VI-C).

3) *Distributed Checkpointing*: To analyze the impact of distributed checkpointing, we measured the number of checkpoint stores of PFI at compile time and run time for both centralized and distributed checkpointing schemes. Figure 9 shows that when the distributed checkpointing is enabled, it can reduce the number of checkpoint stores of the variant of PFI—which uses centralized checkpointing on purpose—by 42% and 21% on average at compile time and run time, respectively.

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

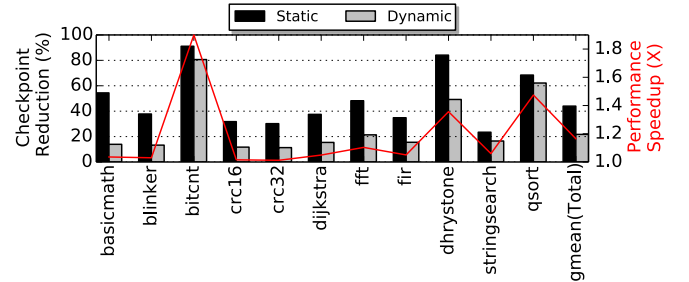


Fig. 9: Checkpoint reduction by distributed checkpointing.

C. Sensitivity Analysis

1) *Experimental Setting*: Rather than placing the RF transmitter in the same position (50cm away from the system), which is conducted by prior works [10], [13], [17], [51], [83]–[85] but considered to be unrealistic, we performed additional experiments to analyze the performance of PFI+ROCKCLIMB compared to PFI-only and Chinchilla with the same 11 benchmarks in various energy harvesting situations including an outage-free case and many other unpredictable power failure cases as shown in Table 10. Each power trace in the table causes a different power outage pattern. We found that the trace 10 caused only one power outage while the trace 12 incurred 12 power outages in one second as shown in Table III. With these different power failure patterns, this paper will discuss how much performance improvement PFI+ROCKCLIMB can achieve by comparing it to prior works for each pattern.

For realistic experiments, we developed a power generator board with MSP430FR5969 to generate various power inputs

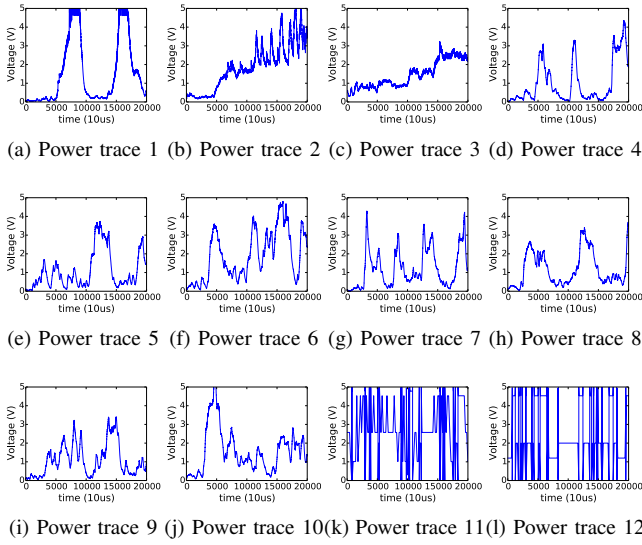


Fig. 10: Energy harvesting trace; the plots in this table show voltage input fluctuations to MCU during 12 different movements from an RF energy harvesting reader [66], [87]

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

TABLE III: The number of power failures per second in traces.

with the power traces collected by prior works [66], [87] in real energy harvesting settings; the power generator provides supply voltage to our target energy harvesting system board through GPIO pins according to the traces.

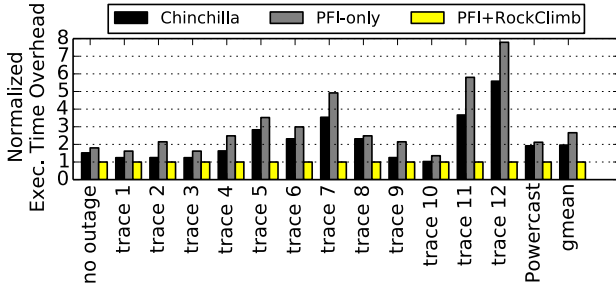


Fig. 11: Performance results in various situations; Y-axis shows the normalized execution time to the baseline.

2) *Overall Performance Trend:* Figure 11 shows the performance results on average in all different situations, e.g., no power failure, and various power patterns from trace 1 to 12 shown in Table 10, including the Powercast RF transmitter (discussed in Section VI-A). The Y-axis is the normalized execution time to PFI+ROCKCLIMB as a baseline. In summary, PFI+ROCKCLIMB is always faster than others, achieving 1.9x and 2.7x average speedups over Chinchilla and PFI-only, respectively. PFI-only shows the worst performance across all traces. Due to the logging and reexecution overheads, it ends up with lower performance compared to others. Overall, Chinchilla shows relatively better performance overhead than

PFI-only, since Chinchilla can skip checkpointing at some points depending on energy source condition.

a) *No Power Failure:* When there is no power failure, Chinchilla and PFI-only cause about 1.5x and 1.9x slowdowns, respectively, compared to PFI+ROCKCLIMB as shown in Figure 11. Here, Chinchilla recognizes that energy source condition is good, and thus it skips most of checkpoint stores. Nevertheless, since it cannot avoid memory logging overhead, it should check log entries at every memory update and flush the logged data at some points, which causes a significant performance overhead. This is why Chinchilla underperforms PFI+ROCKCLIMB even in the power-failure-free case.

b) *Various Power Failure Patterns:* When there is frequent power failure, both Chinchilla and PFI-only cause memory logging and power-interrupted region reexecution overheads unlike PFI+ROCKCLIMB. In particular, with trace 12, Chinchilla and PFI-only show 5.7x and 7.7x slowdowns, respectively, compared to PFI+ROCKCLIMB. The reason is that the trace causes the most frequent power outages among all traces, i.e., generating 15x more outages than trace 10. This implies that when there are frequent power outages, the both prior schemes likely cause a much higher performance overhead. Here, Chinchilla cannot skip register checkpointing due to the frequent power outages—since it recognizes that the harvesting condition is poor. On the other hand, for trace 10, Chinchilla is comparable to PFI+ROCKCLIMB. That is because the trace causes the smallest number of power outages and happens to cause power failure at a right time, i.e., right after register checkpointing, minimizing the waste of Chinchilla’s rollback recovery. Thus, PFI+ROCKCLIMB is only 5% faster than Chinchilla for trace 10.

VII. OTHER RELATED WORKS

A. Single In-Region Outage

While our compiler automatically ensures the single in-region outage with PFI (Section III-A), a couple of HW/SW co-design works manually achieve the similar concept based on HW [36] or compiler support [88]. First, a recent work called HomeRun [36] integrates user-defined regions with HW support for capacitor control. At a high level, if power failure occurs during the execution of a certain region, the HW waits until a sufficient amount of energy is charged for the region to be completed in the wake of the power failure. In this way, the interrupted region never encounters power failure again as with PFI. However, HomeRun forces the users to form the failure-atomic regions unlike our automated solution.

Second, another HW/SW co-design work leverages compiler-assisted region formation with manual size control on top of HW-based register checkpointing [88]. For crash consistency, the compiler first forms idempotent regions (refer to Section II-B) and splits those regions whose size is greater than a user-provided threshold for failure atomicity. Finally, the compiler inserts a trigger point [88] at each region boundary on which the HW checkpoints registers for recovery. However, the prior work still requires user-intervention, HW support, and additional power consumption. In contrast, PFI+ROCKCLIMB

is a software-only approach yet achieves high-performance intermittent computation.

B. Stagnation

To address the stagnation problem, prior works leverage static analysis [89], just-in-time (JIT) checkpointing [17], [51], [67] or copy-on-write mechanism with timer based checkpointing [15].

First, a prior work [89] relies on the worst-case energy (WCE) analysis to address the stagnation. The WCE-based approach estimates the energy consumption of each basic block of given program to partition those basic blocks that are vulnerable to the stagnation as with Chinchilla [13] but through static analysis rather than dynamic testing [13]. However, the WCE-based approach [89] is not sound because its energy cost model has no consideration on environment change, e.g., temperature. According to MSP430 manual [68], when temperature increases, the MCU consumes up to 6x more energy than it operates under room temperature. The implication is three-fold: (1) the WCE-based approach must generate a new energy cost model on environmental change, which requires time-consuming energy measurement; to the best of our knowledge, all the models available in the literature assume a room temperature. (2) Without updating the energy cost model on temperature change, the approach [89] can generate vulnerable regions causing the stagnation and wrong recovery. (3) Even if the temperature goes down, the WCE-based approach still suffers correctness issue—because some NVM locations might have already been corrupted by antidependent stores.

In contrast to the prior work, our compiler achieves stagnation-free region formation based on our time-based SAT modeling, which is robust against temperature change. That is, we calculate the SAT taking into account the highest temperature, in which the MSP430 can operate correctly, by referring to its manual that specifies the maximum current on the highest temperature. To have the SAT cover the worst-case scenario (under the highest temperature), we consider the MCU as a resistance-capacitor (RC) circuit with the maximum current in mind (Sec. IV-A). As a result, PFI+ROCKCLIMB does not suffer the correctness issue.

Second, other prior works introduce the JIT mechanism [17], [51], [67] that makes a checkpoint for all registers when a system is about to die by monitoring energy availability in the energy buffer. Since the mechanism allows energy harvesting systems to resume an power-interrupted program from the power failure point (after restoring the registers from the NVM checkpoint storage), they can make a forward progress across power failure without the stagnation problem.

Unfortunately, any prior work using JIT checkpointing should pay a high energy/run-time cost. In particular, JIT checkpointing comes with high register pressure—causing significant slowdown due to the resulting NVM accesses—because the register restoration is performed through with a load instruction that uses 2 registers as operands. That is, the prior works [17], [51], [67] must dedicate 3 registers so that

its recovery runtime can use them to restore other registers in the wake of power failure.

More seriously, to recognize the impending power failure and ensure failure-atomic JIT checkpointing, the prior works require an additional energy budget by having a large size capacitor or increasing nominal voltage. Unfortunately, such a budget leads to performance degradation, since the additional energy budget can only be used for checkpointing purpose (not for computation at all). Also, it causes energy harvesting systems to take a longer time than original design to secure the high voltage enough to reboot (resuming the microcontroller) across power failure. Unlike the prior works relying on JIT checkpointing, our proposal does not require such additional energy budget but can still eliminate the stagnation without expensive hardware support.

Third, Choi *et al* introduces a stagnation-free real time operating system scheme called Elastin [15]. It leverages a watchdog timer based checkpoint mechanism which checkpoints all registers when the timer is expired. While Elastin also uses a capacitor leakage model as with PFI, the model is used for a different purpose, i.e., Elastin compares the resulting leakage time with the actual progress time to check if the capacitor is not under a security/reliability attack.

During a checkpoint interval, however, Elastin tracks every memory update on a per-page basis and logs the original page as a copy-on-write mechanism to achieve crash consistency. Since current energy harvesting devices do not support special hardware components such as memory management unit for copy-on-write, the scheme ends up inserting expensive code for tracking memory writes into an original binary, causing more than 2x slowdown. Furthermore, since Elastin must roll back to the recent checkpoint point in the wake of power outages, it also suffer a significant re-execution cost. In contrast, PFI+ROCKCLIMB can achieve stagnation-free intermittent computation without rollback or memory log.

VIII. SUMMARY

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

ACKNOWLEDGMENT

We thank the anonymous reviewers and our shepherd for their valuable comments. At Purdue, this work was supported by NSF grants 1750503 (CAREER) and 1814430.

REFERENCES

- [1] P. Sparks, "White paper: The economics of a trillion connected devices," 2017.
- [2] A. Dehghan-Manshadi, M. Bermingham, M. Dargusch, D. StJohn, and M. Qian, "Metal injection moulding of titanium and titanium alloys: Challenges and recent development," *Powder Technology*, vol. 319, pp. 289–301, 2017.
- [3] H. G. Lee and N. Chang, "Powering the iot: Storage-less and converter-less energy harvesting," in *Proceedings of Asia and South Pacific Design Automation and Conference (ASP-DAC)*, 2015.
- [4] J. Hester, K. Storer, L. Sitanayah, and J. Sorber, "Towards a language and runtime for intermittently-powered devices," *sleep*, vol. 9, p. 10, 2016.
- [5] W.-M. Chen, T.-W. Kuo, and P.-C. Hsiu, "Heterogeneity-aware multi-core synchronization for intermittent systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 20, no. 5s, pp. 1–22, 2021.
- [6] C.-K. Kang, H. R. Mendis, C.-H. Lin, M.-S. Chen, and P.-C. Hsiu, "Everything leaves footprints: Hardware accelerated intermittent deep inference," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 11, pp. 3479–3491, 2020.
- [7] S. Lee, B. Islam, Y. Luo, and S. Nirjon, "Intermittent learning: On-device machine learning on intermittently powered system," *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, vol. 3, no. 4, pp. 1–30, 2019.
- [8] H. R. Mendis, C.-K. Kang, and P.-c. Hsiu, "Intermittent-aware neural architecture search," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 20, no. 5s, pp. 1–27, 2021.
- [9] B. Islam and S. Nirjon, "Zygarde: Time-sensitive on-device deep inference and adaptation on intermittently-powered systems," *arXiv preprint arXiv:1905.03854*, 2019.
- [10] A. C. Kiwan Maeng and B. Lucia, "Alpaca: intermittent execution without checkpoints," in *Proc. ACM Program. Lang.*, 1, OOPSLA, Article 96, October 2017.
- [11] J. V. D. Woude and M. Hicks, "Intermittent computation without hardware support or programmer intervention," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*.
- [12] S. S. Bagsorkhi and C. Margiolas, "Automating efficient variable-grained resiliency for low-power iot systems," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pp. 38–49, 2018.
- [13] K. Maeng and B. Lucia, "Adaptive dynamic checkpointing for safe efficient intermittent computing," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, (Carlsbad, CA), pp. 129–144, USENIX Association, 2018.
- [14] 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*, pp. 399–412, 2019.
- [15] 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)*, pp. 331–344, IEEE, 2019.
- [16] A. Colin and B. Lucia, "Termination checking and task decomposition for task-based intermittent programs," in *Proceedings of the 27th International Conference on Compiler Construction*, 2018.
- [17] K. Maeng and B. Lucia, "Supporting peripherals in intermittent systems with just-in-time checkpoints," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 1101–1116, ACM, 2019.
- [18] A. Colin, G. Harvey, A. P. Sample, and B. Lucia, "An energy-aware debugger for intermittently powered systems," *IEEE Micro*, vol. 37, no. 3, pp. 116–125, 2017.
- [19] J. R. Smith, *Wirelessly Powered Sensor Networks and Computational RFID*. New York, NY, USA: Springer, 2013.
- [20] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of program analysis*. Springer Science & Business Media, 2004.
- [21] K. Ma, X. Li, S. Li, Y. Liu, J. J. Sampson, Y. Xie, and V. Narayanan, "Nonvolatile processor architecture exploration for energy-harvesting applications," *IEEE Micro*, vol. 35, no. 5, pp. 32–40, 2015.
- [22] K. Ma, X. Li, J. Li, Y. Liu, Y. Xie, J. Sampson, M. T. Kandemir, and V. Narayanan, "Incidental computing on iot nonvolatile processors," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 204–218, ACM, 2017.
- [23] F. Su, Y. Liu, Y. Wang, and H. Yang, "A ferroelectric nonvolatile processor with 46 μ s system-level wake-up time and 14 μ s sleep time for energy harvesting applications," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 64, no. 3, pp. 596–607, 2017.
- [24] Y. Wang, Y. Liu, S. Li, D. Zhang, B. Zhao, M.-F. Chiang, Y. Yan, B. Sai, and H. Yang, "A 3 μ s wake-up time nonvolatile processor based on ferroelectric flip-flops," in *2012 Proceedings of the ESSCIRC (ESSCIRC)*, pp. 149–152, IEEE, 2012.
- [25] Y. Liu, Z. Li, H. Li, Y. Wang, X. Li, K. Ma, S. Li, M.-F. Chang, S. John, Y. Xie, *et al.*, "Ambient energy harvesting nonvolatile processors: from circuit to system," in *Proceedings of the 52nd Annual Design Automation Conference*, p. 150, ACM, 2015.
- [26] A. Colin and B. Lucia, "Chain: tasks and channels for reliable intermittent programs," *ACM SIGPLAN Notices*, vol. 51, no. 10, pp. 514–530, 2016.
- [27] B. Lucia and B. Ransford, "A simpler, safer programming and execution model for intermittent systems," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '15*, (New York, NY, USA), pp. 575–585, ACM, 2015.
- [28] A. Colin, G. Harvey, B. Lucia, and A. P. Sample, "An energy-interference-free hardware-software debugger for intermittent energy-harvesting systems," *ACM SIGOPS Operating Systems Review*, vol. 50, no. 2, pp. 577–589, 2016.
- [29] T. Instruments, "Msp430fr family of ultra low-power microcontrollers," 2015. http://www.ti.com/lscds/ti/microcontrollers_16-bit_32-bit/msp/ultra-low_power/msp430frxx_fram/what_is_fram.page.
- [30] M. Hicks, "Clank: Architectural support for intermittent computation," in *In Proceedings of ISCA '17*, ACM, 2017.
- [31] B. Islam and S. NIRJON, "Zygarde: Time-sensitive on-device deep inference and adaptation on intermittently-powered systems," *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies (IMWUT/UBICOMP'20)*, vol. 4, no. 3, pp. 1–29, 2020.
- [32] B. Islam, S. Lee, and S. Nirjon, "Time-aware deep intelligence on batteryless systems," *Brief Presentations Proceedings (RTAS 2019)*, p. 5, 2019.
- [33] Y. Luo and S. Nirjon, "Smarton: Just-in-time active event detection on energy harvesting systems," *arXiv preprint arXiv:2103.00749*, 2021.
- [34] B. Islam and S. Nirjon, "Scheduling computational and energy harvesting tasks in deadline-aware intermittent systems," in *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 95–109, IEEE, 2020.
- [35] Y.-C. Lin, P.-C. Hsiu, and T.-W. Kuo, "Autonomous i/o for intermittent iot systems," in *2019 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pp. 1–6, IEEE, 2019.
- [36] C.-K. Kang, C.-H. Lin, P.-C. Hsiu, and M.-S. Chen, "Homerun: Hw/sw co-design for program atomicity on self-powered intermittent systems," in *Proceedings of the International Symposium on Low Power Electronics and Design*, pp. 1–6, 2018.
- [37] W.-M. Chen, T.-S. Cheng, P.-C. Hsiu, and T.-W. Kuo, "Value-based task scheduling for nonvolatile processor-based embedded devices," in *2016 IEEE Real-Time Systems Symposium (RTSS)*, pp. 247–256, IEEE, 2016.
- [38] A. Rodriguez Arreola, D. Balsamo, A. K. Das, A. S. Weddell, D. Brunelli, B. M. Al-Hashimi, and G. V. Merrett, "Approaches to transient computing for energy harvesting systems: A quantitative evaluation," in *Proceedings of the 3rd International Workshop on Energy Harvesting & Energy Neutral Sensing Systems, ENSys '15*, (New York, NY, USA), pp. 3–8, ACM, 2015.
- [39] W. Zhang, S. Liu, M. Lv, Q. Chen, and N. Guan, "Intermittent computing with efficient state backup by asynchronous dma," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 543–548, IEEE, 2021.
- [40] 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*, pp. 517–529, 2021.
- [41] 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)*, pp. 525–538, IEEE, 2020.
- [42] 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*, pp. 654–666, 2021.

- [43] S. Liu, W. Zhang, M. Lv, Q. Chen, and N. Guan, "Latics: A low-overhead adaptive task-based intermittent computing system," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 11, pp. 3711–3723, 2020.
- [44] 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, pp. 1–26, 2016.
- [45] 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, LCTES'15*, (New York, NY, USA), ACM, 2015.
- [46] 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*, pp. 228–239, IEEE, 2016.
- [47] Q. Liu, C. Jung, D. Lee, and D. Tiwari, "Low-cost soft error resilience with unified data verification and fine-grained recovery for acoustic sensor based detection," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12, IEEE, 2016.
- [48] 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)*, pp. 1–6, IEEE, 2016.
- [49] J. Zeng, J. Choi, X. Fu, A. P. Shreepathi, D. Lee, C. Min, and C. Jung, "Replaycache: Enabling volatile caches for energy harvesting systems," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 170–182, 2021.
- [50] 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*, pp. 989–1004, 2020.
- [51] K. Maeng and B. Lucia, "Adaptive low-overhead scheduling for periodic and reactive intermittent execution," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 1005–1021, 2020.
- [52] H. Song, S. Kim, J. H. Kim, E. J. Park, and S. H. Noh, "First responder: Persistent memory simultaneously as high performance buffer cache and storage," in *2021 USENIX Annual Technical Conference (USENIX ATC'21)*, pp. 839–853, 2021.
- [53] O. Kaiyakhmet, S. Lee, B. Nam, S. H. Noh, and Y.-r. Choi, "Slm-db: single-level key-value store with persistent memory," in *17th USENIX Conference on File and Storage Technologies FAST'19*, pp. 191–205, 2019.
- [54] 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)*, pp. 258–270, IEEE, 2018.
- [55] E. Lee, H. Bahn, and S. H. Noh, "Unioning of the buffer cache and journaling layers with non-volatile memory," in *11th USENIX Conference on File and Storage Technologies (FAST'13)*, pp. 73–80, 2013.
- [56] Y. Oh, J. Choi, D. Lee, and S. H. Noh, "Caching less for better performance: balancing cache size and update cost of flash memory cache in hybrid storage systems," in *FAST*, vol. 12, 2012.
- [57] J. Seo, W.-H. Kim, W. Baek, B. Nam, and S. H. Noh, "Failure-atomic slotted paging for persistent memory," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 1, pp. 91–104, 2017.
- [58] E. Lee, J. Kim, H. Bahn, S. Lee, and S. H. Noh, "Reducing write amplification of flash storage through cooperative data management with nvmm," *ACM Transactions on Storage (TOS)*, vol. 13, no. 2, pp. 1–13, 2017.
- [59] S. Muchnick, *Advanced Compiler Design Implementation*. Morgan Kaufmann Publishers, 1997.
- [60] X. Zhang, C. Patterson, Y. Liu, C. Yang, C. J. Xue, and J. Hu, "Low overhead online checkpoint for intermittently powered non-volatile fpgas," in *2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pp. 238–244, IEEE, 2018.
- [61] S. Ahmed, N. A. Bhatti, M. H. Alizai, J. H. Siddiqui, and L. Mottola, "Fast and energy-efficient state checkpointing for intermittent computing," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 19, no. 6, pp. 1–27, 2020.
- [62] W.-M. Chen, T.-W. Kuo, and P.-C. Hsiu, "Enabling failure-resilient intermittent systems without runtime checkpointing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 12, pp. 4399–4412, 2020.
- [63] W.-M. Chen, P.-C. Hsiu, and T.-W. Kuo, "Enabling failure-resilient intermittently-powered systems without runtime checkpointing," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6, IEEE, 2019.
- [64] W.-M. Chen, Y.-T. Chen, P.-C. Hsiu, and T.-W. Kuo, "Multiversion concurrency control on intermittent systems," in *ICCAD*, pp. 1–8, 2019.
- [65] M. de Kruijf and K. Sankaralingam, "Idempotent code generation: Implementation, analysis, and evaluation," in *Code Generation and Optimization (CGO)*, 2013 IEEE/ACM International Symposium on, pp. 1–12, IEEE, 2013.
- [66] 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)*, pp. 147–152, IEEE, 2016.
- [67] 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 *VLSI Design and 2014 13th International Conference on Embedded Systems, 2014 27th International Conference on*, pp. 330–335, IEEE, 2014.
- [68] "Msp430fr5994launchpad development kit (mspexp430fr5994)," Mar 2016.
- [69] M. Karimi, H. Choi, Y. Wang, Y. Xiang, and H. Kim, "Real-time task scheduling on intermittently-powered batteryless devices," *IEEE Internet of Things Journal*, 2021.
- [70] A. Sinha and A. P. Chandrakasan, "Jouletrack-a web based tool for software energy profiling," in *In Proceedings of the 38nd Annual Design Automation Conference, DAC '01*, 2001.
- [71] J. San Miguel *et al.*, "The eh model: Analytical exploration of energy-harvesting architectures," *IEEE Computer Architecture Letters*, 2018.
- [72] S. Chattopadhyay, L. K. Chong, A. Roychoudhury, T. Kelter, P. Marwedel, and H. Falk, "A unified wcet analysis framework for multi-core platforms," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 4s, p. 124, 2014.
- [73] A. Banerjee, S. Chattopadhyay, and A. Roychoudhury, "Precise micro-architectural modeling for wcet analysis via ai+ sat," in *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 87–96, IEEE, 2013.
- [74] B. Blackham, Y. Shi, S. Chattopadhyay, A. Roychoudhury, and G. Heiser, "Timing analysis of a protected operating system kernel," in *2011 IEEE 32nd Real-Time Systems Symposium*, pp. 339–348, IEEE, 2011.
- [75] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, *et al.*, "The worst-case execution-time problem—overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 3, p. 36, 2008.
- [76] "Msp430 family instruction set summary," 2006.
- [77] J.-M. Chang and M. Pedram, "Register allocation and binding for low power," in *Proceedings of the 32nd annual ACM/IEEE Design Automation Conference*, pp. 29–35, ACM, 1995.
- [78] K. Ma, Y. Zheng, S. Li, K. Swaminathan, X. Li, Y. Liu, J. Sampson, Y. Xie, and V. Narayanan, "Architecture exploration for ambient energy harvesting nonvolatile processors," in *Proceedings of 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, HPCA '15, (Piscataway, NJ, USA), pp. 526–537, IEEE Press, 2015.
- [79] C. Lattner *et al.*, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '04, 2004.
- [80] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pp. 3–14, IEEE, 2001.
- [81] Q. Liu, X. Wu, L. Kittinger, M. Levy, and C. Jung, "Benchprime: Effective building of a hybrid benchmark suite," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 16, no. 5s, p. 179, 2017.
- [82] "Evaluation board for p2110 powerharvester."

- [83] K. S. Yildirim, A. Y. Majid, D. Patoukas, K. Schaper, P. Pawelczak, and J. Hester, "Ink: Reactive kernel for tiny batteryless sensors," in *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*, pp. 41–53, 2018.
- [84] J. de Winkel, C. Delle Donne, K. S. Yildirim, P. Pawelczak, and J. Hester, "Reliable timekeeping for intermittent computing," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 53–67, 2020.
- [85] V. Kortbeek, K. S. Yildirim, A. Bakar, J. Sorber, J. Hester, and P. Pawelczak, "Time-sensitive intermittent computing meets legacy software," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 85–99, 2020.
- [86] M. de Kruijf and K. Sankaralingam, "Idempotent processor architecture," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 140–151, ACM, 2011.
- [87] B. Ransford, J. Sorber, and K. Fu, "Mementos: System support for long-running computation on rfid-scale devices," *Acm Sigplan Notices*, vol. 47, no. 4, pp. 159–170, 2012.
- [88] M. Xie, C. Pan, M. Zhao, Y. Liu, C. J. Xue, and J. Hu, "Avoiding data inconsistency in energy harvesting powered embedded systems," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 23, no. 3, pp. 1–25, 2018.
- [89] B. Yarahmadi and E. Rohou, "Compiler optimizations for safe insertion of checkpoints in intermittently powered systems," in *International Conference on Embedded Computer Systems*, pp. 169–185, Springer, 2020.