SweepCache: Intermittence-Aware Cache on the Cheap

Yuchen Zhou
Purdue University
West Lafayette, IN, USA
zhou1166@purdue.edu

Jianping Zeng
Purdue University
West Lafayette, IN, USA
zeng207@purdue.edu

Jungi Jeong*
Google
Mountain View, CA, USA
jungijeong@google.com

Jongouk Choi
University of Central Florida
Orlando, FL, USA
jongouk.choi@ucf.edu

Changhee Jung
Purdue University
West Lafayette, IN, USA
chjung@purdue.edu

ABSTRACT

This paper presents SweepCache, a new compiler/architecture co-design scheme that can equip energy harvesting systems with a volatile cache in a performant yet lightweight way. Unlike prior just-in-time checkpointing designs that persists volatile data just before power failure and thus dedicates additional energy, SweepCache partitions program into a series of recoverable regions and persists stores at region granularity to fully utilize harvested energy for computation. In particular, SweepCache introduces persist buffer—as a redo buffer resident in nonvolatile memory (NVM)—to keep the main memory consistent across power failure while persisting region's stores in a failure-atomic manner. Specifically, for write-backs during region execution, SweepCache saves their cachelines to the persist buffer. At each region end, SweepCache first flushes dirty cachelines to the buffer, allowing the next region to start with a clean cache, and then moves all buffered cachelines to the corresponding NVM locations. In this way, no matter when power failure occurs, the buffer contents or their memory locations always remain intact, which serves as a basis for correct recovery. To hide the persistence delay, SweepCache speculatively starts a region right after the prior region finishes its execution—as if its stores were already persisted—with the two regions having their own persist buffer, *i.e.,* dual-buffering. This region-level parallelism helps SweepCache to achieve the full potential of a high-performance data cache. The experimental results show that compared to the original cache-free nonvolatile processor, SweepCache delivers speedups of 14.60x and 14.86x—outperforming the state-of-the-art work by 3.47x and 3.49x—for two representative energy harvesting power traces, respectively.

CCS CONCEPTS

• **Computer systems organization** → **Embedded systems**.

KEYWORDS

compiler/architecture co-design, energy harvesting, failure-atomic

ACM Reference Format:
Yuchen Zhou, Jianping Zeng, Jungi Jeong*, Jongouk Choi, and Changhee Jung. 2023. SweepCache: Intermittence-Aware Cache on the Cheap. In *56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23), October 28-November 1, 2023, Toronto, ON, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3613424.3623781

1 INTRODUCTION

Energy harvesting systems [79] are becoming more prevalent in a wide range of applications, *e.g.,* vehicle tire pressure sensing, health and wellness monitoring [8, 18, 21, 75], wearable computing [10, 46, 63, 64], just to name a few. However, due to the unstable nature of the energy sources, *e.g.,* radio frequency (RF) and WiFi, these applications experience frequent and unpredictable power failure during program execution—thus being called intermittent computation.

To resist frequent power failure, previous studies proposed a nonvolatile processor (NVP) [57, 59, 60, 87]. It provides an illusion of continuous execution to applications by leveraging both byte-addressable nonvolatile memory (NVM) as the main memory and voltage monitor based volatile data checkpointing. Whenever the monitor detects a voltage drop below a predefined threshold, *i.e.,* a sign of impending power failure, NVP is interrupted to checkpoint all registers before the failure; this is so-called just-in-time (JIT) checkpointing. In the wake of the failure, NVP restores the registers and continues to progress from the interruption point—as if program had never been power-interrupted.

Nonetheless, the performance of NVP is limited by NVM accesses that are the most expensive in terms of both energy consumption and instruction latency. While caching hot data makes more progress under the same amount of harvested energy, equipping NVP with an SRAM cache puts significant pressure on crash consistency mechanisms [97]. For example, upon power failure, the volatile cache loses all the data including dirty cachelines, and therefore only checkpointing the registers is not sufficient for correct recovery.

To address this problem, previous designs checkpoint and restore not only the register file but also the cache across power failure; NVSRAM cache [11, 25, 48, 49, 65, 83, 85] deploys a nonvolatile counterpart to back up the whole SRAM cache right before power failure. Since checkpointing the entire cache consumes too much harvested energy thus limiting forward progress, Liu *et al.*

*This work was done while the author was at Purdue.

devised a partial backup strategy [58] while others exploited hybrid cache architecture that checkpoints only dirty cachelines [86, 94]. Meanwhile, rather than checkpointing cachelines, ReplayCache re-executes any potentially unpersisted stores to restore consistent memory states before resuming power-interrupted program [97]—at the cost of persisting each store during program execution, *i.e.,* giving up on persist coalescing [37, 77].

However, all prior cache-enabled designs rely on JIT checkpointing of registers which incurs nontrivial hardware complexity [26], *e.g.,* the voltage monitor, the backup/restoration signal handling logic, the nonvolatile flip-flops (NVFFs[1]) that should be laid out next to the volatile registers for fast data movement [44, 47, 71, 74, 78], and the NVFFs controller; NVSRAM approaches require additional complexity for JIT checkpointing of the SRAM cache [58, 86, 94]. Furthermore, the JIT checkpointing requires that the backup should be performed in a failure-atomic way without power interruption, which would otherwise fail to achieve crash consistency. Unfortunately, this forces energy harvesting systems to dedicate a large amount of hard-won energy for the failure-atomic backup all the time—since power failure can occur at any time—leaving only a portion of harvested energy for computation. More importantly, JIT checkpointing might suffer capacitor degradation [13, 80], putting the failure-atomicity at stake, and a long voltage detection delay (Section 2.2).

To this end, this paper proposes SweepCache, a novel JIT-checkpoint-free design that achieves lightweight yet performant intermittent computation for cache-enabled energy harvesting systems by using intelligent compiler/architecture interaction. SweepCache's compiler partitions program into a series of recoverable regions—with their live-out registers checkpointed via store instructions—so that the region boundary serves as a recovery point for power failure. Then, to facilitate power failure recovery, SweepCache architecture runs them through region-level persistence, *i.e.,* all stores of a region including the checkpoint stores must be persisted to NVM before the next region starts.

In case a region is power-interrupted, SweepCache holds all data of its stores in a NVM-resident buffer—we call persist buffer—before persisting them to the main memory (NVM). More precisely, during region execution, all cache writebacks (*i.e.,* dirty cacheline evictions) are first quarantined in the persist buffer; so it acts like a redo buffer for protecting the main memory against the stores of power-interrupted regions. Thus, no matter when power failure occurs, either the buffer contents or their target NVM locations always remain intact, which serves as a basis for correct power failure recovery. Especially for region-level persistence, when program control reaches each region end, SweepCache flushes all dirty cachelines to the persist buffer and then moves them to the NVM. This effectively makes each region begin with a clean cache lacking dirty cachelines [2], which allows any power interruption in the middle of a region to be recovered by simply restarting the interrupted region without worrying about persisting prior regions.

Although the region-level persistence simplifies the recovery protocol, there are a couple of challenges that should be addressed to put SweepCache into practice. First, the persistence delay at each region end incurs significant performance degradation because the next region cannot start until the previous one becomes fully persistent in NVM, which is the case for ReplayCache though it exploits *in-region parallelism* by ILP. To minimize the delay across regions, SweepCache introduces *region-level parallelism* overlapping the persistence latency with the execution of the following region. In other words, the next region always speculatively starts without delay as if the prior region were persisted. In addition, an inherent benefit of region-level parallelism is that it hides the latency for persisting the data (*e.g.,* registers and dirty cachelines) that prior schemes [11, 25, 31, 48, 49, 58, 65, 85, 97, 98] must pay for JIT checkpointing when power is about to be cut off.

Second, since the persist buffer exists on the data path, the way to handle load cache misses becomes significantly complicated. They should search the buffer—in case the latest value is held therein—before accessing the NVM, which increases the critical path of handling the cache misses. This is particularly problematic in terms of performance in that the persist buffer is allocated in NVM suffering from the same latency and bandwidth issues. To overcome this challenge, SweepCache makes an important observation that the persist buffer is empty most of the time due to the relatively short region size and the low cache miss rate of the benchmarks tested. The implication is that the buffer search can often be bypassed to shorten the critical path of load misses.

Taking that into account, SweepCache devises a single bit called empty-bit to figure out if the persist buffer is currently empty or populated. That is, load misses should first consult the empty-bit to decide whether to bypass the buffer search. According to our experimental results, this simple bit consultation allows SweepCache to direct 99% of load misses to NVM without accessing the persist buffer, thereby realizing the full potential of a volatile cache for high-performance intermittent computation. The experiment with 26 benchmarks from Mibench [24] and Mediabenchs [45] shows that compared to the original cache-free NVP, SweepCache achieves speedups of 14.60x and 14.86x—outperforming the state-of-the-art work (ReplayCache) by 3.47x and 3.49x—for two representative energy harvesting power traces, respectively.

2 BACKGROUND AND MOTIVATION

2.1 Basics of Energy Harvesting Systems

Energy harvesting systems collect ambient energy, *e.g.,* RF and Wi-Fi, in a small capacitor [6, 7, 13, 14, 16, 39, 52, 81, 82]. However, due to the unstable energy source and the lack of battery, the systems undergo frequent power failure [3, 14, 16, 61, 62, 89]. While they employ NVM as the main memory to survive power failure, it results in data loss of volatile registers.

Thus, prior studies proposed *JIT checkpointing* that persists the register values right before power failure [17, 31, 59, 60]. For example, NVP checkpoints all registers in NVFF closely integrated into the volatile register file [59, 60], while QuickRecall [31] writes the registers to NVM. As shown in Figure 1(a), they both leverage a voltage monitor to detect impending power failure. To be more specific, if the voltage becomes lower than the predefined threshold (*e.g.,* V_{bk}), NVP is interrupted and copies the register values to NVFF to save the architectural state. When the voltage comes back to a

[1]Some prior work such as QuickRecall [31] checkpoints registers to NVM (not NVFFs), but it is more time- and energy-consuming [93].
[2]The name SweepCache is inspired by its action of having the cache swept clean between regions.

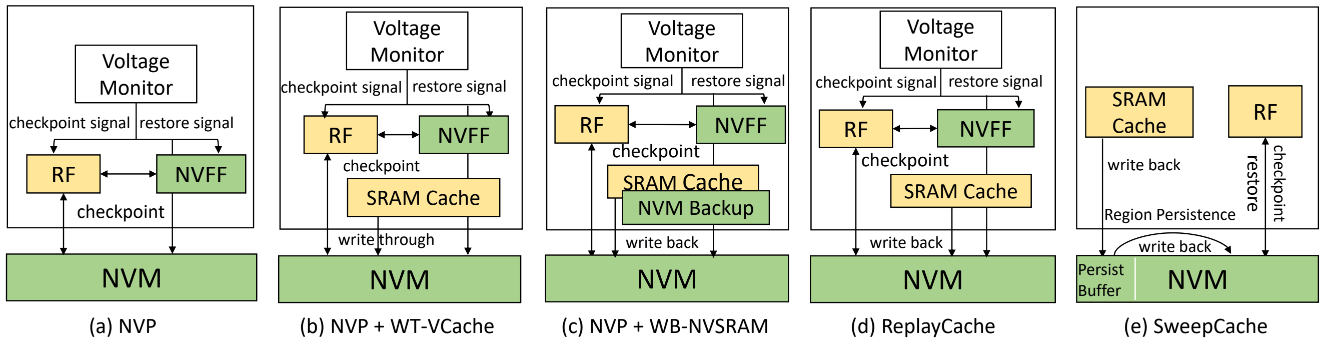


Figure 1: Architecture of energy harvesting systems; green corresponds to non-volatile parts while yellow to volatile parts.

certain level (*e.g.,* V_{rt}), the register values are restored by using the checkpointed values in NVFF, and NVP then resumes exactly from where it had been interrupted.

Since NVP uses NVM as the main memory without a volatile cache in between, all committed stores are guaranteed to be persistent in NVM, *i.e.,* a store is the granularity of failure atomicity in the architecture. By combining the JIT checkpointing and the nonvolatile main memory, cache-free energy harvesting systems such as NVPs can guarantee crash consistency even in the presence of frequent power outages.

However, energy harvesting systems often refrain from exploiting a volatile cache mainly due to its crash consistency challenge. Nevertheless, equipping them with the cache has a high potential to improve the performance and energy efficiency [11, 48, 49, 97], *e.g.,* cache-enabled NVP can deliver further forward execution progress by avoiding NVM accesses on cache hits compared to the original cache-free design. Therefore, enabling caches can open new use cases of energy harvesting systems and put them into practice.

2.2 Enabling Caches with JIT Checkpointing

Recent work has studied the problem of enabling volatile SRAM caches for energy harvesting systems [4, 11, 25, 48, 49, 65, 83, 85, 97]. However, the problem turns out to be challenging because volatile caches may lead to crash inconsistency—unless they are backed up and restored across a power outage. For example, at the moment of an outage, all cache contents disappear, including dirty cachelines. Thus, merely restoring register values is not sufficient for correct recovery—resulting in inconsistent memory states across the outage.

There have been multiple approaches to dealing with such a memory inconsistency issue for cache-enabled energy harvesting systems. First, a straightforward but naive solution is to leverage a volatile *write-through* cache shown in Figure 1(b). Here, both write-through cache and NVM always maintain the same value for every committed store instruction, *i.e.,* it is possible to recover the consistent program state without worrying about the loss of volatile cache data. However, the write-through cache pays for a high persistence overhead in that each store instruction cannot be committed until the corresponding cacheline is written to NVM. Such a long store latency is particularly harmful to energy harvesting systems since they do not use out-of-order pipelines that can tolerate the latency. Meanwhile, the frequent NVM writes consume a large amount of harvested energy.

As shown in Figure 1(c), the second approach (*NVSRAM*) uses a volatile *write-back* cache with the nonvolatile counterpart as a backup storage [11, 25, 48, 49, 58, 65, 83, 85]. NVSRAM, combined with JIT checkpointing, flushes all SRAM cache contents (or only dirty cachelines) to the nonvolatile counterpart before impending power failure, thus being free from the memory inconsistency problem. However, even backing up only dirty cachelines requires NVSRAM to reserve a sufficient amount of energy which can afford the whole-cache backup to guarantee failure-atomic JIT checkpointing in case all cachelines are dirty. Another problem is that for swift backup/restoration, NVSRAM resorts to parallel data transfer; this ends up with non-trivial energy consumption and high inrush current, which may cause significant energy/reliability issues. Moreover, the NVM counterpart also results in extra area costs, *e.g.,* a 32KB NVSRAM cache leads to over a 4.8x larger chip area cost [58].

The state-of-the-art work, *ReplayCache* [97], enables volatile caches in a more advanced way than prior work. Unlike NVSRAM, ReplayCache does not need a NVM backup for the SRAM cache as shown in Figure 1(d). Instead, ReplayCache leverages so-called store integrity; it preserves the operands of each store so that potentially unpersisted stores left behind power failure can be replayed for recovery. To achieve this, the compiler partitions program into a series of regions where the store integrity is enforced, *i.e.,* none of store registers are overwritten in each region. In the wake of power failure that interrupted a region, ReplayCache first replays its unpersisted stores to keep NVM states up-to-date and then resumes program from the interruption point. In this way, ReplayCache resolves the memory inconsistency problem.

Yet, for each region to fully use the register file without breaking the store integrity of the prior region(s), ReplayCache cannot start a region until all stores of the prior region are persisted. To this end, ReplayCache persists them asynchronously using `clwb` during region execution with a store fence placed at the end of each region. Apart from the possible persistence delay between regions, ReplayCache loses persist coalescing [37, 77]—in that every single store is followed by the 64-byte cacheline flush (`clwb`)—causing high write amplification and energy consumption. Moreover, to ensure correct recovery, ReplayCache should load the data from NVFF (or NVM) to execute a recovery block for replaying stores sequentially, which leads to slow recovery.

In particular, a common problem for the aforementioned prior schemes is that they all rely on JIT checkpointing which incurs aforestated hardware complexity. More importantly, the prior work must set voltage thresholds high to ensure the failure-atomic backup, which leaves less energy for computation and therefore degrades performance. It is also important to note that, JIT checkpointing is vulnerable to a capacitor degradation phenomenon as demonstrated by recent work [13, 80], *e.g.,* the capacitor may deliver only 90% of its original output in roughly 7 days with typical power traces. This implies that the voltage threshold should be set higher than usual for JIT backup to work safely in case the capacitor degrades over time. Unfortunately, such a high voltage margin incurs a huge performance overhead, *e.g.,* in our evaluation, a 20% threshold increase leads to a 1.4x slowdown while a 40% increase to a 2.5x slowdown.

Finally, the voltage monitor of JIT checkpointing usually must detect two different voltage thresholds for backup and restoration where $V_{restore} > V_{backup}$ according to MPPT (maximum power point tracking) [91]; instead, SweepCache only needs a single voltage threshold to indicate an appropriate recovery point, *i.e.,* when to reboot. The takeaway is that the prior schemes rely on a more complex voltage detection circuit than SweepCache's single-threshold voltage comparator, thereby causing longer voltage detection delays—also known as propagation delays. For example, in prior work [23, 58, 87, 97], the voltage detector has 1.5 us (T_{phl}) and 10.3 us (T_{plh}) propagation delays with at least 20 uA current supply whereas a simple same-year-technology voltage comparator [28] has only 0.88 us (T_{phl}) and 1.1 us (T_{plh}) propagation delays with 12 uA current supply.

3 SWEEPCACHE APPROACH

SweepCache is JIT-checkpoint-free and lightweight. It pursues performant cache-enabled energy harvesting systems that consume the majority of harvested energy for computation, instead of reserving the energy for JIT checkpointing. However, it is challenging to provide crash consistency for both a register file and a cache without the JIT checkpointing that facilitates the checkpoint/recovery to a large extent.

To address this challenge, SweepCache proposes a compiler and architecture co-design that performs region-level persistence and failure recovery at a low cost. With SweepCache's compiler, the input program is partitioned into a series of regions where live-out registers are checkpointed via store instructions, and the region boundary serves as a recovery point of power failure (Section 3.1). Besides, as shown Figure 1(e), SweepCache presents a persist buffer. It acts as a redo buffer not only to keep the main memory from the stores of a power-interrupted region but also to delegate the persistence of the data being stored in each region at its end, letting the pipeline keep executing the following regions to hide the persistence latency (Section 3.2 & Section 3.3). Across power failure, SweepCache consults the persist buffer, if necessary depending on where the program is interrupted (*e.g.,* in-region or between regions), to resume the interrupted program correctly (Section 3.4). Figure 2 shows the design overview of SweepCache.

3.1 Compiler-Assisted Register Checkpointing

To eliminate expensive hardware structures for JIT checkpointing volatile registers, SweepCache leverages compiler techniques to transform program source code so that register values are checkpointed (via stores) to NVM in a region granularity and thus can be used for region-level failure recovery.

Unfortunately, it is hard to design such a checkpoint-based power failure resilient scheme because of two problems: (1) determining which registers should be checkpointed to NVM and (2) where to put register checkpoints. To address the problems, SweepCache leverages the persist buffer directed region formation [16, 27, 35, 99]; it partitions the program to a series of regions (a sequence of instructions regardless of branches) so that the persist buffer never overflows during the execution of each region with its live-out [2] registers checkpointed. As shown in Figure 2(a) where the *stores* are normal stores but *ckpt stores* are register checkpointing stores, the number of stores in each region is guaranteed to be smaller than the buffer size (see more in Section 4.1).

3.2 Region-Level Store Persistence

The main obstacle to enabling a volatile cache in energy harvesting systems is that the partial updates from the cache to NVM cause an inconsistency across power failure. Unlike JIT-checkpoint designs, SweepCache has no way of interrupting a program on an outage and restoring the cache in the wake of the outage, thus being unable to resume from the interruption point. Without the luxury of JIT checkpointing, SweepCache instead offers persistence and recovery in a region granularity by deploying the persist buffer as a safety net to prevent partial updates for crash consistency.

To be more specific, SweepCache utilizes the persist buffer as an intermediate between the cache and NVM. That is, spatially, the persistence process of SweepCache is divided into two phases (*s-phase1* and *s-phase2*) depending on actions made to the persist buffer; as shown in Figure 2(b), SweepCache writes back the cache to the persist buffer (①), and then it flushes the buffer to NVM [3](②). But temporally, the persistence process is split into three phases. During the execution of each region, all the writebacks from the cache are piled in the persist buffer (*t-phase1*), keeping NVM intact to protect NVM from them, *i.e.,* partial updates, in case the region is power-interrupted. At the region boundary where the region finishes, SweepCache flushes all the dirty cachelines into the persist buffer (*t-phase2*). Finally, the persist buffer contents are all moved to NVM[4] (*t-phase3*). In case power failure occurs in the middle of *t-phase3*, the persist buffer must be NVM-resident, which would otherwise lose the buffer contents that have not yet been persisted in NVM; such partial NVM updates make it impossible to ensure correct power failure recovery. To deal with the power failure during *t-phase3*, SweepCache restarts *t-phase3* in the wake of the failure with accessing the NVM-resident persist buffer (more details are deferred to Section 4.2).

In this way, SweepCache ensures correct region-level persistence no matter when power failure happens in that either the NVM or

[3]Additional details, *e.g.,* why 2 persist buffers, are deferred to Section 3.3
[4]The buffer is FIFO and can have multiple entries of the same cache line in case of multiple evictions. When SweepCache flushes the buffer to NVM, the younger one always overwrites the old one.

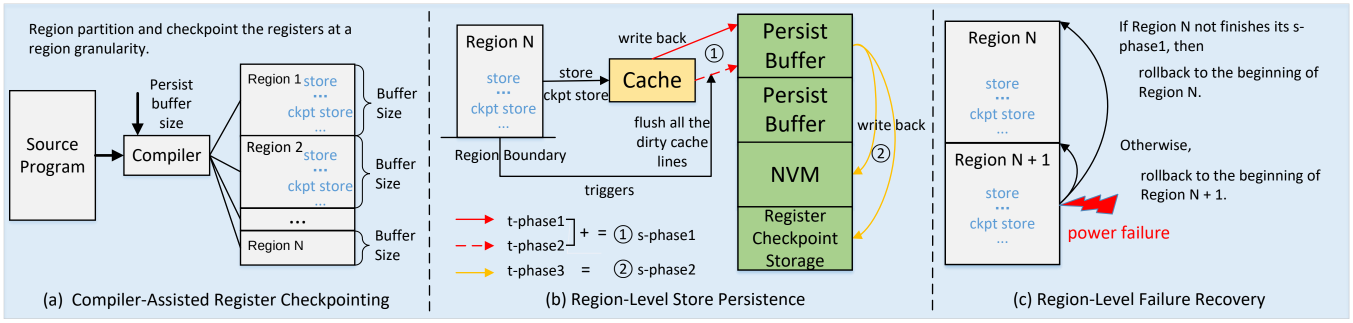

Region partition and checkpoint the registers at a region granularity.
Source Program
Compiler
Persist buffer size
Region 1
Region 2
Region N
store
ckpt store
Buffer Size
(a) Compiler-Assisted Register Checkpointing
Region N
store
ckpt store
Cache
write back
Persist Buffer
Persist Buffer
NVM
Register Checkpoint Storage
flush all the dirty cache lines
Region Boundary
triggers
write back
t-phase1
t-phase2
t-phase3
① s-phase1
② s-phase2
(b) Region-Level Store Persistence
Region N
Region N + 1
If Region N not finishes its s-phase1, then
rollback to the beginning of Region N.
Otherwise,
rollback to the beginning of Region N + 1.
power failure
(c) Region-Level Failure Recovery

Figure 2: The high-level view of SweepCache compiler and architecture

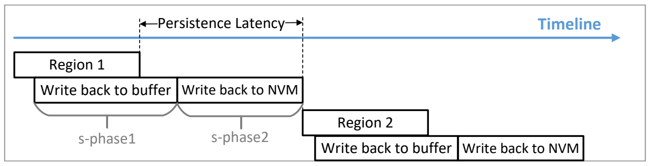
the buffer can always remain consistently available. In particular, for fast data movement from the buffer to NVM, SweepCache leverages direct memory access (DMA), an existing hardware component in commodity energy harvesting systems, *e.g.,* MSP430 series microcontrollers have already adopted DMA [30, 88].

Note that, unlike the traditional 2-phase commit, the 3-phase design of SweepCache effectively offloads the data handled by the first phase of the original 2-phase commit to the *t-phase1*. This design improves the performance since the in-region writebacks during the *t-phase1* can be overlapped with regular program execution, entering the last phase faster than the original 2-phase commit. Yet, to simplify the description, the following sections use two phases (① and ②) in the spatial context to refer to our persistence process.

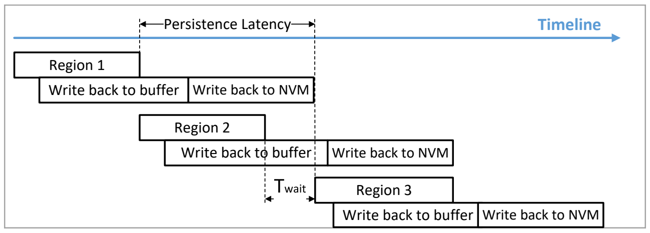
3.3 Region-Level Parallelism

To achieve the region-level persistence, a region cannot start executing until the previous region is persisted, see *persistence latency* in Figure 3(a). One critical issue here is that the prolonged persistence latency at each region boundary can significantly degrade overall performance. To mitigate this issue, SweepCache introduces region-level parallelism, which allows the next region to speculatively execute as if the prior region were already persisted. This helps to hide the persistence latency but ends up with structural hazards as adjacent regions compete for the persist buffer. Specifically, before the prior region finishes its persistence, the next region's speculative execution can overwrite the buffer, thereby making the region-level persistence fail to achieve crash consistency. Ideally, each region should be assigned a separate buffer, which incurs nontrivial hardware costs. Alternatively, the following region should wait for the prior region to complete its persistence, which hurts performance a lot. Fortunately, it turns out that two persist buffers are sufficient to provide high parallelism (Section 6.3) hiding the persistence latency without compromising the crash consistency guarantee.

Figure 3(b) shows how SweepCache handles 3 consecutive regions with the region-level parallelism. Since the 2 persist buffers are assigned to the first 2 regions respectively, Region 2 can start immediately after Region 1 ends due to the absence of the structural hazard. As shown in the figure, SweepCache effectively hides the *persistence latency* of Region 1 by overlapping it with the execution of Region 2. Nonetheless, since SweepCache only has two persist buffers, Region 3 should not start to execute until Region 1 completes its second phase (*s-phase2*) to avoid the structural hazard, *i.e.,* T_{wait} in the figure indicates the actual time Region 3 should wait for[5]. According to experimental results, the efficiency of SweepCache's parallelism is over 91%, *i.e.,* T_{wait} is insignificant most of the time. Such effective region-level parallelism is the basis for SweepCache to checkpoint the register values into NVM in a performant way—though it lacks expensive NVFF/NVSRAM. In contrast, JIT-checkpoint designs cannot hide the latency of persisting both the register file and the cache in their backup stage.



(a) No Parallelism Case



(b) Parallelism Case

Figure 3: Hiding region-level store persistence latency

3.4 Region-Level Failure Recovery

To recover from power failure, SweepCache takes appropriate recovery actions according to where the failure occurs, *i.e.,* before the completion of *s-phase1* or after that as shown in Figure 2(c). Since the persist buffer is a nonvolatile intermediate between the cache and NVM, its persistence status indicates whether a region is persisted or not. If the buffer's persistence process is incomplete at the point of power failure, *i.e.,* it occurs before *s-phase1*, the current region is not persisted yet while not affecting the NVM state (see

[5]To keep sequential store persistence ordering, SweepCache also guarantees that the second phase of a region should not start until the prior one completes its second phase, which might otherwise overwrite NVM locations. However, this situation is rare and thus not depicted in the figure.

Section 4.2 for more details). Thus, in the wake of the power failure, SweepCache discards the buffer contents and rolls back to the beginning of the power-interrupted region for correct recovery. On the other hand, if the buffer's persistence process is already complete upon power failure, *i.e.,* it occurs after *s-phase1*, the region is considered successfully persisted, and therefore SweepCache restarts from the next region's beginning after power comes back as shown in the figure.

4 IMPLEMENTATION DETAILS

4.1 SweepCache Compiler

High-Level Workflow: SweepCache leverages the size of the persist buffer to guide region partitioning with the store threshold equal to the buffer size—conservatively assuming that every store leads to a cacheline writeback. During the partitioning process, SweepCache's compiler counts the number of stores while traversing the program's control flow graph (CFG). Once this count reaches the predefined threshold (*i.e.,* buffer size), a region boundary is introduced to start a new region thereafter. The compiler then analyzes the *live-out* [2] registers of the region and inserts checkpoint stores to save them into a designated register checkpoint storage in NVM. Additionally, the program counter (PC) is saved at the end of the region, which serves as a recovery point in case the next region is power-interrupted. By reading the value of the PC, SweepCache can roll back to the corresponding recovery point to re-execute the interrupted region.

Nevertheless, the region formation is not as simple as sequentially performing region partitioning and live-out register checkpointing because of circular dependence. That is, on the one hand, checkpoint stores influence the number of stores that can be accommodated in a region. On the other hand, the number of stores can change the location of the initial region boundary, possibly leading to more live-out registers being checkpointed, in which case the region boundary might move further, thus forming a circular dependence. To break the dependence, SweepCache's compiler leverages the region formation techniques of prior work [16, 35, 56, 99] like the following.

Region Formation: The compiler first partitions program at callsites and loop headers. Specifically, it inserts a region boundary at all the entry and exit points of functions. Then, to avoid exceeding the store threshold in a loop, a region boundary is also placed at the header of every loop[6], *i.e.,* each loop iteration starts/ends with the boundary, as shown in Figure 4(a); of course, the loop body may need additional boundaries (not shown in the figure) to keep the store count of their regions under the threshold during the CFG traversal. In this way, the number of stores per region is guaranteed not to exceed the threshold even for a long-running loop with many iterations no matter how many stores exist in the loop body.

However, for a small loop body comprised of a few stores, such a boundary at the header could end up with a limited region size. For example, as shown in Figure 4(a), each iteration forms a region with 5 stores, which is way smaller than it should be assuming the store threshold is 10. The problem of such a loop-header boundary is that it might generate many small regions, which could in turn increase the number of checkpoint stores due to additional live-outs across more region boundaries. To tackle this issue, SweepCache's compiler leverages *loop unrolling* to enlarge the region size, as shown in Figure 4(b) where the loop body is unrolled two times making the region size 2x bigger.

[6]The only exception is the loop that has no store in the loop body.

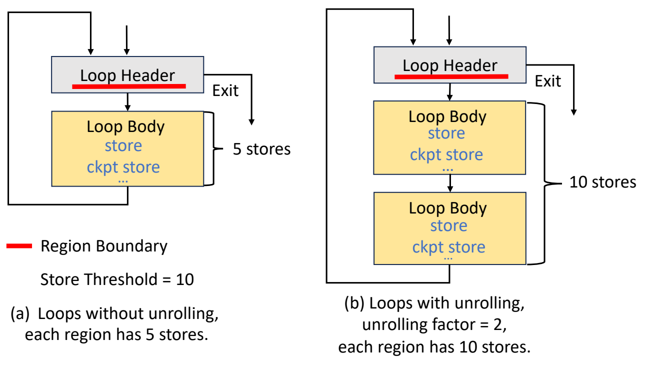


Figure 4: Region formation for loops

After the initial region formation, the compiler proceeds to the step of checkpoint store insertion. In particular, to facilitate this step, the compiler first splits the basic blocks that have region boundaries therein, thereby ensuring that regions always start at the beginning of basic blocks. That is because of the granularity mismatch of the two compiler analyses, *i.e.,* liveness analysis is generally conducted at the level of basic blocks whereas checkpoint store insertion is performed at the granularity of regions. After the basic block splitting, the compiler analyzes the regions to identify the live-out variables and inserts their checkpoint stores—right after the last update point of the variables in each region.

Then, the compiler traverses the CFG again in a topological order trying to combine the initial regions, whose store count is smaller than the threshold, into larger regions as much as possible. This leads to two benefits: (1) extending the region size and (2) often eliminating many checkpoint stores in that the live-out register of their region is no longer live—provided the following region being merged redefines the register. Because of the region combining, the store count of the merged region may exceed the store threshold; if that is the case, the compiler places a new boundary in the middle of the region to guarantee its stores never overflow the persist buffer and recalculates the number of live-out registers of the newly partitioned regions. It is important to note that this merging/repartitioning process is repeated until no region has more stores than the threshold, which resolves the issue of the circular dependence. Nevertheless, it would be a mistake to expect that the resulting regions have the same number of stores as the threshold; recall that the threshold indicates the maximum number of stores in each region.

Checkpoint Storage Management: To facilitate access to register checkpoints during power failure recovery, the compiler maps all registers to a global array with dedicated slots. For example, register $r0$ is mapped to index zero, *i.e.,* a checkpoint store uses fixed destination addresses depending on which register is checkpointed, for them to be easily accessed through an index of the array. This is feasible since the number of architectural registers is predetermined by ISA. In the wake of power failure, SweepCache's recovery runtime reloads the values of the checkpointed live-out registers from NVM using the mapping in order to ensure crash consistency.

Forward Progress and I/O Functions: To guarantee forward progress without stagnation [14, 15, 82], SweepCache leverages the EH model [82] for estimating the worst-case energy of a region execution and its recovery. That is, among the regions, the compiler checks if some are too long to be executed with the underlying capacitor energy and splits such a long region so that it can be finished across power failure. Finally, supporting non-recoverable operations such as I/O operations has still remained an open problem. However, since SweepCache places region boundaries at all callsites, the function that implements I/O operations is treated as a separate region. Then, SweepCache can leverage the techniques in prior work [15] to guarantee I/O operations always start with a fully charged capacitor. Thus, I/O operations can always successfully complete without power failure.

4.2 Recovery Protocol

To ensure the correct recovery, SweepCache leverages the persistence status of persist buffer to determine appropriate protocols. To manage the persistence status, SweepCache introduces two additional bits, namely phase1Complete and phase2Complete, for each buffer. These bits indicate whether the corresponding phase is complete, with their initial values set to 0. When a phase is completed, the corresponding bit is set to 1. These bits are stored in a single persistent register that exists in a memory controller and gets read/written by a similar controller logic to that of prior work [101]. At runtime, depending on the power failure points, SweepCache has different persistence statuses corresponding to a different status of the phaseComplete bits: (0, 0), (1, 0), and (1, 1).

The first case is (0, 0), meaning a power outage occurs before the *s-phase1*. In such a case, buffer persistence is not complete and the NVM is not affected at all. After power comes back, SweepCache ignores the contents of the buffer and restores the saved registers including the PC register, and jumps to the PC. Note that the PC here points to the beginning of the current region. This PC was preserved at the end of the preceding region. The PC saved at the current region that points to the start of the subsequent region has not yet been written to the NVM.

For the second case, *i.e.,* (1, 0), indicating that *s-phase1* is complete but *s-phase2* is not. In such a case, since the first phase is complete, all the updated data are already in the buffer. Because the buffer is non-volatile, all the data including the saved register values still remain in the buffer during the power outage. Thus, SweepCache does not need to roll back to the beginning of the current region. Instead, SweepCache re-executes the second phase. After that, it restores the saved register values and jumps to the PC. Here, the PC points to the beginning of the next region. Let us explain here why SweepCache needs to flush dirty cachelines at each region end. Upon power failure, the cache loses all the data including the dirty cachelines. However, during regions, only evicted dirty cachelines are written back. Thus, jumping to the next region to do the recovery without considering those non-evicted dirty lines (which may include register checkpoint stores) cannot realize correct recovery. Therefore, SweepCache needs to flush all the dirty cachelines at each region boundary. Note that the flushed data still remain in the cache with their dirty bits reset to 0.

For the third case, *i.e.,* (1, 1), both the two phases are complete. In such a case, the recovery is simple. SweepCache just restores the saved register values in NVM and jumps to the PC that points to the beginning of the region interrupted by the power outage.

4.3 Write-After-Write

To ensure region-level persistence, SweepCache must be careful about write-after-write (WAW) cases. Since SweepCache utilizes region-level parallelism to hide the persistence latency, a dirty cacheline of the prior region may be overwritten by the current region's stores before being written back to the persist buffer. To avoid this problem, SweepCache leverages the phase1Complete bit, as mentioned in Section 4.2, in conjunction with the cache dirty bits to indicate whether a specific cacheline resides in the *s-phase1* of the preceding region. To be more specific, if the prior region's phase1Complete is 0 and the dirty bit is 1, meaning that the cacheline is awaiting flush, the store of the current region that tries to write to the same address of the dirty cacheline needs to wait until the phase1Complete bit becomes 1. Such a method can sometimes cause unnecessary waiting. For example, the previous region's phase1Complete bit is 0 but the dirty cacheline is caused by the current region, we still wait even if this WAW does not cause any persistence issue. However, such a case is very rare in our evaluations so it is acceptable for such rare unnecessary waiting.

4.4 Cache Misses Handling

In cases where a cache miss occurs, SweepCache first checks the persist buffer before accessing NVM, as the most recent data may still be present in the buffer. To search for the requested data in the buffers, CAM (content addressable memory) would be the fastest technology, but too expensive for energy harvesting systems. Sequential search is energy-efficient, but too slow since two buffers may need to be searched if the previous region has not completed its *s-phase2*. Therefore, SweepCache requires a cost-effective and high-performance search method for handling cache misses.

Since SweepCache only needs to consult the buffers at the cache miss cases, the cache miss rate determines the frequency of consulting the buffers. However, our evaluation shows that the average cache miss rate is only 3.43% when the cache size is 4kB, meaning that it is affordable for SweepCache to leverage a sequential search logic rather than the expensive CAM-based associative search.

Though the sequential search is affordable because of the low cache miss rate, it is still slow because it may incur double NVM accesses for cache miss cases. Whenever a cache miss happens, SweepCache searches the persist buffers first. If the data is not found, SweepCache accesses the NVM.

However, in our evaluations, we found that the persist buffers only contain a few entries and are empty most of the time (we do not consider the flushed entries at the region boundary since they still remain in the cache and do not cause any cache misses). This is not difficult to understand. For a certain cacheline, it has to satisfy two conditions to be written back to persist buffer: (1) it is dirty; (2)

it is evicted. For the first condition, the number of dirty cachelines in each region is limited since SweepCache always flushes dirty cachelines at each region end, leaving a clean cache for the next region. For the second condition, based on the low cache miss rate, the eviction rate is also low.

Given these, SweepCache leverages a simple but effective method — deploying a single bit (referred to as empty-bit) to indicate whether the buffer is empty. Thus, for a cache miss, SweepCache only does the sequential buffer search when the bit is 0, *i.e.,* the buffer is not empty[7]. Otherwise, it bypasses the buffer. Thanks to the low fill rate of the buffer, empty-bit bypasses 99% of buffer searches in our evaluation, realizing high search performance with pretty low hardware costs (only two bits for two persist buffers).

4.5 The Size of the Persist Buffer

The size of the buffer determines the number of entries it can hold, which in turn sets the store threshold used by the compiler. This threshold then affects the region size. Longer regions are more likely to be interrupted during their *s-phase1*, which can cause slow forward progress due to our roll-back recovery property. Moreover, more stores may generate more evictions, increasing the frequency of searching buffers in cache miss cases. However, longer regions tend to hide more persistence delay, leading to higher region-level parallelism. Therefore, determining the size of the persist buffer is a trade-off to achieve optimal performance.

We experimentally found that setting the size (threshold) to 64 has relatively small average store numbers and high parallelism. The buffer entry consists of address and data where data has the same granularity as the cacheline (64B).

4.6 Write-Back-Instructive Table

SweepCache is required to flush all the dirty cachelines at the end of each region (Section 3.4) in order to guarantee the correct recovery and region persistence. To realize this, SweepCache needs to scan all the cachelines at each region boundary to determine which cachelines are dirty. However, such a scan process not only lengthens *s-phase1* but also impacts the accuracy of dirty cacheline identification (before scanning all the cachelines, the next region's stores may cause new dirty cachelines; flushing the next region's dirty cacheline may cause incorrect power failure recovery).

To eliminate such overhead and guarantee correct recovery, we leverage a small SRAM bit-table (referred to as write-back-instructive table) to indicate which cachelines are dirty at each region boundary. The table is updated during the region execution, allowing the identification of all the dirty cachelines at each region boundary by reading the table instead of scanning the entire cache. As with the persist buffer design, to prevent structure hazards between regions, SweepCache employs two tables. The table size is equal to the number of cachelines (one bit indicates one cacheline), *e.g.,* for a 4kB cache with a 64B block, a 64-bit table is enough.

5 DISCUSSION

In general, frequent power failure is a norm of energy harvesting systems, in which case SweepCache performs the best regardless of energy sources (RF, solar, thermal) as will be shown in Figure 10. However, for a system backed with a bulky supercapacitor that can sustain for a while, SweepCache might waste harvested energy for region-level persistence (Section 3.2)—saving the data being stored in regions to NVM in case they are power-interrupted. That is because the regions rarely encounter power failure owing to abundant energy piled in the supercapacitor, though it causes several issues, *i.e.,* slow reboot time, bulky area cost, and energy inefficiency due to the leakage proportional to the size of a capacitor.

While SweepCache mainly targets tiny energy harvesting systems (*e.g.,* wearables) backed with energy-efficient small capacitors with a few hundred *nF* [8], it is possible to mitigate the problem of wasting energy for supercapacitor-equipped systems. To achieve this, we aim to enlarge our region size so that the region-level persistence is conducted less frequently thereby reducing the overhead. There are a couple of ways to do that: (1) small function inlining [70] and (2) aggressive loop unrolling including its speculative optimization [35]. We leave applying them as future work.

Multi-core: To the best of our knowledge, there is no commodity multi-core energy harvesting systems. In the literature, a single in-order core is predominantly used due to the power efficiency issue, *e.g.,* RF energy harvesting cannot afford to power even dual-core systems. For this reason, we do not delve into the topic of multi-core systems on purpose.

6 EVALUATION

We implement compiler techniques described in Section 4.1 on top of the LLVM 13.0.1 [42]. All evaluated programs are compiled with the default O3 flag except for our compiler optimizations. To measure the impact of runtime libraries as well, we instruct the linker to link evaluated programs against the *MUSL* C library which is also compiled by SweepCache's compiler with our compilation optimizations enabled.

We conduct our experiments atop the gem5 [5] with ARM ISA to simulate a single-core in-order processor as the original NVP simulator [23]. As in prior work [57, 97], SweepCache only modifies the L1DCache as the volatile cache while maintaining the L1ICache as an NVM cache. In all cache-enabled designs, the default cache size is configured as 4kB with a 2-way association. The capacitor size is set to *470nF*, consistent with prior real fabricated chips [60, 87, 93]. The propagation delay of JIT-checkpoint designs is configured in line with prior work [23, 58, 87, 97]. In particular, since SweepCache does not have a backup stage thus only has the restore propagation delay. To precisely set the delay, we deliberately selected the technology [28] built in the same year as the JIT-checkpoint designs' voltage detector. By default, SweepCache's persist buffer size is set to 64. Other configurations [16, 23, 97] can be seen in Table 1. We evaluated applications with two real power traces (RFHome and RFOffice) which were collected from real RF energy harvesting systems [23].

The rest of this section compares two variants of SweepCache (with NVM Search or Empty-Bit Search, see Section 4.4) against ReplayCache and NVSRAM (only backups dirty cachelines), in terms of their speedups over the cache-free baseline NVP.

[7]Since the buffer is FIFO, every time the younger one is found first if there are multiple entries for the same cacheline.

[8]Note that such a capacitor size can still support peripherals such as LCD—that can be powered by even a *100nF* capacitor [29], UV sensor, and NFC transceiver [60].

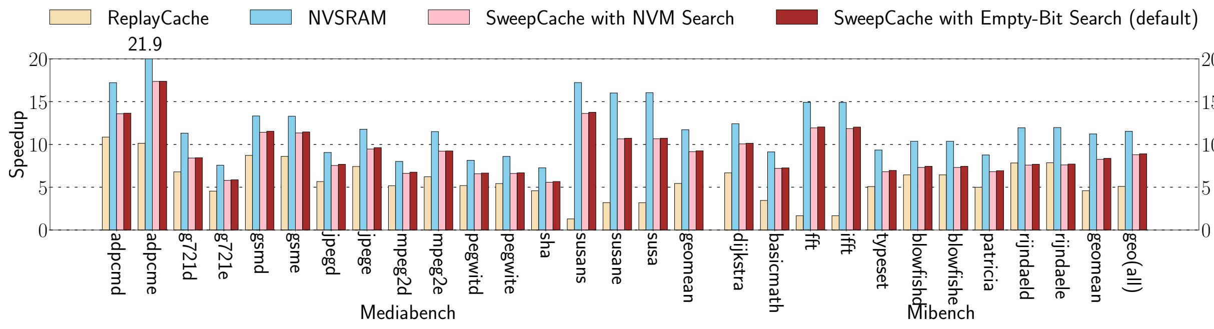


Figure 5: Speedups over NVP without power failure

Table 1: Simulation Configuration

	NVP	ReplayCache	NVSRAM	SweepCache
V_{max}/V_{min}	3.5/2.8	3.5/2.8	3.5/2.8	3.5/2.8[1]
Backup/Restore	2.9/3.2	2.9/3.2	3.2/3.4	No/3.3
Cache size	N/A	4KB	4KB	4KB
capacitor size	470nF	470nF	470nF	470nF
NVM size	16MB	16MB	16MB	16MB
NVM tech	ReRAM	ReRAM	ReRAM	ReRAM
Write/Read (latency)	120ns/20ns	120ns/20ns	120ns/20ns	120ns/20ns
Propagation delay[2]	1.5/10.3us	1.5/10.3us	1.5/10.3us	No/1.1us

6.1 Performance without Power Outage

To analyze the performance of SweepCache, we first evaluate it without a power outage. Figure 5 shows such outage-free performance results. On average, NVM search and Empty-Bit search achieve 8.80x and 8.91x speedups, respectively, while ReplayCache exhibits a speedup of 5.10x. We find that the ReplayCache's speedup over the NVP is lower than what the original paper reports. That is because we compile all the libraries for both SweepCache and ReplayCache in addition to the application code, which would otherwise lead to similar speedups. NVSRAM performs the best with a speedup of 11.53x. There is a two-fold reason for the performance gap between the NVSRAM and SweepCache: (1) NVSRAM has fewer instructions since it does not generate checkpoint stores; and (2) the persistence latency of SweepCache may not be fully hidden by its region-level parallelism as shown in Section 6.3.

For most of the applications, SweepCache performs better than ReplayCache. On average, NVM Search and Empty-Bit Search achieve speedups of 1.73x and 1.75x over the ReplayCache, respectively. The performance gain is mainly because of: (1) the instructions generated by SweepCache are fewer than the ReplayCache (Section 6.5). Since ReplayCache's compiler has to generate store-fence instructions and *clwb* instructions for every store to guarantee persistence while SweepCache only generates checkpoint stores for live-out registers; (2) SweepCache has a high parallelism efficiency, as demonstrated in Section 6.3, which can overlap most of the persistence latency; (3) low latency paid for searching persist buffers since the cache miss rate is low and the average number of filled entries in the persist buffer is very small (0.00012 per region).

We notice that there are two exceptions, *i.e., rijndaeldec* and *rijndaelenc*, where SweepCache is not better than the ReplayCache. These two programs are small while SweepCache generates around 2x more regions than the ReplayCache. For each region, SweepCache needs to complete the two-phase persistence. Most of the time, the persistence latency can be hidden by SweepCache's region-level parallelism, but the non-overlapped part still plays a non-trivial role in such small programs' execution time.

Compared with NVM Search, Empty-Bit Search obtains a speedup of 1.18%. As mentioned before, the cache miss rate is very low leading to only an average of 0.00012 per-region access to the persist buffers. Thus, even though the Empty-Bit Search can bypass over 99% buffer search, based on the rare accesses to the persist buffers, the performance gain is limited.

6.2 Performance with Power Outages

Figure 6 and Figure 7 show the performance results in power outage cases with RFHome and RFOffice power traces. For RFHome trace, NVM Search and Empty-Bit Search achieve average speedups of 14.60x and 14.86x while ReplayCache and NVSRAM exhibit speedups of 4.26x and 7.37x. Compared with ReplayCache, NVM Search and Empty-Bit Search attain average speedups of 3.43x and 3.49x, respectively. In contrast to NVM search, Empty-Bit obtains a speedup of 1.75% on average. For RFOffice trace, NVM search and Empty-Bit achieve average speedups of 14.31x and 14.60x while ReplayCache and NVSRAM do speedups of 4.20x and 7.32x. Compared with ReplayCache, NVM Search and Empty-Bit Search deliver average speedups of 3.41x and 3.47x, respectively, while Empty-Bit Search yields a speedup of 1.96% over the NVM Search.

For the power outage cases, it is impressive that SweepCache achieves much better performance than JIT-checkpoint designs, *i.e.,* ReplayCache and NVSRAM, which highlights the huge benefits of our JIT-checkpoint-free property. Compared with JIT-checkpoint designs, SweepCache maintains higher energy efficiency as shown in Section 6.6. That is because SweepCache does not need to pay any hard-won energy for the JIT-backup or other necessary logic such as the NVFF, backup/restore controller, etc. Therefore, SweepCache allows a larger portion of harvested energy to be used for computation. In this way, SweepCache experiences less power failure

[1]Owing to our simpler logic as described in Section 2.2, SweepCache actually can afford a lower V_{min} as prior work [16] uses 1.8v V_{min}; according to our evaluation, SweepCache can obtain extra 10% ~ 15% performance gain with 1.8v V_{min}. While using the same V_{min} (*i.e.,* 2.8) of JIT-checkpoint designs serves as the lower-bound performance of SweepCache, it still outperforms them significantly.
[2]Propagation delay: backup/restore voltage detection delay

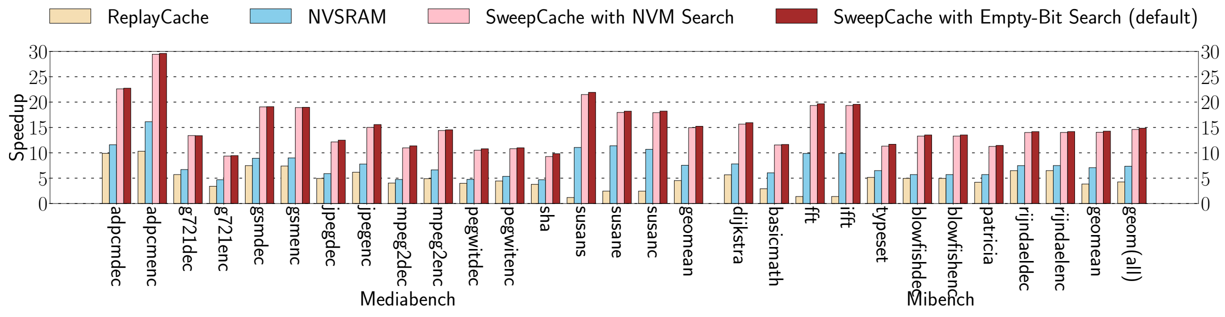


Figure 6: Speedups over NVP for RFHome trace

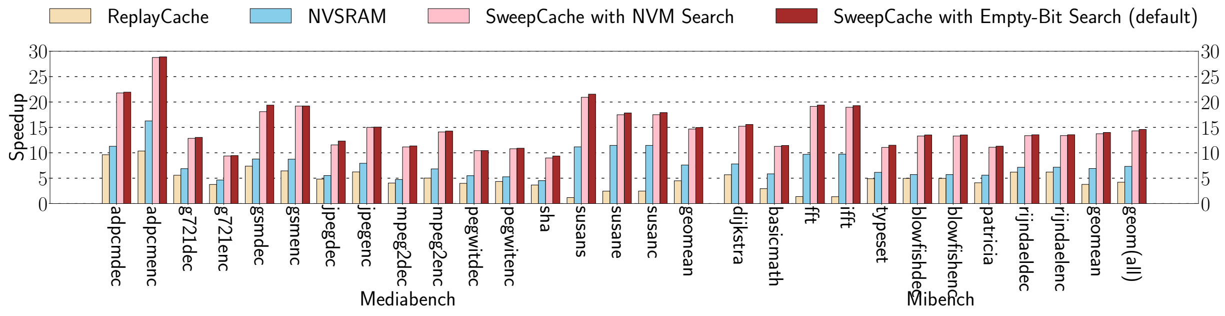


Figure 7: Speedups over NVP for RFOffice trace

(Table 2) which saves a significant amount of charging time to reboot the system. Moreover, SweepCache's region-level parallelism greatly reduces the persistence delay while JIT-checkpoint designs cannot. Besides, the low-cost design of SweepCache also makes it possible to have much less propagation delay leading to faster restoration. In addition to the above reasons, the three others mentioned in Section 6.1 still contribute to SweepCache's significant speedup over ReplayCache for these RFOffice and RFHome traces.

In particular, the speedup of Empty-Bit Search over NVM Search under the power traces is slightly higher than that under the power-outage-free case. That is because the higher cache miss rates caused by frequent outages lead to more accesses for the persist buffer, which highlights the role of empty-bit. Taking into account the superior performance of Empty-Bit Search, we choose the Empty-Bit as the default design of SweepCache for the following evaluation.

6.3 Region-Level Parallelism Efficiency

SweepCache exploits region-level parallelism to hide the persistence latency. This is one of the reasons for its outstanding performance. Thus, we evaluate the parallelism efficiency of SweepCache in the power-outage-free/power-outage cases. We use the following formula to calculate region-level parallelism efficiency.

$$Parallelism_{eff}\% = ((T_p - T_{wait})/(T_p)) * 100 \quad (1)$$

T_p is the *Persistence Latency* (without parallelism) and T_{wait} is the actual waiting time. Higher efficiency means more persistence latency can be hidden. Overall, we can achieve an average parallelism efficiency of 91.70% for power-outage-free scenarios and 91.95% for power-outage cases.

6.4 Sensitivity Study

Cache size: Figure 8 shows the speedups of SweepCache and its competing schemes across different cache sizes from 512B to 16KB with the RFOffice power trace. The performance is basically in proportion to the cache size. Compared with the NVP, the greater the cache size is, the higher speedup SweepCache can achieve.

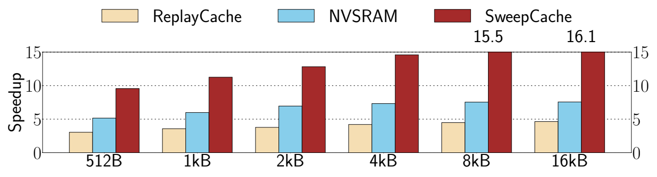


Figure 8: Speedups over NVP across different cache sizes

Capacitor size: We also explore the impact of the capacitor size on the performance. Figure 9 shows two kinds of speedups with different baselines for RFOffice power trace: (1) bars represent speedups over NVP varying its capacitor size from *100nF* to *1mF* as with other schemes; (2) the line shows another speedup over NVP whose capacitor size is fixed to *100nF* and how they vary as the capacitor of other schemes gets bigger.

In addition, we show the number of average power outages across different capacitor sizes in Table 2 without considering the initial power-off state. Overall, increasing the capacitor size leads to better performance. However, performance gains become limited once the capacitor size reaches *10uF*, as all designs experience fewer power outages with a capacitor size of *10uF* or greater. Both

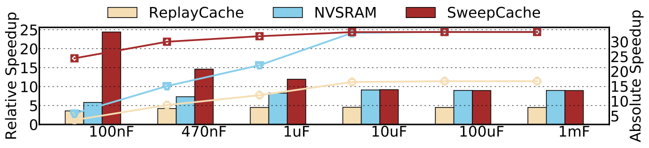


Figure 9: Speedups over NVP across different capacitors with and without fixing the size of NVP's capacitor

Table 2: The Number of Average Power Outages

Capacitor size	NVP	ReplayCache	NVSRAM	SweepCache
100nF	458.26	151.19	77.65	34.11
470nF	144.65	38.77	20.23	14.42
1uF	50.85	18.31	9.50	6.23
10uF	7.23	1.46	0.62	0.35
100uF	0.46	0.04	0.00	0.00
1mF	0.00	0.00	0.00	0.00

NVSRAM and SweepCache exhibit no power failure when the capacitor size comes to *100uF*, but NVSRAM is only slightly better than SweepCache while NVSRAM clearly outperforms ours in Figure 5. That is because NVSRAM must pay a longer propagation delay than ours when transitioning from the initial power-off state to the power-on state.

Our evaluation yields three key insights: Firstly, for a given capacitor size, SweepCache always experiences fewer power outages than JIT-checkpoint designs due to its superior energy efficiency. Secondly, given a certain capacitor size with power failure, SweepCache consistently outperforms JIT-checkpoint designs. Finally, a larger capacitor does not necessarily translate to better performance. For example, a $1uF$ SweepCache can deliver comparable performance to a $10uF$ NVSRAM, while the latter incurs 1.43x area costs with only marginal performance improvements.

Power Traces: Figure 10 shows the performance with different power traces. In general, JIT-checkpoint designs exhibit higher speedups over NVP when operating with more stable power traces such as solar and thermal, as opposed to RF traces. Conversely, SweepCache tends to demonstrate higher speedups when dealing with RF traces. That is because JIT-checkpoint designs generate fewer checkpoints when exposed to more stable power traces, whereas SweepCache conducts checkpoints at each region no matter the power failure frequency. But overall, SweepCache still delivers the best performance among all the different traces owing to its higher energy efficiency.

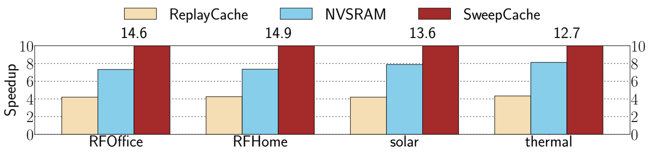
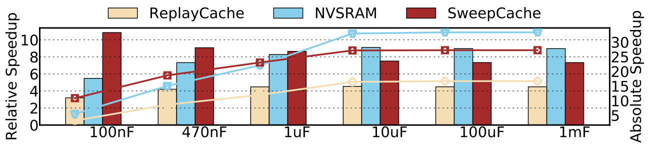


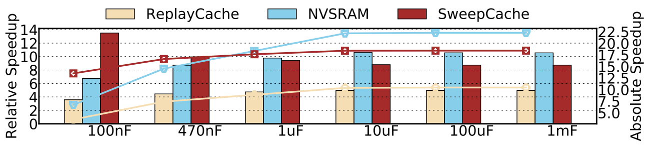
Figure 10: Speedups over NVP across different power traces

Propagation delay: We also analyze the impact of propagation delay and conduct tests in two distinct settings: (1) set our delay the same as JIT-checkpoint designs (set SweepCache's T_{plh} to $10.3us$); (2) set the delay of JIT-checkpoint designs to the shortest shown in the paper (set JIT-checkpoint designs' T_{plh} to $3.0us$ and their T_{phl} to $0.5us$) [87]. Figure 11 shows the results. We find that propagation delay plays a non-trivial role in the execution time since it is related to the speed of backup and restoration. Whether extending the delay in SweepCache (*i.e.,* setting 1) or reducing the delay in JIT-checkpoint designs (*i.e.,* setting 2), both settings result in an earlier occurrence of the performance turning point—when NVSRAM outperforms SweepCache—compared to our default settings. This is because these two settings either slow down our restoration or expedite the backup and restoration of JIT-checkpoint designs. However, it is hard to shrink JIT-checkpoint designs' delay to so short since it needs huge power consumption [87] and we believe SweepCache's propagation delay can also be shrunk much shorter with the same power consumption.

Store threshold: Recall that the threshold is the maximum number of stores in a region, which does not mean that each region should have as many stores as the threshold. We evaluated the average store counts of regions at run time with different thresholds,

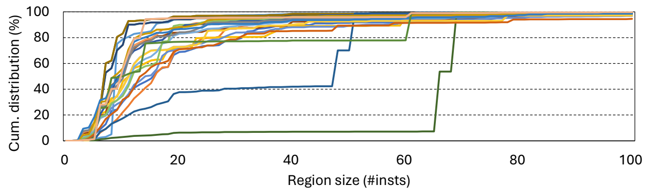


(a) Speedups over NVP across different capacitors when SweepCache's propagation delay is set the same as that of JIT-checkpoint designs

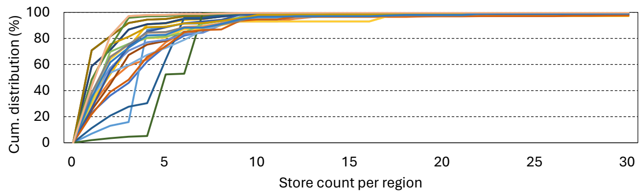


(b) Speedups over NVP across different capacitors when JIT-checkpoint designs's propagation delay is significantly reduced

Figure 11: Normalized speedup compared to NVP with different propagation delay settings.



(a) Region size (Each color indicates a different benchmark).



(b) Number of stores per region

Figure 12: Analysis on region size and store count per region

such as 32, 64, 128, and 256. The resulting counts are relatively small showing insignificant differences across the thresholds. This phenomenon results from the initial region boundaries inserted at the entry and exit points of each function call and at each loop header in that they cannot be optimized to extend the region size (Section 4.1). For example, *region combining* cannot merge such callsite boundaries while *loop unrolling* is not feasible for those whose iteration counts are not known at compile time. Nonetheless, SweepCache can address the problems by leveraging the techniques mentioned in Section 5, *i.e.,* aggressive function inlining [70] and speculative loop unrolling [35]. Figure 12 shows CDF results of (1) region size and (2) store count per region for all benchmarks tested with the default threshold, *i.e.,* 64; the average store number is 3.92 (while the average region size is 19.47) which to some extent accounts for why the persist buffers are empty most of the time.

6.5 Instruction Counts

Overall, compared with SweepCache, ReplayCache generates 1.64x as many instructions, which mainly comes from extra *clwb* instructions and store fence instructions. This ratio is much greater than only compiling the programs, which is only 1.03x. Compared with NVSRAM, SweepCache generates 15.04% more instructions.

6.6 Energy Consumption

To figure out the energy efficiency of SweepCache, we evaluate the total energy consumption under the default setting by using the power model provided by NVPSim [23] with RFOffice trace. Compared with NVP, the normalized total energy consumption for ReplayCache, NVSRAM, and SweepCache are 20.86%, 12.37%, and 10.21%, respectively. Figure 13 also shows their backup/restore energy consumption breakdowns—normalized to NVP's—which are 23.74%, 15.42%, and 0.28%, respectively. SweepCache turns out to be the most energy-effective.

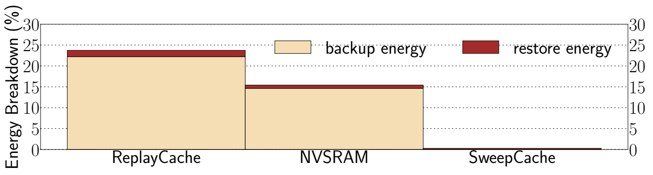


Figure 13: Breakdown of backup and restore energy consumptions normalized to those of NVP

6.7 SweepCache vs. NvMR

This section compares SweepCache with NvMR, the state-of-the-art work that performs memory renaming to eliminate write-after-read (WAR) dependences [4], *i.e.,* the reason for idempotence violations [26]. Once they are detected, NvMR attempts to rename their memory location to be written. If this is not possible due to structure hazards in the architectural components of NvMR, it triggers a backup to persist the registers, dirty cachelines, and other volatile states necessary for the memory renaming. Also, NvMR takes advantage of JIT checkpointing techniques that in their original form, do not start a power-interrupted program until the restoration voltage ($V_{restore}$, see Section 2.2) is reached, which could otherwise encounter WAR dependences. The beauty of NvMR is that it enables the program to keep running even after the JIT backup, without waiting for the capacitor to be charged enough to offer $V_{restore}$—because the memory renaming can resolve the WAR dependences. In particular, if power failure happens while the backup voltage is not secured, NvMR must roll back to the latest JIT backup point.

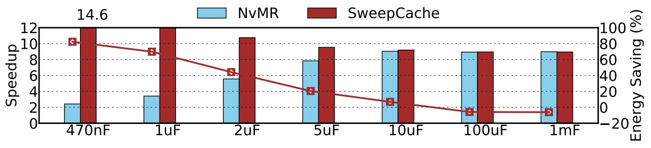


Figure 14: Performance gain and energy saving over NvMR.

We implemented NvMR with its parameters kept the same as SweepCache's memory hierarchy shown in Table 1. Figure 14 shows the results of SweepCache and NvMR (a bar corresponds to their speedup over NVP while a curve to SweepCache's energy reduction compared to NvMR) when the RFOffice trace is used. SweepCache is significantly faster than NvMR for all capacitor settings but *1mF* that is a lot bigger than our target capacitor size, *i.e.,* a few hundred *nF* (Section 5). Overall, across 7 different capacitor sizes, SweepCache achieves an average of 1.71x speedup (up to 6.04x when the capacitor size is *470nF*) over NvMR mainly due to its superior energy efficiency—resulting from the JIT-checkpoint-free nature and the lightweight hardware design. On average, SweepCache saves 19.94% of the energy consumed by NvMR (up to 82.3% with the *470nF* capacitor).

6.8 Cache Miss Rate and Write Amplification

Compared to NVSRAM, SweepCache may encounter more cold misses since it does not save any cachelines before power failure. To evaluate the cache miss rate, we consider NVSRAM, NVSRAM-E (backs up the entire cache), SweepCache, and ReplayCache. The results are shown in Figure 15. Overall, the cache miss rate for all designs (excluding NVSRAM-E) decreases as power traces become more stable. We notice that ReplayCache has a higher miss rate than SweepCache, even though both designs do not save cachelines before a power outage. This is because ReplayCache experiences more power failure with the same capacitor size. Compared to NVSRAM, SweepCache incurs only a 7.50% increase in average cache miss rates as it experiences fewer power outages due to its higher energy efficiency.

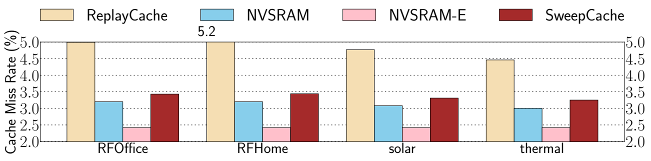


Figure 15: Cache miss rate for different traces.

Furthermore, SweepCache suffers from write amplification due to the presence of the persist buffers, resulting in twice NVM writes for every writeback. The situation worsens for ReplayCache, which generates an NVM write (*clwb*) for every store. However, NVSRAM also results in additional NVM writes for backing up the registers to NVFF and dirty cachelines to their NVM counterpart. Therefore, we count the number of NVM writes for the four designs mentioned above, as shown in Figure 16. The NVM writes of SweepCache mainly come from the writebacks at each region end, while the NVM writes of ReplayCache mainly come from its *clwb* for every store. Unlike NVSRAM and NVSRAM-E, whose NVM writes primarily stem from the backup preceding the power failure and are largely impacted by the power outage frequency, resulting in fewer NVM writes in more stable power traces, *i.e.,* solar and thermal, the NVM writes of SweepCache and ReplayCache mainly come from regular persistence operations which does not show a significant difference among various power traces. On average, SweepCache incurs 4.62x as many NVM writes compared to NVSRAM. However, NVM writes only consume about 0.01% and 0.23% of the total energy for NVSRAM and SweepCache, respectively. Therefore, thanks to SweepCache's superior energy efficiency and its substantial region-level parallelism, which effectively conceals the majority of NVM write latency, SweepCache still delivers better performance.

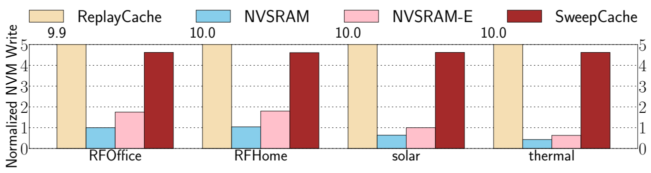


Figure 16: Analysis on the number of NVM writes normalized to those of NVSRAM when RFOffice trace is used

6.9 Hardware Costs

The hardware costs of SweepCache are not significant owing to its intelligent compiler-architecture co-design. Apart from the two persist buffers, SweepCache only needs a total of 134 bits, *i.e.,* two empty-bits (one of each persist buffer), four phaseComplete bits, and two 64-bit SRAM tables, for a 4kB cache. This is rather minimal compared to the hardware costs of prior JIT-checkpoint designs.

7 RELATED WORK

There is a large body of prior research on energy harvesting systems. Apart from the NVP architecture, QuickRecall [31] is another choice to deal with frequent power failure in a crash-consistent way. Although QuickRecall obviates the need for nonvolatile flip-flops by saving registers to NVM before impending power failure, it still relies on JIT checkpointing to ensure the failure atomicity of the register saving. Since QuickRecall should save the registers by executing a series of store instructions without special hardware support, the resulting performance overhead is quite significant compared to those of NVP approaches.

Unlike QuickRecall, some prior work attempts to equip energy harvesting systems with a data cache. NVCache explores the use of NVM as the material of persistent cache implementation [1, 36, 67, 68, 76, 92, 95]. However, NVCache causes both longer latency and more energy consumption than a traditional SRAM cache. Thus, other researchers delve into the integration of SRAM on top of NVM, *i.e.,* leveraging the NVM as JIT-checkpoint storage to save SRAM cache contents and consult it across power failure [12, 25, 43, 48, 49, 65, 66, 69, 83, 86, 94, 96]. To enhance the performance of the NVM backup and the restoration, the researchers propose various NVM technologies. For example, STT-RAM provides faster access time and higher energy efficiency than alternative NVM technologies at the cost of more sensitive process, voltage, and temperature (PVT) variations [20], causing more errors with a higher possibility. Furthermore, the speeds of the NVM backup and the restoration are lacking and remain a daunting challenge, as no NVM technology currently affords to match the performance of SRAM [16, 41, 97].

One might ask if such a cache-enabled energy harvesting system can benefit from existing crash consistency mechanisms for high-performance computing systems backed with deep cache hierarchy and persistent memory. For instance, prior software-based recovery schemes based on undo/redo logging or idempotent processing [9, 32, 33, 35, 40, 50, 51, 53–56, 90, 99, 100] may seem like a potential avenue. However, they tend to incur significant performance degradation, *e.g.,* iDO [51] for failure-atomic sections (FASEs) and Mnemosyne [90] for transactions result in up to 2-3x slowdown because of so-called persist barriers that serialize the out-of-order pipeline execution significantly.

Taking that into consideration, some other recovery schemes come up with hardware-based logging [19, 22, 34, 38, 73, 84] to lower the performance overhead. However, they still suffer significant pipeline stalls waiting at the end of each atomic region *e.g.,* a transaction or a FASE, to ensure the persistence of the stores in the region. Furthermore, neither the software nor the hardware logging is devised for whole system persistence [41, 72, 98] that is essential for energy harvesting systems. That is, these prior recovery schemes offer crash consistency only to a region of the code in transactions or failure-atomic sections, leaving other code outside the region inconsistent across power failure. Consequently, unlike SweepCache, the prior schemes are not appropriate for an energy harvesting system, *i.e.,* they cannot enable it to take advantage of a data cache in a performant and low-cost manner.

8 CONCLUSION

This paper presents SweepCache, a novel compiler and architecture co-design approach that enables energy harvesting systems to exploit a volatile cache in a performant and lightweight way. To ensure correct power failure recovery, the compiler generates recoverable regions while the architecture runs them in a failure-atomic way. Thanks to SweepCache's region-level persistence that cleans up the cache across the region boundary, energy harvesting systems do not have to rely on expensive just-in-time (JIT) checkpointing, and thus they can fully utilize harvested energy for computation. As a result, SweepCache achieves 3.47x and 3.49x speedups over the state-of-art work for two representative energy harvesting traces, respectively.

ACKNOWLEDGMENTS

We thank anonymous reviewers and our shepherd for their valuable comments. At Purdue, this work was supported by NSF grants 2001124 (CAREER), 2153749, and 2314681. At UCF, this work was supported by NSF grant 2314680.

REFERENCES

[1] Sukarn Agarwal and Hemangee K Kapoor. 2019. Improving the lifetime of non-volatile cache by write restriction. *IEEE Trans. Comput.* 68, 9 (2019), 1297–1312.

[2] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. 2007. *Compilers: principles, techniques, & tools.* Pearson Education India.

[3] Domenico Balsamo, Alex S Weddell, Geoff V Merrett, Bashir M Al-Hashimi, Davide Brunelli, and Luca Benini. 2014. Hibernus: Sustaining computation during intermittent supply for energy-harvesting systems. *IEEE Embedded Systems Letters* 7, 1 (2014), 15–18.

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

[5] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH computer architecture news* 39, 2 (2011), 1–7.

[6] Jo Bito, Ryan Bahr, Jimmy G Hester, Syed Abdullah Nauroze, Apostolos Georgiadis, and Manos M Tentzeris. 2017. A novel solar and electromagnetic energy harvesting system with a 3-D printed package for energy efficient Internet-of-Things wireless sensors. *IEEE Transactions on Microwave Theory and Techniques* 65, 5 (2017), 1831–1842.

[7] Paul Cahill, Rosemary O'Keeffe, Nathan Jackson, Alan Mathewson, and Vikram Pakrashi. 2014. Structural health monitoring of reinforced concrete beam using piezoelectric energy harvesting system. In *EWSHM-7th European workshop on structural health monitoring.*

[8] Shihua Cao and Jianqing Li. 2017. A survey on ambient energy sources and harvesting methods for structural health monitoring applications. *Advances in Mechanical Engineering* 9, 4 (2017), 1687814017696210.

[9] Dhruva R Chakrabarti, Hans-J Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging locks for non-volatile memory consistency. *ACM SIGPLAN Notices* 49, 10 (2014), 433–452.

[10] Qijia Cheng, Zhuoteng Peng, Jie Lin, Shanshan Li, and Fei Wang. 2015. Energy harvesting from human motion for wearable devices. In *10th IEEE International Conference on Nano/Micro Engineered and Molecular Systems.* IEEE, 409–412.

[11] Pi-Feng Chiu, Meng-Fan Chang, Shyh-Shyuan Sheu, Ku-Feng Lin, Pei-Chia Chiang, Che-Wei Wu, Wen-Pin Lin, Chih-He Lin, Ching-Chih Hsu, Frederick T Chen, et al. 2010. A low store energy, low VDDmin, nonvolatile 8T2R SRAM with 3D stacked RRAM devices for low power mobile applications. In *2010 Symposium on VLSI Circuits.* IEEE, 229–230.

[12] Pi-Feng Chiu, Meng-Fan Chang, Che-Wei Wu, Ching-Hao Chuang, Shyh-Shyuan Sheu, Yu-Sheng Chen, and Ming-Jinn Tsai. 2012. Low store energy, low VDDmin, 8T2R nonvolatile latch and SRAM with vertical-stacked resistive memory (memristor) devices for low power mobile applications. *IEEE Journal of Solid-State Circuits* 47, 6 (2012), 1483–1496.

[13] Jongouk Choi, Hyunwoo Joe, and Changhee Jung. 2022. CapOS: Capacitor Error Resilience for Energy Harvesting Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 11 (2022), 4539–4550.

[14] Jongouk Choi, Hyunwoo Joe, Yongjoo Kim, and Changhee Jung. 2019. Achieving Stagnation-Free Intermittent Computation with Boundary-Free Adaptive Execution. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS).* 331–344. https://doi.org/10.1109/RTAS.2019.00035

[15] Jongouk Choi, Larry Kittinger, Qingrui Liu, and Changhee Jung. 2022. Compiler-directed high-performance intermittent computation with power failure immunity. In *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS).* IEEE, 40–54.

[16] Jongouk Choi, Qingrui Liu, and Changhee Jung. 2019. CoSpec: Compiler directed speculative intermittent computation. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture.* 399–412.

[17] Jongouk Choi, Jianping Zeng, Dongyoon Lee, Changwoo Min, and Changhee Jung. 2023. Write-Light Cache for Energy Harvesting Systems. In *Proceedings of the 50th Annual International Symposium on Computer Architecture.* 1–13.

[18] Yung-Wey Chong, Widad Ismail, Kwangman Ko, and Chen-Yi Lee. 2019. Energy harvesting for wearable devices: A review. *IEEE Sensors Journal* 19, 20 (2019), 9047–9062.

[19] Kshitij Doshi, Ellis Giles, and Peter Varman. 2016. Atomic persistence for SCM with a non-intrusive backend controller. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA).* IEEE, 77–89.

[20] Moritz Fieback, Christopher Münch, Anteneh Gebregiorgis, Guilherme Cardoso Medeiros, Mottaqiallah Taouil, Said Hamdioui, and Mehdi Tahoori. 2022. PVT Analysis for RRAM and STT-MRAM-based Logic Computation-in-Memory. In *2022 IEEE European Test Symposium (ETS).* IEEE, 1–6.

[21] Tzeno Galchev, James McCullagh, Rebecca L Peterson, and Khalil Najafi. 2010. A vibration harvesting system for bridge health monitoring applications. *Proc. PowerMEMS* 1 (2010), 179–182.

[22] Ellis Giles, Kshitij Doshi, and Peter Varman. 2013. Bridging the programming gap between persistent and volatile memory using WrAP. In *Proceedings of the ACM International Conference on Computing Frontiers.* 1–10.

[23] Yizi Gu, Yongpan Liu, Yiqun Wang, Hehe Li, and Huazhong Yang. 2016. NVPsim: A simulator for architecture explorations of nonvolatile processors. In *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC).* IEEE, 147–152.

[24] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the fourth annual IEEE international workshop on workload characterization. WWC-4 (Cat. No. 01EX538).* IEEE, 3–14.

[25] Christian E Herdt and CA Paz de Araujo. 1992. Analysis, measurement, and simulation of dynamic write inhibit in an nvSRAM cell. *IEEE transactions on electron devices* 39, 5 (1992), 1191–1196.

[26] Matthew Hicks. 2017. Clank: Architectural support for intermittent computation. *ACM SIGARCH Computer Architecture News* 45, 2 (2017), 228–240.

[27] Shao-Yu Huang, Jianping Zeng, Xuanliang Deng, Sen Wang, Ashrarul Haq Sifat, Burhanuddin Bharmal, Jiabin Huang, Ryan Williams, Haibo Zeng, and Changhee Jung. 2023. RTailor: Parameterizing Soft Error Resilience for Mixed-Criticality Real-Time Systems. In *2023 IEEE Real-Time Systems Symposium (RTSS).* IEEE.

[28] Texas Instruments. 2015. LMV7275-Q1 Automotive Single 1.8-V Low Power Comparator With Rail-to-Rail Input. https://www.ti.com/lit/ds/symlink/lmv7291.pdf. Accessed: 2023-02-16.

[29] Texas Instruments. 2015. MSP low-power microcontroller. https://e2e.ti.com/support/microcontrollers/msp-low-power-microcontrollers-group/msp430/f/msp-low-power-microcontroller-forum/454244/what-size-capacitor-is-needed-across-the-charge-pump-lcdcapx-pins-for-the-fr4133-lcd_e-module. Accessed: 2023-03-16.

[30] Texas Instruments. 2016. MSP430FR5994 LaunchPad Development Kit. https://www.ti.com/lit/ug/slau678c/slau678c.pdf. Accessed: 2023-02-16.

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

[32] Jungi Jeong, Jaewan Hong, Seungryoul Maeng, Changhee Jung, and Youngjin Kwon. 2020. Unbounded hardware transactional memory for a hybrid DRAM/NVM memory system. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).* IEEE, 525–538.

[33] Jungi Jeong and Changhee Jung. 2021. 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.* 517–529.

[34] Jungi Jeong, Chang Hyun Park, Jaehyuk Huh, and Seungryoul Maeng. 2018. Efficient hardware-assisted logging with asynchronous and direct-update for persistent memory. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).* IEEE, 520–532.

[35] Jungi Jeong, Jianping Zeng, and Changhee Jung. 2022. Capri: Compiler and architecture support for whole-system persistence. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing.* 71–83.

[36] Mohammad Reza Jokar, Mohammad Arjomand, and Hamid Sarbazi-Azad. 2015. Sequoia: A high-endurance NVM-based cache architecture. *IEEE transactions on very large scale Integration (VLSI) systems* 24, 3 (2015), 954–967.

[37] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. 2015. Efficient persist barriers for multicores. In *Proceedings of the 48th International Symposium on Microarchitecture.* 660–671.

[38] Arpit Joshi, Vijay Nagarajan, Stratis Viglas, and Marcelo Cintra. 2017. Atom: Atomic durability in non-volatile memory through hardware logging. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA).* IEEE, 361–372.

[39] Pouya Kamalinejad, Chinmaya Mahapatra, Zhengguo Sheng, Shahriar Mirabbasi, Victor CM Leung, and Yong Liang Guan. 2015. Wireless energy harvesting for the Internet of Things. *IEEE Communications Magazine* 53, 6 (2015), 102–108.

[40] Hongjune Kim, Jianping Zeng, Qingrui Liu, Mohammad Abdel-Majeed, Jaejin Lee, and Changhee Jung. 2020. 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.* 989–1004.

[41] Aasheesh Kolli. 2017. *Architecting persistent memory systems.* Ph.D. Dissertation.

[42] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 75–86.

[43] Albert Lee, Meng-Fan Chang, Chien-Chen Lin, Chien-Fu Chen, Mon-Shu Ho, Chia-Chen Kuo, Pei-Ling Tseng, Shyh-Shyuan Sheu, and Tzu-Kun Ku. 2015. RRAM-based 7T1R nonvolatile SRAM with 2x reduction in store energy and 94x reduction in restore energy for frequent-off instant-on applications. In *2015 Symposium on VLSI Circuits (VLSI Circuits).* IEEE, C76–C77.

[44] Albert Lee, Chieh-Pu Lo, Chien-Chen Lin, Wei-Hao Chen, Kuo-Hsiang Hsu, Zhibo Wang, Fang Su, Zhe Yuan, Qi Wei, Ya-Chin King, et al. 2017. A ReRAM-based nonvolatile flip-flop with self-write-termination scheme for frequent-off fast-wake-up nonvolatile processors. *IEEE Journal of Solid-State Circuits* 52, 8 (2017), 2194–2207.

[45] Chunho Lee, Miodrag Potkonjak, and William H Mangione-Smith. 1997. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of 30th Annual International Symposium on Microarchitecture*. IEEE, 330–335.

[46] Vladimir Leonov. 2011. Energy harvesting for self-powered wearable devices. In *Wearable monitoring systems*. Springer, 27–49.

[47] Xueqing Li, Sumitha George, Yuhua Liang, Kaisheng Ma, Kai Ni, Ahmedullah Aziz, Sumeet Kumar Gupta, John Sampson, Meng-Fan Chang, Yongpan Liu, et al. 2018. Lowering area overheads for FeFET-based energy-efficient nonvolatile flip-flops. *IEEE Transactions on Electron Devices* 65, 6 (2018), 2670–2674.

[48] Xueqing Li, Kaisheng Ma, Sumitha George, Win-San Khwa, John Sampson, Sumeet Gupta, Yongpan Liu, Meng-Fan Chang, Suman Datta, and Vijaykrishnan Narayanan. 2017. Design of nonvolatile SRAM with ferroelectric FETs for energy-efficient backup and restore. *IEEE Transactions on Electron Devices* 64, 7 (2017), 3037–3040.

[49] Chao Liu, Jianguo Yang, Pengfei Jiang, Qiao Wang, Donglin Zhang, Tiancheng Gong, Qingting Ding, Yuling Zhao, Qing Luo, Xiaoyong Xue, et al. 2020. A Low Power 4T2C nvSRAM With Dynamic Current Compensation Operation Scheme. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 28, 11 (2020), 2469–2473.

[50] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. 2017. DudeTM: Building durable transactions with decoupling for persistent memory. *ACM SIGPLAN Notices* 52, 4 (2017), 329–343.

[51] Qingrui Liu, Joseph Izraelevitz, Se Kwon Lee, Michael L Scott, Sam H Noh, and Changhee Jung. 2018. iDO: Compiler-directed failure atomicity for nonvolatile memory. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 258–270.

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

[53] Qingrui Liu, Changhee Jung, Dongyoon Lee, and Devesh Tiwari. 2015. Clover: Compiler directed lightweight soft error resilience. *ACM Sigplan Notices* 50, 5 (2015), 1–10.

[54] Qingrui Liu, Changhee Jung, Dongyoon Lee, and Devesh Tiwari. 2016. Compiler-directed lightweight checkpointing for fine-grained guaranteed soft error recovery. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 228–239.

[55] Qingrui Liu, Changhee Jung, Dongyoon Lee, and Devesh Tiwari. 2016. Compiler-directed soft error detection and recovery to avoid DUE and SDC via Tail-DMR. *ACM Transactions on Embedded Computing Systems (TECS)* 16, 2 (2016), 1–26.

[56] Qingrui Liu, Changhee Jung, Dongyoon Lee, and Devesh Tiwari. 2016. Low-cost soft error resilience with unified data verification and fine-grained recovery for acoustic sensor based detection. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–12.

[57] Yongpan Liu, Zewei Li, Hehe Li, Yiqun Wang, Xueqing Li, 3 usKaisheng Ma, Shuangchen Li, Meng-Fan Chang, Sampson John, Yuan Xie, et al. 2015. Ambient energy harvesting nonvolatile processors: From circuit to system. In *Proceedings of the 52nd Annual Design Automation Conference*. 1–6.

[58] Yongpan Liu, Jinshan Yue, Hehe Li, Qinghang Zhao, Mengying Zhao, Chun Jason Xue, Guangyu Sun, Meng-Fan Chang, and Huazhong Yang. 2017. Data backup optimization for nonvolatile SRAM in energy harvesting sensor nodes. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 36, 10 (2017), 1660–1673.

[59] Kaisheng Ma, Xueqing Li, Shuangchen Li, Yongpan Liu, John Jack Sampson, Yuan Xie, and Vijaykrishnan Narayanan. 2015. Nonvolatile Processor Architecture Exploration for Energy-Harvesting Applications. *IEEE Micro* 35, 5 (2015), 32–40. https://doi.org/10.1109/MM.2015.88

[60] Kaisheng Ma, Yang Zheng, Shuangchen Li, Karthik3 us Swaminathan, Xueqing Li, Yongpan Liu, Jack Sampson, Yuan Xie, and Vijaykrishnan Narayanan. 2015. Architecture exploration for ambient energy harvesting nonvolatile processors. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 526–537. https://doi.org/10.1109/HPCA.2015.7056060

[61] Kiwan Maeng, Alexei Colin, and Brandon Lucia. 2017. Alpaca: Intermittent Execution without Checkpoints. *Proc. ACM Program. Lang.* 1, OOPSLA.

[62] Kiwan Maeng and Brandon Lucia. 2018. Adaptive Dynamic Checkpointing for Safe Efficient Intermittent Computing. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI'18)*.

[63] Michele Magno and David Boyle. 2017. Wearable energy harvesting: From body to battery. In *2017 12th International Conference on Design & Technology of Integrated Systems In Nanoscale Era (DTIS)*. IEEE, 1–6.

[64] Michele Magno, Dario Kneubühler, Philipp Mayer, and Luca Benini. 2018. Micro kinetic energy harvesting for autonomous wearable devices. In *2018 International symposium on power electronics, electrical drives, automation and motion (SPEEDAM)*. IEEE, 105–110.

[65] Swatilekha Majumdar, Sandeep Kaur Kingra, Manan Suri, and Manish Tikyani. 2016. Hybrid CMOS-OxRAM based 4T-2R NVSRAM with efficient programming scheme. In *2016 16th Non-Volatile Memory Technology Symposium (NVMTS)*. IEEE, 1–4.

[66] Shoichi Masui, Wataru Yokozeki, Michiya Oura, Tsuzumi Ninomiya, Kenji Mukaida, Yoshihisa Takayama, and Toshiyuki Teramoto. 2003. Design and applications of ferroelectric nonvolatile SRAM and flip-flop with unlimited read/program cycles and stable recall. In *Proceedings of the IEEE 2003 Custom Integrated Circuits Conference, 2003.* IEEE, 403–406.

[67] Sparsh Mittal, Jeffrey S Vetter, and Dong Li. 2014. LastingNVCache: A technique for improving the lifetime of non-volatile caches. In *2014 IEEE Computer Society Annual Symposium on VLSI*. IEEE, 534–540.

[68] Sparsh Mittal, Jeffrey S Vetter, and Dong Li. 2014. WriteSmoothing: Improving lifetime of non-volatile caches using intra-set wear-leveling. In *Proceedings of the 24th edition of the great lakes symposium on VLSI*. 139–144.

[69] Tohru Miwa, Junichi Yamada, Hiroki Koike, Hideo Toyoshima, Kazushi Amanuma, Sota Kobayashi, Toru Tatsumi, Yukihiko Maejima, Hiromitsu Hada, and Takemitsu Kunio. 2001. NV-SRAM: A nonvolatile SRAM with backup ferroelectric capacitors. *IEEE Journal of Solid-State Circuits* 36, 3 (2001), 522–527.

[70] Steven Muchnick. 1997. *Advanced compiler design implementation*. Morgan kaufmann.

[71] Taehui Na, Kyungho Ryu, Jisu Kim, Seung H Kang, and Seong-Ook Jung. 2013. A comparative study of STT-MTJ based non-volatile flip-flops. In *2013 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 109–112.

[72] Dushyanth Narayanan and Orion Hodson. 2012. Whole-system persistence. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*. 401–410.

[73] Matheus Almeida Ogleari, Ethan L Miller, and Jishen Zhao. 2018. Steal but no force: Efficient hardware undo+ redo logging for persistent memory systems. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 336–349.

[74] Santhosh Onkaraiah, Marina Reyboz, Fabien Clermidy, Jean-Michel Portal, Marc Bocquet, Chritophe Muller, Costin Anghel, Amara Amara, et al. 2012. Bipolar ReRAM based non-volatile flip-flops for low-power architectures. In *10th IEEE International NEWCAS Conference*. IEEE, 417–420.

[75] Gyuhae Park, Tajana Rosing, Michael D Todd, Charles R Farrar, and William Hodgkiss. 2008. Energy harvesting for structural health monitoring sensor networks. *Journal of Infrastructure Systems* 14, 1 (2008), 64–79.

[76] Sang Phill Park, Sumeet Gupta, Niladri Mojumder, Anand Raghunathan, and Kaushik Roy. 2012. Future cache design using STT MRAMs for improved energy efficiency: Devices, circuits and architecture. In *Proceedings of the 49th Annual Design Automation Conference*. 492–497.

[77] Steven Pelley, Peter M Chen, and Thomas F Wenisch. 2014. Memory persistency. *ACM SIGARCH Computer Architecture News* 42, 3 (2014), 265–276.

[78] Jean-Michel Portal, Marc Bocquet, Mathieu Moreau, Hassen Aziza, Damien Deleruyelle, Yue Zhang, Wang Kang, Jacques-Olivier Klein, YG Zhang, Claude Chappert, et al. 2014. An overview of non-volatile flip-flops based on emerging memory technologies. *Journal of Electronic Science and Technology* 12, 2 (2014), 173–181.

[79] Shashank Priya and Daniel J Inman. 2009. *Energy harvesting technologies*. Vol. 21. Springer.

[80] Emily Ruppel, Milijana Surbatovich, Harsh Desai, Kiwan Maeng, and Brandon Lucia. 2022. An Architectural Charge Management Interface for Energy-Harvesting Systems. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 318–335.

[81] Alanson P Sample, Daniel J Yeager, Pauline S Powledge, Alexander V Mamishev, and Joshua R Smith. 2008. Design of an RFID-based battery-free programmable sensing platform. *IEEE transactions on instrumentation and measurement* 57, 11 (2008), 2608–2615.

[82] Joshua San Miguel, Karthik Ganesan, Mario Badr, Chunqiu Xia, Rose Li, Hsuan Hsiao, and Natalie Enright Jerger. 2018. The eh model: Early design space exploration of intermittent processor architectures. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 600–612.

[83] Shyh-Shyuan Sheu, Chia-Chen Kuo, Meng-Fan Chang, Pei-Ling Tseng, Lin Chih-Sheng, Min-Chuan Wang, Chih-He Lin, Wen-Pin Lin, Tsai-Kan Chien, Sih-Han Lee, et al. 2013. A ReRAM integrated 7T2R non-volatile SRAM for normally-off computing application. In *2013 IEEE Asian Solid-State Circuits Conference (A-SSCC)*. IEEE, 245–248.

[84] Seunghee Shin, Satish Kumar Tirukkovalluri, James Tuck, and Yan Solihin. 2017. Proteus: A flexible and fast software supported hardware logging approach for nvm. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. 178–190.

[85] Jeetendra Singh and Balwinder Raj. 2019. Design and investigation of 7T2M-NVSRAM with enhanced stability and temperature impact on store/restore energy. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27, 6 (2019), 1322–1328.

[86] Weining Song, Yang Zhou, Mengying Zhao, Lei Ju, Chun Jason Xue, and Zhiping Jia. 2018. EMC: Energy-aware morphable cache design for non-volatile processors. *IEEE Trans. Comput.* 68, 4 (2018), 498–509.

[87] Fang Su, Yongpan Liu, Yiqun Wang, and Huazhong Yang. 2016. A Ferroelectric Nonvolatile Processor with 46μs System-Level Wake-up Time and 14μs Sleep Time for Energy Harvesting Applications. *IEEE Transactions on Circuits and Systems I: Regular Papers* 64, 3 (2016), 596–607.
[88] P Thanigai and W Goh. 2015. Maximizing Write Speed on the MSP430 FRAM. *Online] https://www. ti. com/lit/an/slaa498b/slaa498b. pdf* (2015).
[89] Joel Van Der Woude and Matthew Hicks. 2016. Intermittent computation without hardware support or programmer intervention. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 17–32.
[90] Haris Volos, Andres Jaan Tack, and Michael M Swift. 2011. Mnemosyne: Lightweight persistent memory. *ACM SIGARCH Computer Architecture News* 39, 1 (2011), 91–104.
[91] Cong Wang, Naehyuck Chang, Younghyun Kim, Sangyoung Park, Yongpan Liu, Hyung Gyu Lee, Rong Luo, and Huazhong Yang. 2014. Storage-less and converter-less maximum power point tracking of photovoltaic cells for a nonvolatile microprocessor. In *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 379–384.
[92] Jue Wang, Xiangyu Dong, Yuan Xie, and Norman P Jouppi. 2013. i 2 WAP: Improving non-volatile cache lifetime by reducing inter-and intra-set write variations. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 234–245.
[93] Yiqun Wang, Yongpan Liu, Shuangchen Li, Daming Zhang, Bo Zhao, Mei-Fang Chiang, Yanxin Yan, Baiko Sai, and Huazhong Yang. 2012. A 3us wake-up time nonvolatile processor based on ferroelectric flip-flops. In *2012 Proceedings of the ESSCIRC (ESSCIRC)*. IEEE, 149–152.
[94] Mimi Xie, Mengying Zhao, Chen Pan, Hehe Li, Yongpan Liu, Youtao Zhang, Chun Jason Xue, and Jingtong Hu. 2016. Checkpoint aware hybrid cache architecture for NV processor in energy harvesting powered systems. In *Proceedings of the eleventh IEEE/ACM/ifip international conference on hardware/software codesign and system synthesis*. 1–10.
[95] Wei Xu, Hongbin Sun, Xiaobin Wang, Yiran Chen, and Tong Zhang. 2009. Design of last-level on-chip cache using spin-torque transfer RAM (STT RAM). *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 19, 3 (2009), 483–493.
[96] Shuuichirou Yamamoto, Yusuke Shuto, and Satoshi Sugahara. 2009. Nonvolatile SRAM (NV-SRAM) using functional MOSFET merged with resistive switching devices. In *2009 IEEE Custom Integrated Circuits Conference*. IEEE, 531–534.
[97] Jianping Zeng, Jongouk Choi, Xinwei Fu, Ajay Paddayuru Shreepathi, Dongyoon Lee, Changwoo Min, and Changhee Jung. 2021. ReplayCache: Enabling Volatile Cachesfor Energy Harvesting Systems. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 170–182.
[98] Jianping Zeng, Jungi Jeong, and Changhee Jung. 2023. Persistent Processor Architecture. In *MICRO-56: 56th Annual IEEE/ACM International Symposium on Microarchitecture*.
[99] Jianping Zeng, Hongjune Kim, Jaejin Lee, and Changhee Jung. 2021. Turnpike: Lightweight Soft Error Resilience for In-Order Cores. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 654–666.
[100] Yida Zhang and Changhee Jung. 2022. Featherweight soft error resilience for GPUs. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 245–262.
[101] Kazi Abu Zubair and Amro Awad. 2019. Anubis: ultra-low overhead and recovery time for secure non-volatile memories. In *Proceedings of the 46th International Symposium on Computer Architecture*. 157–168.