ReplayCache: Enabling Volatile Caches for Energy Harvesting Systems

Jianping Zeng
Purdue University
USA

Jongouk Choi
Purdue University
USA

Xinwei Fu
Virginia Tech
USA

Ajay P. Shreepathi
Stony Brook University
USA

Dongyoon Lee
Stony Brook University
USA

Changwoo Min
Virginia Tech
USA

Changhee Jung
Purdue University
USA

ABSTRACT

Energy harvesting systems have shown their unique benefit of ultra-long operation time without maintenance and are expected to be more prevalent in the era of Internet of Things. However, due to the batteryless nature, they suffer unpredictable frequent power outages. They thus require a lightweight mechanism for crash consistency since saving/restoring checkpoints across the outages can limit forward progress by consuming hard-won energy. For the reason, energy harvesting systems have been designed with a non-volatile memory (NVM) only. The use of a volatile data cache has been assumed to be not viable or at least challenging due to the difficulty to ensure cacheline persistence.

In this paper, we propose ReplayCache, a software-only crash consistency scheme that enables commodity energy harvesting systems to exploit a volatile data cache. ReplayCache does not have to ensure the persistence of dirty cachelines or record their logs at run time. Instead, ReplayCache recovery runtime *re-executes the potentially unpersisted stores* in the wake of power failure to restore the consistent NVM state, from which interrupted program can safely resume. To support store replay during recovery, ReplayCache partitions program into a series of regions in a way that store operand registers remain intact within each region, and checkpoints all registers just before power failure using the crash consistency mechanism of the commodity systems. For performance, ReplayCache enables *region-level* persistence that allows the stores in a region to be asynchronously persisted until the region ends, exploiting ILP. The evaluation with 23 benchmark applications show that compared to the baseline with no caches, ReplayCache can achieve about 10.72x and 8.5x-8.9x speedup (on geometric mean) for the scenarios without and with power outages, respectively.

ACM Reference Format:
Jianping Zeng, Jongouk Choi, Xinwei Fu, Ajay P. Shreepathi, Dongyoon Lee, Changwoo Min, and Changhee Jung. 2021. ReplayCache: Enabling Volatile Caches for Energy Harvesting Systems. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21), October 18–22, 2021, Virtual Event, Greece.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3466752.3480102

MICRO '21, October 18–22, 2021, Virtual Event, Greece

ACM ISBN 978-1-4503-8557-2/21/10.
https://doi.org/10.1145/3466752.3480102

1 INTRODUCTION

Energy harvesting systems [65] have been deployed in a wide range of application domains, such as Internet of Things (IoT) devices [5, 17, 26, 79], wearables [8, 13, 36, 51, 52], stream and river surveillance [27, 71], health and wellness monitors [6, 7, 16, 61], etc. Energy harvesting systems are well-suited to these domains with the superb property of ultra-long operation time without maintenance by collecting energy from variant ambient sources such as solar, thermal, piezoelectric, and radio-frequency radiation.

However, due to the batteryless nature, energy harvesting systems suffer unpredictable frequent power failure and thus require some form of crash consistency which must be lightweight; otherwise checkpointing/restoring consistent program states across the failure can limit forward progress by consuming hard-won energy. Thus, existing systems [3, 11, 12, 21, 22, 50, 70] have been designed with byte-addressable non-volatile memory (NVM), where data are immediately persisted and thus recoverable at the cost of long latency. While volatile write-back caches can hide the store latency and improve performance with a load hit exploiting data locality, they have been assumed to be not viable or at least challenging in energy harvesting systems.

The crux of the problem is that volatile write-back cache states are not preserved across a power outage. This may lead to an inconsistent NVM state, and therefore the power-interrupted program may fail to resume correctly. That is why existing energy harvesting systems do not use volatile data caches; prior work [50] uses a read-only NVM-based instruction cache where a crash consistency (without stores) is not an issue. Unfortunately, it is a challenging problem to ensure correct data cache persistence in a lightweight manner to maintain forward progress. For example, software logging causes serious performance degradation (100-300% slowdown) since each regular store is preceded by the log store, cacheline flush, and store fence [23, 24, 31, 40, 66, 73, 75].

One possible hardware solution is to use a volatile write-through cache. It allows energy harvesting systems to benefit from load hits and to ensure crash consistency by enforcing that the completion of a store instruction guarantees the persistence of the data in NVM. However, write-through cache comes with a performance penalty on each store as conventional cache-free energy harvesting processors. Since they use a simple in-order core without any form of speculation, they cannot hide the data persistence latency.

Alternatively, one can design a persistent write-back data cache, e.g., non-volatile cache (NVCache) [1, 25, 55, 56, 62, 74, 77] and non-volatile SRAM cache (NVSRAMCache) [9, 20, 38, 39, 53, 68, 69].

However, both cache designs have their own problems. Due to the NVM-based design, NVCaches incur high latency and power consumption for each access. NVSRAMCaches embed NVM to backup an SRAM-based cache, and checkpoint/restore the entire SRAM to/from the NVM backup across power failure, leading to consume high energy. While NVSRAMCaches may be as fast as a volatile SRAM cache without power failure, it is hard to maintain the performance with frequent failure—i.e., the norm of energy harvesting—unless they use a lower-power yet fast non-volatile technology which has not been commercialized yet.

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

To support the store replay, ReplayCache partitions program into a series of regions so that the operand registers of store instructions are intact (*i.e.,* not overwritten by the other following instructions) in each region. We refer to this process *store-register-preserving region formation*. Then, at run time, ReplayCache checkpoints all registers just before power failure to secure the store operand registers. We note that the just-in-time register checkpointing is already available in energy harvesting systems: *e.g.,* QuickRecall[22], Hibernus[3], and NVP[50]. During recovery, these checkpointed registers are used to *re-execute the stores* along the same program path as the one before a power failure; for the store replay, a recovery code block is generated for each region, *i.e.,* ReplayCache directs program control to the recovery code in the wake of the power failure. After that, ReplayCache can safely resume from the interrupted program point with the checkpointed registers and the recovered consistent NVM.

Experiments with 23 applications from Mibench [19] and Mediabench [35] benchmarks show that compared to the baseline with no caches, ReplayCache can make them 10.72x and 8.5x-8.9x faster (on geometric mean) for the scenarios without and with power outages, respectively. This paper makes the following contribution:

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

2 BACKGROUND AND MOTIVATION

This section discusses the architectures of existing energy harvesting systems (section 2.1), the potential crash consistency problem of using a volatile write-back data cache as is (section 2.2) and the limitations of existing cache solutions (section 2.3).

2.1 Architecture of Energy Harvesting Systems

Energy harvesting systems derive energy from external sources (*e.g.,* solar, thermal, ambient electromagnetic radiation) and mostly store it in a tiny capacitor for small IoT devices such as wearables. Due to the nature of unreliable power supply, energy harvesting systems should be able to save (checkpoint) the current state upon power failure, and restore the program state and seamlessly resume the execution when the power comes backs as if nothing had happened. A power interruption in energy harvesting systems is a frequent, normal event, unlike in high performance computing context. It is thus crucial to design systems for whole system persistence (WSP) [30, 59] so that they efficiently save/restore the program state and make a progress no matter where power failure happens.

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

Figure 1(a) shows the architecture of Non-Volatile Processor (NVP) [49], representing the first group that checkpoints and restores registers in place with some additional hardware support [34, 49, 70]. NVP is equipped with an energy harvester, a voltage monitor, and capacitors (not shown). When the monitor detects impending power failure, *i.e.,* the voltage is about to drop below a certain threshold, it signals the processor to checkpoint all the registers (so-called just-in-time checkpointing) into their neighboring non-volatile flip-flops (NVFF) [37, 58, 60, 64]. When power is secured enough across the failure, the processor restores the register states from the NVFF and resumes the execution from the power-interruption point. As both register and memory states on the resumption point are guaranteed to be the same as the states before a power failure, there is no crash consistency problem. A downside of NVP is the use of additional hardware NVFF.

Figure 1(b) illustrate the architecture of QuickRecall [22], representing the second group that checkpoints/restores the registers to/from the NVM. Similar to NVP (and others), QuickRecall also implements just-in-time (JIT) register checkpointing with a voltage monitor and a capacitor (not shown). When the monitor detects upcoming power failure, it triggers an interrupt whose handler checkpoints all the registers into the NVM. When the power comes back, the recovery runtime reads the checkpointed states from the NVM in order to restore the registers. As in NVP, QuickRecall (and others [2, 3] in this group) has no crash consistency issue. A drawback of QuickRecall is that it should secure a lot more energy than NVP to atomically checkpoint all registers in NVM before impending power failure.

2.2 Crash Inconsistency of Write-back Caches

Adding a cache to energy harvesting systems has a high potential to improve their performance (with load hits) and allow them to make more progress for a given energy harvested. However, a naive integration of volatile write-back data cache with existing energy harvesting systems (*e.g.,* NVP, QuickRecall) for performance, may lead to a crash consistency problem, as depicted in Figure 1(c).

Suppose the NVM has the memory state $X = 0$ and $Y = 0$ initially. And suppose a program has a power outage after executing two stores $W(X) = 1$ and $W(Y) = 1$. Before the outage, the cache had the

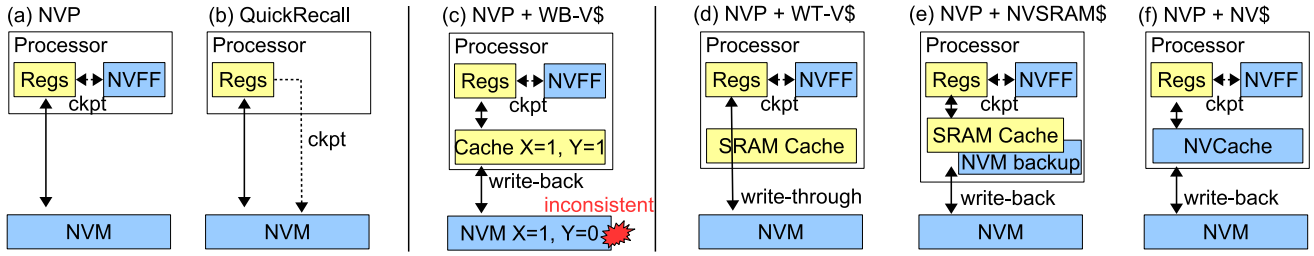


Figure 1: The architectures of existing energy harvesting systems

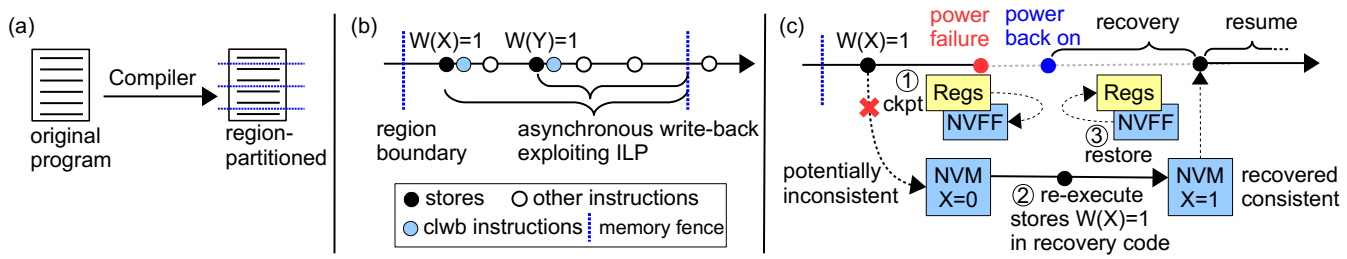


Figure 2: An overview of ReplayCache.

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

2.3 Limitations of Existing Cache Solutions

There are four possible solutions to address the crash consistency problem. The first approach is to use a write-through cache. Figure 1(d) illustrates a case in which NVP is configured with a volatile write-through cache (a traditional SRAM-based one). The write-through policy ensures data consistency as the completion of a store instruction ensures the data persistence to NVM. However, the downside is a long store latency (as in the case without a cache); more precisely, for a write miss, the critical path is lengthened due to the write-allocation policy. Since most of the energy harvesting systems are designed with a simple in-order processor, it is impossible hide the store latency.

As shown in Figure 1(e), the second approach is to equip the processor with the NVSRAMCache that embeds NVM (*e.g.,* ReRAM) to traditional SRAM cache for its backup and restoration [10, 33, 57, 76]. As with NVP, NVSRAMCache also relies on a voltage monitor for just-in-time checkpointing of the SRAM cache. When power is about to be cut, NVSRAMCache triggers a copy from SRAM to NVM for all the cachelines. Along with their restoration, the entire cache backup makes NVSRAMCache consume high energy across power failure. Moreover, NVSRAMCache significantly postpones the booting time due to the high amount of energy that must be secured for failure-atomic cache checkpointing. Although researchers attempt to improve the backup latency [38, 69], their NVSRAMCaches are more of a forward-looking technology in an ideal form—since none of current non-volatile materials can provide comparable latency to SRAM [12].

The third approach is NVCache [20, 54] that leverages a pure non-volatile technology as the cache material; see Figure 1(f). Since NVCache usually uses a slight faster NVM technology for the cache than the non-volatile main memory, the NVCache accesses are a lot slower—consuming more energy—than those of traditional SRAM cache. Thus, NVCache-equipped energy harvesting systems only occasionally outperform cache-free systems when there is very high locality. In sum, the second and third approaches—Figures 1(e) and (f)—are to make a cache itself persistent surviving power failure, but they suffer from their own problems.

Finally, data loggings are another approach to crash consistency in the presence of a volatile cache. However, they dramatically increase execution time (or power consumption if implemented in hardware), prohibiting their use in energy harvesting systems. For example, iDO [40] and Mnemosyne [73] incur 100-300% slowdown, prohibiting their use in an energy harvesting system. Furthermore, since they only supports crash consistency for a few transactions or failure-atomic sections, additional overheads should be paid for whole system persistence (WSP) [30, 59]. Similarly, existing WSP schemes for cache-free harvesting systems such as Alpaca [29] and Ratchet [72] also cause unacceptable slowdown (60% - 500%). Since they assume no cache, their overheads would be even worse for cache-enabled systems due to the additional cacheline flush and fence overhead.

3 OVERVIEW OF REPLAYCACHE

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

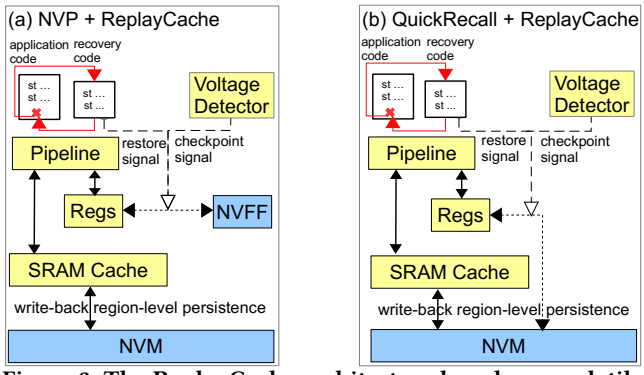


Figure 3: The ReplayCache architecture based on a volatile write-back cache. ReplayCache can be combined with existing energy harvesting systems such as (a) NVP and (b) QuickRecall. Voltage detector is available in every energy harvesting system.

3.1 Program Region Partitioning

As shown in Figure 2(a), ReplayCache compiler partitions entire program input to a series of regions. Each region ensures that the operand registers (*e.g.,* address, value) of a store therein are not overwritten by any other succeeding instructions in that region.

3.2 Region-level Persistence

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

Unlike a traditional write-back cache, ReplayCache ensures that all the stores in a region are persisted (written back to the NVM) before the region ends; this paper calls this *region-level* persistence guarantee in which the persistence latency of in-region stores can be naturally hidden by ILP; Figure 2(b) illustrates the window of potential ILP gain, and the unpersisted state of each store. This region-level persistence assures that at the moment of a power outage, all the stores in the preceding program regions have already been persisted, and only the stores in the interrupted region could not potentially be unpersisted.

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

3.3 Register Checkpointing

Across a power outage, ReplayCache saves register states just before the outage and restores them in the wake of the outage using the voltage monitor based JIT checkpointing mechanism (section 2.1) in commodity energy harvesting systems. For instance, NVP and QuickRecall can both checkpoint register states before the power off and to restore them after the power on as discussed in section 2.1. In Figure 2(c), step ① illustrates that ReplayCache checkpoints the registers when power is about to be cut off.

3.4 Power Failure Recovery

The recovery protocol works as follows. Upon a power outage, the interrupted region's stores before the outage may or may not be persisted, *e.g.,* $W(X) = 1$ in Figure 2(c) unpersisted till the outage—while all preceding regions' stores are guaranteed to be persisted and thus consistent (due to the region-level persistence). In the wake of the outage, ReplayCache jumps to the recovery code block of the interrupted region to replay all the stores left behind the outage. The recovery code block re-executes such unpersisted stores using the checkpointed register values in either NVFF (NVP) or NVM (QuickRecall). This is shown as a step ② of Figure 2(c). Finally, the recovery code sets off a restoration signal to restore all registers (including PC) from NVFF or NVM, and then resumes the program from the outage point with the restored register and the recovered NVM states as in step ③ of Figure 2(c). In this way, ReplayCache allows energy harvesting systems to seamlessly leverage a data cache without amplifying NVM stores.

Figure 3 depicts how ReplayCache works for existing energy harvesting systems, *i.e.,* NVP and QuickRecall, using the aforementioned recovery protocol. The takeaway is that ReplayCache enables the commodity systems to leverage write-back volatile data caches as is with help of the region-level persistence and the recovery code based recovery. The details of recovery code block generation is presented in Section 5.

4 REPLAYCACHE COMPILER

This section describes how ReplayCache compiler realizes the *store-register-preserving region formation*. The compiler's role is 3-fold: (1) region formation (2) CLWB insertion after each store, and (3) recovery code generation whose discussion is deferred to Section 5.

For region formation, the compiler partitions program into a series of small regions so that in each region, no operand registers of a store instruction are overwritten by the following instructions. That way store registers remain intact from the execution of their region all the way to the power failure recovery time on which ReplayCache replays the same stores in case they were not persisted before the failure. We refer to this property as *store integrity*.

Figure 4 shows a high-level workflow of ReplayCache compiler which introduces 3 additional phases (shaded in the figure) to the standard backend compilation passes. This region formation is performed in a whole-program manner to cover the entire program stores, *i.e.,* every single program point belongs to one of the regions.

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

Problem 1. Circular Dependence: Intuitively, the store-register-preserving region formation can be realized with two phases: (1) region partitioning that counts stores to place a region boundary, i.e., store fence instruction, in program and then (2) register preservation that extends the live interval of store operands to the end of each region for their exclusive register use. Thus, the register preservation depends on the region partitioning. However, since the partitioning counts the stores to determine where to place a region boundary, it also depends on the register preservation—forming

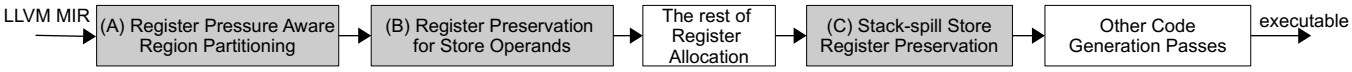


Figure 4: The workflow of ReplayCache compiler.

a circular dependence; the live interval extension of the register preservation increases the register pressure, *i.e.,* the number of necessary registers. Due to the register file size limitation, some registers could be spilled (written) to stack through stores. We call them stack-spill stores.

Problem 2. Stack-Spill Stores: In addition to regular stores, ReplayCache also needs to ensure the integrity of stack-spill stores for correct failure recovery. However, it is hard for the region partitioning to figure out in advance what variables are to be spilled to stack. That is because stack-spill stores are determined in the later register allocation pass assigning physical registers. One might try to perform the region partitioning after the register preservation to exactly count the number of stores. However, this is not a viable option since the region partitioning depends on the register preservation in the first place.

ReplayCache Approach to the Problems: To break the circular dependence between the region partitioning and the register preservation, ReplayCache first considers a function call boundary as initial regions and conducts (A) *register-pressure aware region partitioning* (the first box of Figure 4) to fine-cut the initial regions as needed. Our register-pressure tracking algorithm allows the region partitioning phase not only to estimate the number of stack-spill stores, breaking the dependence on the register preservation, but also possibly to form a region with no spill in a best-effort manner. In case register allocation actually generates stack-spill stores in the formed region after the (B) register preservation phase, ReplayCache runs a post-processing (C) *stack-store register preservation* phase (the fourth box of the figure) that runs through the register-allocated code to find those stack-spill stores whose registers are overwritten in their region, and places a region boundary before the register updates. The rest of this section details the three phases with referring to them with (A), (B), and (C), respectively.

4.1 Register Pressure Aware Partitioning

ReplayCache initially forms regions at function call boundaries and the end of conditional branches, and then runs the register-pressure aware region partitioning algorithm, which aims to achieve two goals. First, it attempts to maximize the length of a region to provide ReplayCache with long potential ILP window for its region-level persistence; see Figure 2 (b). Second, it tries to minimize stack spills generated by the later register allocation phase.

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

Figure 5(a) shows an example code where there are variables x, y, z and their live intervals; x and y are used as store operands, and their live intervals overlap in basic block A as shown in the left of the figure. Suppose there are only 2 physical registers. Figure 5(b) demonstrates how the register-pressure aware region partitioning works for the example code. Basically, whenever stores are encountered, the algorithm carries the live interval of their operands for the rest of the CFG traversal. For example, when the traversal hits the store y at the end point of basic block B in the left control path, the algorithm will start carrying the live interval of y thereafter (illustrated as a hatched box in the figure); the same action is taken with the store x in the right path. Thus, when the traversal hits the point where z's assignment is found in the join basic block D, the live intervals of both x and y have been carried to the point. Since z's live interval starts there, the algorithm places a region boundary at that point, which would otherwise end up making the number of overlapping intervals (3 thereafter) bigger than the number of physical registers (2).

4.2 Regular Store Register Preservation

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

For example, as shown in Figure 5(c), the actual live intervals of x stops at its last use point in basic block C, the resulting interval is extended to the next region boundary placed in the middle of the bottom basic block D; similarly, y's interval is extended to the same following region boundary. In this way, x and y never share their physical registers—even after their last use point—with other variables. In other words, the next register allocation phase ensures that neither x nor y is assigned to any physical register used by other variables. Consequently, ReplayCache guarantees the integrity of the regular stores' registers.

4.3 Stack-Spill Store Register Preservation

The register allocation might spill some variable to stack and generate the stack-spill stores. This actually happens since register allocation performs in a function level (not a region level) and makes a global decision across all the regions in a function—though the (A) register-pressure aware region partitioning tries to form spill-free regions in a best-effort manner. Just in case, this stack-spill store register preservation phase searches the register-allocated code of each region for any update on the spill store registers. For example, in Figure 5(d), a $r1$ is spilled to the stack in basic block D, *i.e.,* the stack-spill store of $r1$ is generated there. However, in the

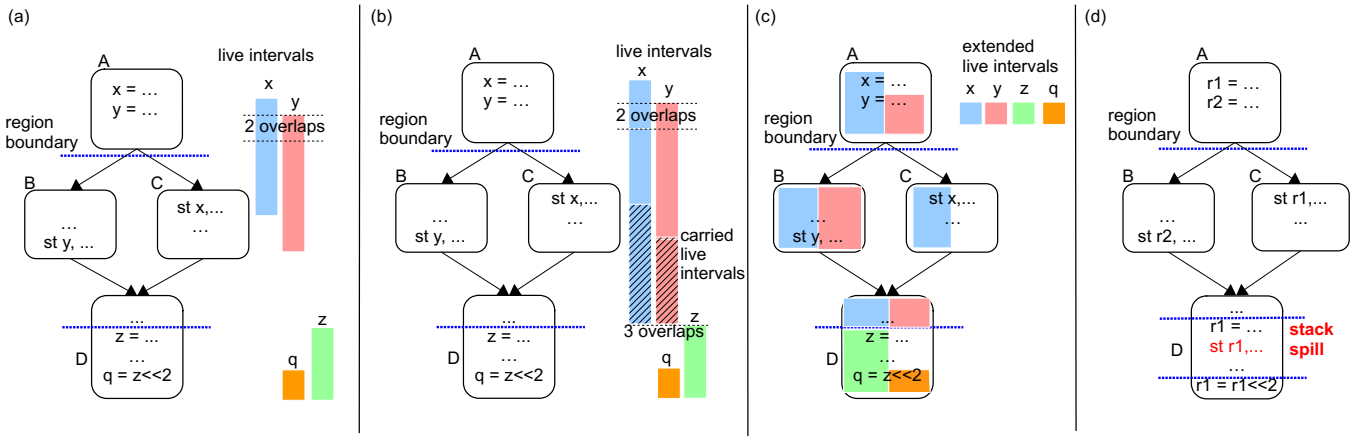


Figure 5: An example partitioned program with live intervals; (a) shows an initial region boundary at the function beginning (basic block A) and live intervals of variables x, y, and z; (b) shows the second boundary inserted in basic block D over which live intervals of x, y are carried, when the partitioning threshold (physical registers) is two; (c) shows extended intervals of all three variables towards the second region boundary; and (d) shows the case of redefining the register $r1$ of spill store in basic block D after variables are assigned to the physical register.

region, the spill store is followed by the instruction that changes the $r1$, *i.e.*, $r1 = r1 \ll 2$. Thus, the region cannot guarantee the integrity of $r1$ used by the stack-spill store from that moment. To deal with this problem, this phase places an additional region boundary right before the register updating instruction to separate it from the stack-spill store; the resulting boundary is shown near the bottom of basic block D in Figure 5(d). Consequently, ReplayCache compiler guarantees the integrity of all the store registers in all regions.

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

5 RECOVERY PROTOCOLS

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

5.1 Recovery Code Generation

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

In particular, to ensure the absence of power failure during the recovery process, ReplayCache compiler leverages the EH model [67] to estimate the worst-case execution energy of the recovery code block. If the energy is greater than what the underlying capacitor can deliver with it full capacitance[1], the compiler splits the corresponding region into two smaller regions and generate their recovery code blocks; this process is repeated unless the resulting code blocks are small enough to complete with the fully charged capacitor. In this way, ReplayCache guarantees the power-failure-free recovery. According to experimental results (§6), ReplayCache regions are not that long; we have not encountered any regions that must be split during our evaluation of total 23 benchmark applications.

5.2 Recovery by Re-execution

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

When the power comes back, ReplayCache first finds out the start address of an interrupted region. It loads the checkpointed *region register* – a dedicated general-purpose register by compiler as mentioned in section 3.2 – from NVFF or checkpointing storage, and

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

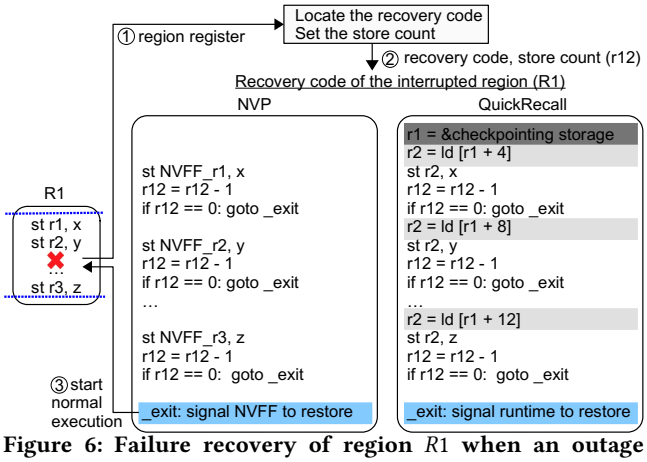


Figure 6: Failure recovery of region $R1$ when an outage happens in the middle of basic block A. Upon recovery, ReplayCache locates a recovery code and counts the number of stores needed to be re-executed ①. Then it re-plays all stores in the recovery block by using checkpointed store operand registers in NVFF ②. Finally, it goes back to the failure point by restoring registers from NVFF and continues the normal execution ③.

locates the recovery code and the SC table of the interrupted region by looking up the RM and CM, respectively. ReplayCache gets the number of store instructions to be re-executed from the beginning of the region to the failure PC by performing binary search of the SC table with the region register as a key. Subsequently, ReplayCache runtime jumps to the recovery code of the interrupted region with the re-executing store count in a register. As illustrated in Figure 6, the recovery code is a series of re-executing the store instruction, decrementing the store counter, and checking if the counter is zero. After executing the specified number of store instructions (*i.e.,* the store counter becomes zero), ReplayCache runtime signals voltage monitor to restore register files from either NVFF or checkpointing storage and thus goes back to the failure point because PC now points to the failure point.

5.3 A Running Example

We illustrate a recovery example in Figure 6. ReplayCache compiler ensures that registers that are used for store operand ($r1$, $r2$, and $r3$) are never updated in region $R1$. When entering into $R1$, ReplayCache sets the region register to the beginning of $R1$. When a power outage happens in the region indicated by a red cross, all registers, including the region register and PC, are checkpointed. At this point, the stores to memory locations x and y may or may not be persisted due to the volatile cache.

When the power comes back, ReplayCache first loads the region register, which points to the beginning of the interrupted region. Then it locates the corresponding recovery code and the number of stores to be re-executed from the RM and SC table ①. ReplayCache jumps to the recovery code to re-execute the same number of store instructions in the region before the failure ②. In the recovery code examples in Figure 6, $r12$ is the number of stores to be re-executed during recovery. In the recovery code, ReplayCache runtime loads the checkpointed store operand registers (*e.g., NVFF_r1* in NVM, and *ld* [$r1 + 4$] in QuickRecall) and re-executes store instructions. Once ReplayCache runtime re-executes the same number of store instructions – *i.e.,* all store instructions to the failure PC are re-executed, the store counter ($r12$) becomes zero and the runtime prepares to resume the normal execution (*goto_exit* colored in blue). The runtime signals voltage monitor to restore register files from NVFF and jumps to failure point ③. The recovery code are slight different between NVP and QuickRecall. As shown in the right, QuickRecall loads the checkpointed registers from the storage (colored in gray).

6 EVALUATION

6.1 Methodology

6.1.1 Compiler. We implemented all ReplayCache compiler passes using the LLVM compiler infrastructure [32]. In particular, we implemented our LLVM passes on MIR (Machine IR) level after instruction selection to precisely measure the number of live intervals during the region construction. The all compiler passes consist of about 1700 LOC excluding comments.

6.1.2 Architecture. We evaluate ReplayCache using a gem5 simulator [4] with ARM ISA, modeling a single core in-order processor with 16 registers, based on the NVPsim [18]; Table 1 summarizes our NVM write/read latency based on [18, 47, 48, 63]. In particular, we only modified L1D cache leaving L1I cache as NVM cache as with the original NVP [49]. Note that ReplayCache works for any energy harvesting processors that support just-in-time (JIT) register checkpointing. In addition to NVP, we test ReplayCache on top of QuickRecall whose simulation configuration follows that of NVP other than the JIT checkpointing/restoration parameters. Table 2 shows the detailed simulation parameters of NVP and QuickRecall. Since QuickRecall checkpoints registers in NVM, its checkpoint/restore voltage thresholds are higher than those used by NVP.

6.1.3 Other Cache Designs and the Default Setting. In addition to ReplayCache, we test 3 alternative cache designs: non-volatile cache (NVCache), non-volatile SRAM cache (NVSRAM), and volatile write-through cache (WT-VCache). All 4 cache designs are assumed to run with NVP unless noted otherwise. Especially for NVSRAM, we use the same configuration used by NVPsim [18], which is based on advanced ReRAM technology. That is, it writes 3x faster with 5x less energy compared to conventional ReRAM based non-volatile main memory does. Similarly, it reads 2x faster with 24x less energy compared to the main memory does. Thus, NVSRAM here serves as the upper bound for performance comparison due to the forward-looking technology used. As our default setting, we set the size of all the caches to 8KB, and they are all 2-way set-associative. For non-volatile main memory, we used Re-RAM by default and set its size as 16MB by leveraging NVMain [63]. We also perform sensitivity studies with STT-RAM and PCM using the parameters in Table 1.

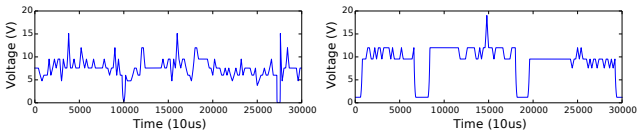
6.1.4 Benchmarks and Power Traces. We use 8 applications in Mibench [19] and 15 applications in Mediabench [35] benchmark suites [46]. All the applications are compiled by ReplayCache compiler with -O3 optimization level. To evaluate ReplayCache for

Table 1: The timing parameters (ns) of different NVM technologies: *e.g.,* tCK stands for clock period.

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

Table 2: Simulation configuration.

	NVP (default)	**NVP** (NVSRAM)	**QuickRecall**
Vmax/Vmin[70]	3.3/2.8	3.5/2.8	3.5/2.8
Ckpt/Restore[70]	2.9/3.2	3.2/3.4	3.1/3.3
Recovery	NVFF+Cache	NVFF+Cache	VFF+Cache



(a) Power Trace 1 (Home) **(b) Power Trace 2 (Office)**

Figure 7: Energy harvesting traces showing voltage input fluctuations in two different places within about 250~400ms from an RF energy harvesting reader [18].

realistic energy harvesting environment with frequent power outages, we use two power traces of the NVPsim which were collected from real RF energy harvesting systems [18]. Figure 7 describes the shape of those two power traces; (a) shows the voltage fluctuations across time in home, and (b) shows those in office. Trace 2 (office) has more power outages than Trace 1 (home); in every 30 seconds, Trace 1 and 2 incur ≈20 and ≈400 power outages, respectively.

6.2 Performance Comparison

6.2.1 Performance without Power Outage. Figure 8 shows the performance results of power-failure-free executions. The Y-axis shows the normalized speedup over the baseline without a cache. Overall, ReplayCache improves the performance of all the applications, achieving 11x speedup on (geometric) average. It turns out that NVCache is the worst design as expected because of higher latency (especially stores) then SRAM, but it still improve the performance due to locality exploitation.

Recall that NVSRAM uses a traditional SRAM cache with an NVM (advanced ReRAM) backup, and checkpoints/restores the whole cache state to/from the NVM backup across power failure. Thus, with no power outage, NVSRAM should perform as an original write-back volatile cache. NVSRAM performs the best as expected achieving 14x speedup compared to the baseline. Here, the performance gap between NVSRAM and ReplayCache results from the store write-back latency that our region-level persistence did not manage to fully hide with ILP. Later in section 6.3, we present the detailed results on ReplayCache's ILP efficiency, reflecting the amount of stalls at the region boundary.

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

6.2.2 Performance with Power Outages. Figures 9 and 10 show the performance results with power failures, simulated on Power Traces 1 and 2 in Figure 7. The simulation includes different sequences of power up/down and downtime during charging. Again, the Y-axis is the normalized speedup over the baseline without a cache.

Although NVCache uses the same NVM technology as main memory, it can be placed close to a core as cache in that core-to-NVCache access is faster than core-to-NVM one. NVCache remains the worst mainly due to a long cache access latency and higher energy consumption of NVM access wasting hard-won energy.

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

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

Overall, ReplayCache achieves 8.95x (Trace 1) and 8.46x (Trace 2) average speedups compared to the baseline (no cache), outperforming NVCache and WT-VCache. The reason for the performance gain over them is two-fold. First, ReplayCache costs less cache power consumption compared to the NVCache and WT-VCache as shown in Figure 11. Second, due to the ILP nature, ReplayCache can hide the most of write-back latency as will be shown Figure 12.

6.2.3 Energy Consumption Breakdown. To figure out the energy consumption behavior of ReplayCache, we measured how much energy was consumed for each part of the system, *i.e.,* cache, memory, and core (NVP computation), by using the power model provided by NVPsim [18]. Figure 11 shows the resulting energy consumption breakdown, normalized to the same no-cache baseline, using the Power Trace 2. Overall, ReplayCache turns out to be very effective, allowing NVP to spend more energy for computation rather than memory access compared to other schemes. Also, ReplayCache's energy consumption is on par with the ideal NVSRAM. As a result, ReplayCache enables NVP to make a significantly further forward progress than the no-cache baseline.

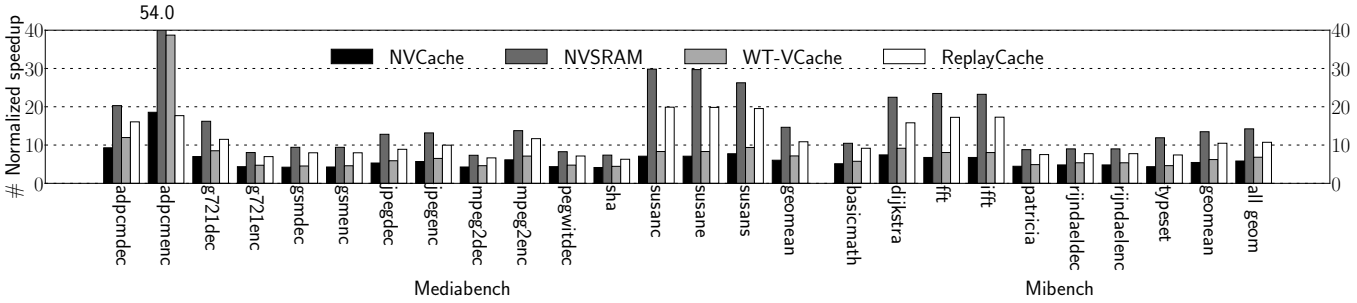


Figure 8: Performance results "without" power outages. We compare ReplayCache with NVCache, NVSRAMCache, and WT-VCache. Y-axis shows the normalized speedup over the baseline without a cache. The higher, the faster.

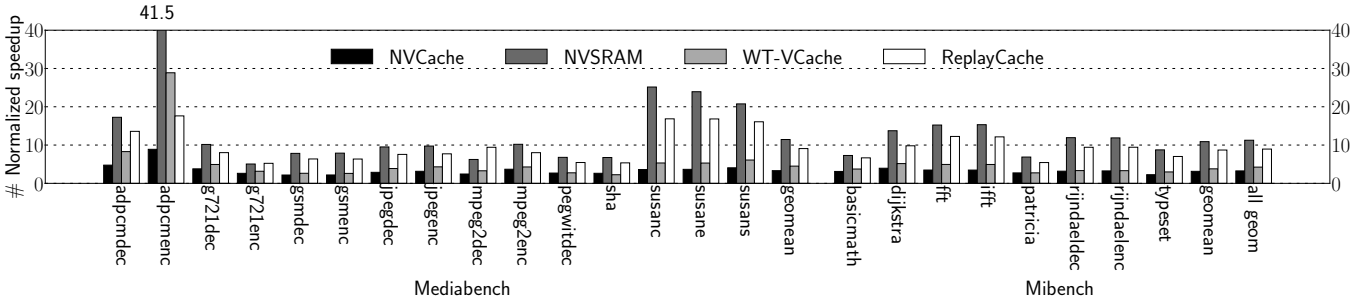


Figure 9: Performance results "with" power outages, simulated with Power Trace 1 in Figure 7(a). We compare ReplayCache with NVCache, NVSRAMCache, and WT-VCache. Y-axis shows the normalized speedup over the baseline without a cache.

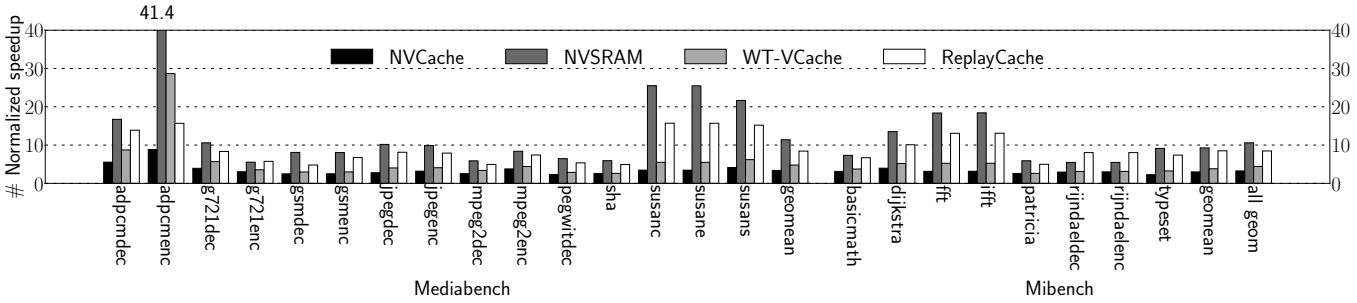


Figure 10: Performance results "with" power outages, simulated with Trace 2 in Figure 7(b). We compare ReplayCache with NVCache, NVSRAMCache, and WT-VCache. Y-axis shows the normalized speedup over the baseline without a cache.

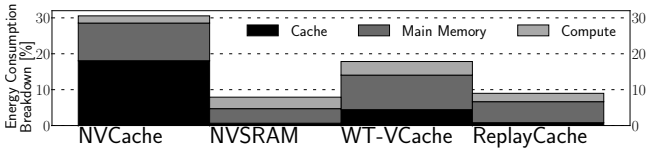


Figure 11: Normalized energy consumption breakdown (trace 2) compared to the baseline without a cache.

6.3 Instruction Level Parallelism Efficiency

ReplayCache exploits ILP for stores and thus is faster than a volatile write-through cache. Nevertheless, its ILP can be bounded by region-level persistence guarantee, *e.g.,* a region end is reached before the preceding store completes the NVM persistence, in which case ReplayCache is slower than an ideal write-back cache. With that in mind, we investigate the amount of ILP that ReplayCache can exploit, based on the power-failure-free simulation results, to reason about ReplayCache's high performance.

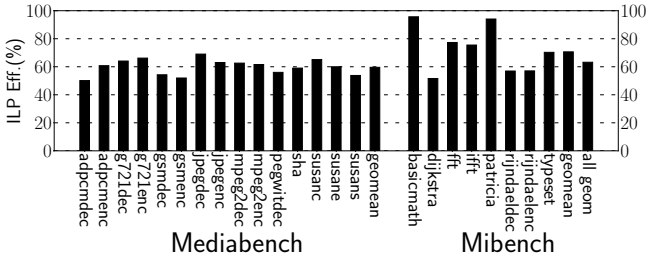


Figure 12: Instruction-level parallelism efficiency "without" power failure.

Let N be the total number (dynamic instances) of stores in a region. Among them, N_{no_stall} represents the number of stores that do not stall, and N_{stall} represents the number of stores that stall at the region boundary for region-level persistence guarantee. Let C be the cycles required for a store to be persisted in the NVM (*i.e.,*

the write-through NVM store latency; 31 cycles in our evaluation for default ReRAM); and $S(i)$ be the stall cycles of i's store in the region. We then calculate the ILP efficiency at a 0-to-100% scale. For each store, the worst efficiency 0% is made when the processor waits for C cycles after the region finishes, and the best efficiency 100% reflects 0 stall cycle. Equation (1) defines the ILP efficiency for N stores in a region as follows.

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

Figure 12 shows the ILP efficiency of the tested applications. On average, ReplayCache achieves 63% ILP across the evaluated applications, and the ILP efficiency explains why ReplayCache achieves the performance shown in Figure 8. Again, in our evaluation, the write-through store latency takes 31 cycles [18], i.e., $C = 31$. This implies that ReplayCache can hide about 20 cycles out of the 31 cycles on average.

6.4 Binary Size Analysis

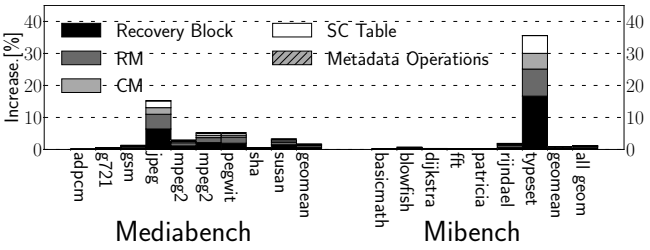


Figure 13: Binary size increase due to recovery block, metadata (RM, CM, SC table), and metadata operations (code).

Figure 13 demonstrates the breakdown of binary size increase of ReplayCache binaries as a percentage increase compared to the baseline binary. Overall, ReplayCache incurs only 1.2% binary size overhead on average. Metadata operations are comprised of roughly 110 instructions, leading to near-zero overhead. Only 2 applications, *e.g.,* `jpeg` and `typeset`, have observable binary size increase because they have lots of small regions. Note that the binary size overhead never puts pressure on application's memory usage at run time. That is because the metadata is accessed only at boot time on which ReplayCache's recovery starts with empty cache—already wiped out upon the prior failure—without cache pollution.

6.5 Dynamic Instruction Count Analysis

Figure 14 demonstrates that ReplayCache compiler only increases dynamic instruction count by 2.49% on average compared to the baseline binary. Note that this is not a critical performance limiting factor as confirmed in Figure 8-10 where ReplayCache consistently shows significant speedups.

6.6 Sensitivity Study

6.6.1 Cache Size. Figure 15 shows the normalized execution time (to the baseline without a cache) of alternative cache schemes with a different cache size from 512B to 8KB using Power Trace 2. The results show that ReplayCache matches the performance of NVSRAM cache (that is an ideal write-back cache in power-failure-free scenarios) for small cache size, such as 512B and 1KB.

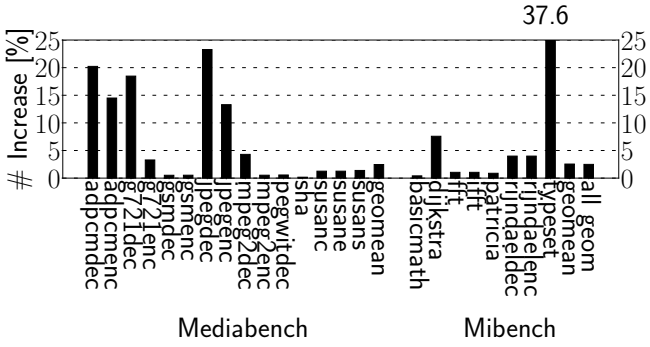


Figure 14: Dynamic instruction count increase due to ReplayCache compiler code generation; lower is better.

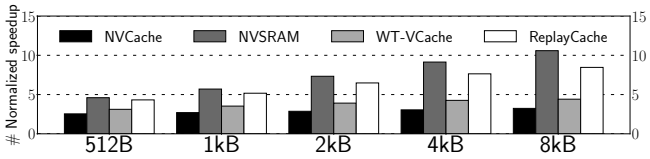


Figure 15: Cache size sensitivity analysis for Trace 2.

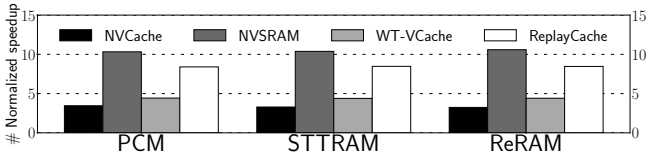


Figure 16: Sensitivity study on different NVMs with trace 2 used.

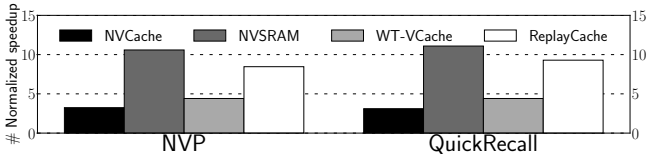


Figure 17: Performance overhead comparison with trace 2.

6.6.2 NVM Technology. Different NVM technologies (*e.g.,*, ReRAM, PCM, and STT-RAM) have different write/read latency properties as summarized in Table 1. For ReRAM, PCM, and STT-RAM (as the main memory), Figure 16 shows the normalized speedup of alternative cache schemes, compared to their 3 baselines without a cache. It turns out that ReplayCache consistently achieves significant speedups across the NVM technologies (8.4x-8.46x).

6.6.3 NVP versus QuickRecall. To analyze the impact of the underlying just-in-time register checkpointing on ReplayCache's performance, we tested all four cache schemes on top of QuickRecall and compared the results with those of NVP. Again, we used the Power Trace 2 and normalized the speedup over their baselines, *i.e.,* NVP/QuickRecall without cache. Figure 17 describes that the performance trend is similar to NVP; however, it is worth noting that QuickRecall requires higher checkpoint/restoration voltage due to data backup as shown in Table 2—though it is a less expensive system than NVP due to the lack of non-volatile flip-flops.

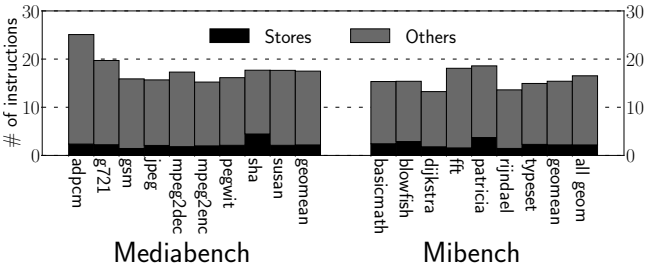


Figure 18: Breakdown of per-region instructions on average.

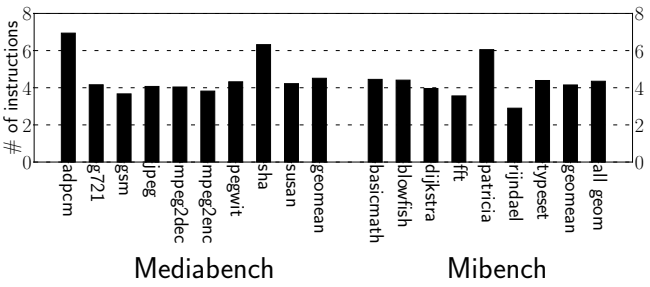


Figure 19: Average distance (the number of instructions) between region's last store and the following region boundary.

6.7 ReplayCache Compiler Region Statistics

We study the region statistics, statically calculated from the binary built by our compiler. Figure 18 presents the average number of instructions per region. On average, there are 16.4 instructions per region. We also break them down into two categories: stores and other instructions. On average, there are 2.18 stores and 14.35 others per region. This implies that the recovery code blocks are not long either (smaller than their regions). In fact, we did not encounter any recovery block that requires the corresponding region to be split to ensure the absence of power failure during the recovery.

Moreover, Figure 19 shows the average distance (the number of instructions) between the last store of a region and the following region boundary, i.e., 4.35 instructions on average. The distance here reflects ReplayCache's ILP opportunities.

7 RELATED WORKS

Many prior works [1, 25, 55, 56, 62, 74, 77] have been proposed to leverage non-volatile caches to speed up the performance and leverage their zero standby leakage and crash consistency free properties. However, the cell endurance of NVM techniques ranges from 10^5 in flash to 10^{12} in STT-RAM. Non-volatile caches may only be able to endure few months for most of real applications [25]. Thus, prior works focus on increasing the lifetime of NVM cells. Furthermore, NVM has the asymmetric performance property. A write is considerably slower than a read, compared to the SRAM counterpart. Both the short lifetime and the long write latency severely limit the use of NVM as L1 cache in practice.

To use the synergy of NVM and SRAM, many researches [10, 20, 33, 38, 39, 53, 54, 57, 68, 69, 69, 78] proposed to incorporate different NVM technologies (*e.g.,* STT-RAM, ReRAM, etc.) with SRAM. Many proposals leverage the NVM part as a just-in-time checkpointing storage of the traditional SRAM-based cache in case of power failure. Thus, the NVM speed is the critical aspect for the success of such SRAM/NVM hybrid design. Although researchers attempt to improve the NVM backup/restoration latency [38, 69], they assume forward-looking technologies; no current NVM technologies provide comparable latency to SRAM [12, 30].

The idea of partitioning a program into multiple regions to design more efficient energy harvesting systems has been explored. Ratchet [72] proposed to partition program into a series of anti-dependence-free (*i.e.,* write-after-read dependence free) regions for idempotent processing as with others [14, 15, 28, 40–43, 45]. Since idempotent regions can be safely re-executed multiple times, it can recover a power-interrupted region by rolling back to the beginning in the wake of power failure, provided the inputs value of the region can survive the power failure. Due to the absence of the anti-dependence, Ratchet only needs to checkpoint all live-in registers of the region at its entry point. Unfortunately, such consecutive NVM writes are not only expensive but also dangerous increasing the chance of power failure in the middle of their writes. To address the issues in Ratchet, Clank [21] proposed hardware-based idempotent processing. Despite its improved performance, Clank requires relatively heavy and complex hardware components such as a fast scratchpad memory for speeding up the writes to the underlying NVM and an expensive CAM (content-addressed matching) search based load/store address tables to dynamically detect anti-dependence. Alternatively, CoSpec [12] proposed power failure speculation assuming that power failure is not likely to occur. Thus, it buffers all the application writes in a gated store buffer [44, 80] in case of misspeculation, *i.e.,* actual power failure. Also, the CoSpec compiler partitions program into a series of regions so that they never overflow the store buffer. When power failure occurs in the middle of a region, it is rolled back to the beginning in the wake of power failure. As with Ratchet, CoSpec needs to pay the overhead of checkpointing all live-in registers of every region. Unlike ReplayCache, neither Clank nor CoSpec supports a volatile data cache. Thus, we suspect that ReplayCache can significantly outperform them.

8 CONCLUSION

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

ACKNOWLEDGMENTS

We thank anonymous reviewers for their comments. At Purdue, this work was supported by NSF grants 1750503 (CAREER) and 1814430. At Stony Brook, this work was supported by NSF grant 2029720. At Virginia Tech, this work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No. 2014-3-00035).

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] Domenico Balsamo, Alex S Weddell, Anup Das, Alberto Rodriguez Arreola, Davide Brunelli, Bashir M Al-Hashimi, Geoff V Merrett, and Luca Benini. 2016. Hibernus++: a self-calibrating and adaptive system for transiently-powered embedded devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35, 12 (2016), 1968–1980.

[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] 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.

[5] 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.

[6] 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*.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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)*. IEEE, 331–344.

[12] 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.

[13] 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.

[14] Marc De Kruijf and Karthikeyan Sankaralingam. 2013. Idempotent code generation: Implementation, analysis, and evaluation. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 1–12.

[15] Marc A De Kruijf, Karthikeyan Sankaralingam, and Somesh Jha. 2012. Static analysis and compiler design for idempotent processing. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*. 475–486.

[16] Tzeno Galchev, J McCullagh, RL Peterson, and K Najafi. 2010. A vibration harvesting system for bridge health monitoring applications. In *Proc. PowerMEMS*. 179–182.

[17] Maria Gorlatova, John Sarik, Guy Grebla, Mina Cong, Ioannis Kymissis, and Gil Zussman. 2014. Movers and shakers: Kinetic energy harvesting for the internet of things. In *The 2014 ACM international conference on Measurement and modeling of computer systems*. 407–419.

[18] 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.

[19] 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.

[20] 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.

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

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] Ervin Kamenar, Saša Zelenika, David Blažević, Senka Maćešić, Goran Gregov, Kristina Marković, and Vladimir Glažar. 2016. Harvesting of river flow energy for wireless sensor network technology. *Microsystem Technologies* 22, 7 (2016), 1557–1574.

[28] 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.

[29] Alexei Colin Kiwan Maeng and Brandon Lucia. 2017. Alpaca: intermittent execution without checkpoints. In *Proc. ACM Program. Lang.1, OOPSLA, Article 96*.

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

[31] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M Chen, and Thomas F Wenisch. 2016. High-performance transactions for persistent memories. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. 399–411.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

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

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.
[43] 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.
[44] 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 *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 25.
[45] Qingrui Liu, Changhee Jung, Dongyoon Lee, and Devesh Tiwari. 2017. Compiler-directed soft error detection and recovery to avoid DUE and SDC via Tail-DMR. *ACM Transactions on Embedded Computing Systems (TECS)* 16, 2 (2017), 32.
[46] Qingrui Liu, Xiaolong Wu, Larry Kittinger, Markus Levy, and Changhee Jung. 2017. Benchprime: Effective building of a hybrid benchmark suite. *ACM Transactions on Embedded Computing Systems (TECS)* 16, 5s (2017), 1–22.
[47] Sihang Liu, Aasheesh Kolli, Jinglei Ren, and Samira Khan. 2018. Crash consistency in encrypted non-volatile main memory systems. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 310–323.
[48] Sihang Liu, Korakit Seemakhupt, Gennady Pekhimenko, Aasheesh Kolli, and Samira Khan. 2019. Janus: optimizing memory and storage support for non-volatile memory systems. In *Proceedings of the 46th International Symposium on Computer Architecture*. ACM, 143–156.
[49] Yongpan Liu, Zewei Li, Hehe Li, Yiqun Wang, Xueqing Li, Kaisheng 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.
[50] Kaisheng Ma, Yang Zheng, Shuangchen Li, Karthik 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)*. IEEE, 526–537.
[51] 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.
[52] 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.
[53] 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.
[54] 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.
[55] 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.
[56] 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.
[57] 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.
[58] 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.
[59] 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.
[60] 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.
[61] 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.
[62] 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.
[63] Matt Poremba and Yuan Xie. 2012. Nvmain: An architectural-level main memory simulator for emerging non-volatile memories. In *2012 IEEE Computer Society Annual Symposium on VLSI*. IEEE, 392–397.
[64] 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. *J. Electron. Sci. Technol* 12, 2 (2014), 173–181.
[65] Shashank Priya and Daniel J Inman. 2009. *Energy harvesting technologies*. Vol. 21. Springer.
[66] Madhava Krishnan Ramanathan, Jaeho Kim, Ajit Mathew, Xinwei Fu, Anthony Demeri, Changwoo Min, and Sudarsun Kannan. 2020. Durable Transactional Memory Can Scale with Timestone. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, James R. Larus, Luis Ceze, and Karin Strauss (Eds.). ACM, 335–349.
[67] 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.
[68] 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.
[69] 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.
[70] Fang Su, Yongpan Liu, Yiqun Wang, and Huazhong Yang. 2016. A Ferroelectric Nonvolatile Processor with 46mus System-Level Wake-up Time and 14mus Sleep Time for Energy Harvesting Applications. *IEEE Transactions on Circuits and Systems I: Regular Papers* 64, 3 (2016), 596–607.
[71] Weipeng Sun, Ting Tan, Zhimiao Yan, Daoli Zhao, Xingqi Luo, and Wenhu Huang. 2018. Energy harvesting from water flow in open channel with macro fiber composite. *AIP Advances* 8, 9 (2018), 095107.
[72] 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.
[73] Haris Volos, Andres Jaan Tack, and Michael M Swift. 2011. Mnemosyne: Lightweight persistent memory. *ACM SIGARCH Computer Architecture News* 39, 1 (2011), 91–104.
[74] 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.
[75] William Wang and Stephan Diestelhorst. 2018. Quantify the performance overheads of PMDK. In *Proceedings of the International Symposium on Memory Systems*. 50–52.
[76] Wei Wei, Kazuteru Namba, Jie Han, and Fabrizio Lombardi. 2014. Design of a nonvolatile 7T1R SRAM cell for instant-on operation. *IEEE transactions on nanotechnology* 13, 5 (2014), 905–916.
[77] 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.
[78] 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.
[79] Cheuk-Wang Yau, Tyrone Tai-On Kwok, Chi-Un Lei, and Yu-Kwong Kwok. 2018. Energy harvesting in internet of things. In *Internet of Everything*. Springer, 35–79.
[80] Jianping Zeng, Hongjune Kim, Jaejin Lee, and Changhee Jung. 2021. Turnpike: Lightweight Soft Error Resilience for In-Order Cores. In *The 54th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press.