Write-Light Cache for Energy Harvesting Systems

Jongouk Choi*
University of Central Florida
jongouk.choi@ucf.edu

Jianping Zeng
Purdue University
zeng207@purdue.edu

Dongyoon Lee
Stony Brook University
dongyoon@cs.stonybrook.edu

Changwoo Min
Virginia Tech
changwoo@vt.edu

Changhee Jung
Purdue University
chjung@purdue.edu

ABSTRACT

Energy harvesting system has huge potential to enable battery-less Internet of Things (IoT) services. However, it has been designed without a cache due to the difficulty of crash consistency guarantee, limiting its performance. This paper introduces Write-Light Cache (WL-Cache), a specialized cache architecture with a new write policy for energy harvesting systems. WL-Cache combines benefits of a write-back cache and a write-through cache while avoiding their downsides. Unlike a write-through cache, WL-Cache does not access a non-volatile main memory (NVM) at every store but it holds dirty cache lines in a cache to exploit locality, saving energy and improving performance. Unlike a write-back cache, WL-Cache limits the number of dirty lines in a cache. When power is about to be cut off, WL-Cache flushes the bounded set of dirty lines to NVM in a failure-atomic manner by leveraging a just-in-time (JIT) checkpointing mechanism to achieve crash consistency across power failure. For optimization, WL-Cache interacts with a run-time system that estimates the quality of energy source during each power-on period, and adaptively reconfigures the possible number of dirty cache lines at boot time. Our experiments demonstrate that WL-Cache reduces hardware complexity and provides a significant speedup over the state-of-the-art volatile cache design with non-volatile backup. For two representative power outage traces, WL-Cache achieves 1.35x and 1.44x average speedups, respectively, across 23 benchmarks used in prior work.

ACM Reference Format:
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 (ISCA '23), June 17–21, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3579371.3589098

1 INTRODUCTION

Energy harvesting systems offer battery-less computing that is deemed to be the next step in the evolution of IoT [56]. Without a battery, energy harvesting systems can self-power their devices by collecting ambient energy from external sources (*e.g.,* radio frequency, WiFi, etc.). They enable various applications such as wearables, sensors, and implantable medical devices in which a battery-equipped design could be bulky, environment-unfriendly, and cost-inefficient—apart from regular battery replacements [3, 15, 30, 54, 55, 63].

However, due to unreliable nature of ambient energy sources, energy harvesting systems suffer from frequent power failures. To mitigate the problem, existing energy harvesting systems leverage a small capacitor as an energy buffer and employ a non-volatile processor (NVP) that can instantly checkpoint/restore all on-chip data, i.e., volatile registers to/from neighboring non-volatile flip-flops at a power failure/recovery point [69]. In addition, the systems use non-volatile memory (NVM), not Flash, as main memory to persist all off-chip data across power failure.[1] Notably, they do not make use of (volatile) cache since its states are lost across power failure causing a crash consistency problem [14, 35]. That is, without a cache, they directly persist data on NVM at the cost of long NVM access latency for every memory operation, resulting in poor performance.

A cache has high potential to significantly improve performance for energy harvesting systems. Given an energy budget, they can make a further forward execution progress by avoiding NVM accesses on cache hits. Unfortunately, leveraging a cache in energy harvesting systems remains a challenge due to the difficulty of crash consistency guarantee. Different cache write policies (*i.e.,* write-though or write-back) have different implications on the crash consistency. A (volatile) write-through cache directly achieves crash consistency as it has no concern about losing volatile cache states across power failures. However, it requires updating both the cache and NVM on each memory store and thus consumes more power, offsetting the caching benefits.

In contrast, a write-back cache does not update NVM until dirty cachelines are evicted to NVM, i.e., write hits do not involve NVM access. However, to ensure crash consistency, it requires additional hardware support that can flush all the updated (dirty) cachelines to NVM before impending power failure. For example, prior works introduce a write-back NVSRAM cache [16, 41, 69] that uses a volatile SRAM cache backed with the same size non-volatile (NV) cache counterpart. When power is about to be cut off, the NVSRAM cache triggers just-in-time (JIT) checkpointing that can failure-atomically flush the entire cache [41, 69] or dirty lines [16] to the neighboring NV counterpart. This approach has two downsides: (1) its hardware modification cost is high—no such fabrication has been adopted for production yet; and (2) it requires reserving a large amount of extra energy enough to backup all cache lines (in the worst case all

*This work was mostly done when the author was a PhD student at Purdue.

ISCA '23, June 17–21, 2023, Orlando, FL, USA.

ACM ISBN 979-8-4007-0095-8/23/06…$15.00
https://doi.org/10.1145/3579371.3589098

[1] Flash memory requires (9x) higher voltage and incurs orders-of-magnitude slower write latency (1000x) than byte-addressable NVM such as FRAM [1].

may be dirty). The reserved energy cannot be used for computation, significantly limiting the forward progress and the energy efficiency.

This paper presents Write-Light[2] Cache (WL-Cache), specialized cache architecture with a new write policy for energy harvesting systems. In particular, WL-Cache builds upon traditional SRAM cache design without requiring non-volatile cache counterpart and combines the benefits of a write-through cache and a write-back cache while avoiding their downsides. Unlike a write-through cache, WL-Cache does not update data in NVM main memory at every store, but it holds dirty lines in a cache to take advantage of locality, saving energy and improving performance. Unlike a write-back cache, WL-Cache permits only a limited number of dirty lines in a cache. That way WL-Cache has only to secure a small amount of energy—so that the bounded number of dirty lines can be JIT-checkpointed—without expensive backup/restoration costs.

To achieve this, WL-Cache tracks a set of dirty cache lines in a separate small hardware queue, called DirtyQueue. Then, WL-Cache uses two reconfigurable thresholds, named maxline and waterline. On the other hand, the maxline threshold ($\leq$ |DirtyQueue|) defines the maximum number of dirty cache lines in WL-Cache. When the number of dirty cache lines reaches maxline, WL-Cache stalls a store instruction until a free slot in DirtyQueue becomes available. Importantly, maxline determines and bounds the amount of energy WL-Cache needs to reserve to failure-atomically checkpoint dirty cache lines—which is much lower than that of the NVSRAM cache—when power failure is impending.

The waterline threshold ($\leq$ maxline) determines when WL-Cache writes back dirty cache lines to NVM. When the number of dirty lines exceed waterline, WL-Cache picks one of dirty lines and asynchronously writes it back to NVM. The persisted cache line remains in the cache with a "clean" state for future references. The asynchronous write-back operation overlaps with the execution of following instructions, realizing instruction-level parallelism (ILP). The gap between maxline and waterline defines the potential ILP opportunity, not available in a write-through cache.

For optimization, WL-Cache interacts with a run-time system that adaptively adjusts the two thresholds depending on the energy harvesting quality. The run-time system measures a power-on period across power outages and estimates the quality of energy source at each reboot time. When the power-on time increases (*i.e.,* the energy source condition is seemingly good), the runtime raises the waterline/maxline and the JIT checkpointing threshold (V_{backup}) accordingly to hold more dirty cache lines in WL-Cacheat each reboot time, making it behave more like a write-back cache. On the contrary, when the power-on time decreases (*i.e.,* possible sign of poor energy harvesting), the runtime lowers the waterline/maxline and the V_{backup} threshold to hold a smaller number of dirty lines. That way WL-Cache starts to act more like a write-through cache and can use hard-won energy more for forward progress—rather than lavishing it on recurring JIT checkpointing of many lines across frequent outages.

Our experiments with 23 applications and 4 different capacitors highlight that WL-Cache always provides significant speedups over the non-volatile cache baseline. With a default capacitor (1uF) where all cache designs deliver the best performance, WL-Cache achieves a 3.1x speedup in the absence of power failure while it does 2.8x and 2.48x speedups for two representative power failure traces, respectively. In particular, WL-Cache even outperforms the ideal NVSRAM cache with a significant margin, achieving 1.35x and 1.44x speedups for the two traces, respectively. Overall, WL-Cache performs the best regardless of the capacitor size.

This paper makes the following contributions:

- This paper presents WL-Cache, a new cache design for energy harvesting systems.
- WL-Cache introduces a new cache write policy that takes advantages of both write-back's efficiency and write-through's persistence.
- WL-Cache adaptively behaves as a write-through or a write-back cache by changing its characteristic back and forth with the quality of energy source in mind.
- Our experiments demonstrate that WL-Cache significantly outperforms existing cache designs, including the state-of-the-art NVSRAM cache, regardless of capacitor size.

2 BACKGROUND AND MOTIVATION

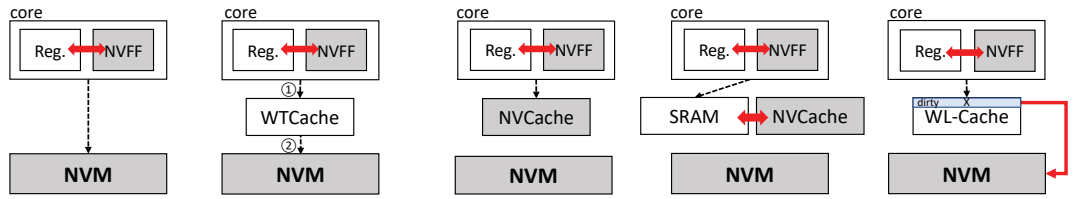
2.1 Energy Harvesting System Architecture

Energy harvesting systems collect ambient energy into a small capacitor and spend the energy for computing without a battery. Due to the battery-less design, they suffer frequent power failures and thus should ensure *crash consistency* to resume a power-interrupted program [9–13, 36, 61, 70]. Unfortunately, prior undo/redo logging-based crash consistency solutions [8, 21–23, 26, 34, 35, 37–40, 68, 76, 77] do not fit well for energy harvesting systems because logging requires additional memory writes, consuming hard-won energy.

To address the challenge, recent works [19, 50, 51, 69] have adopted checkpoint-and-restore-based crash consistency, called *just-in-time (JIT) checkpointing*. They monitor the capacitor by using a voltage monitoring system, and checkpoint all volatile architectural states to NVM just before a power outage occurs. When the power returns, they restore the states and resume the interrupted program.

At a high level, a representative energy harvesting system, Non-Volatile Processor (NVP) [42–49, 66, 67], uses NVM as main memory (without cache) and provides hardware support for JIT checkpointing of volatile registers to ensure whole system persistence guarantee [28, 53]. When its operating voltage drops below a certain threshold (V_{backup}), NVP instantly checkpoints all the registers into their neighboring non-volatile flip-flops (NVFFs). When power is secured enough again (V_{on}), it restores the checkpointed register states from the NVFFs and resumes the interrupted program; we assume that the voltage monitoring system is reliable as with all prior works built on JIT checkpointing [20, 43–49, 66, 67]. Alternatively, other approaches [19, 58] store the volatile registers into the (main memory) NVM by using the (software-based) JIT checkpointing mechanism, instead of using a separate hardware NVFF (at the cost of reserving more energy for software-based JIT checkpointing). Note, they all have been designed without a volatile cache due to the potential crash consistency issue.

[2]As light can behave simultaneously as a particle and a wave, WL-Cache takes advantages of both write-through and write-back writing policies; and WL-Cache is "light"weight.



(a) NVP without cache volatile registers are checkpointed to NVFF in a core at a power failure point.

(b) NVP with WTCache, a write-though volatile cache. A processor updates data in both cache and NVM.

(c) NVP with NVCache, a write-back non-volatile cache. A processor updates data in cache, but it is slow.

(d) NVP with NVSRAMCache. Registers and cache data are checkpointed to NVFF and NVM.

(e) WL-Cache. A processor holds a few dirty lines in the cache and flushes them at a power outage.

Figure 1: Design comparison of cache architectures in NVP. Gray boxes are non-volatile while white boxes are volatile. Red arrows represent JIT checkpointing.

2.2 Cache and Write Policy

A cache allows a system to exploit temporal and spatial locality and thus improves overall performance. Conventional caches employ either write-through or write-back policies, which affect the crash consistency design discussed in the next section. A write-through cache updates both the cache and the main memory at every store. A write-back cache updates only the cache and keeps track of dirty cache lines. It coalesces subsequent write hits on the cache and reduces the write traffic to the main memory, achieving higher performance than the write-through design in many cases.

2.3 Crash Consistency with a Cache

2.3.1 Volatile Write-through Cache. A traditional SRAM-based write-through cache can be used in energy harvesting systems without modification (Figure 1(b)). The write-through policy naturally supports crash consistency by persisting data at every store in a synchronous manner while updating the same data in the cache. However, the requirement of synchronous writes prevents store buffer optimization. The system should pay the long store latency as in the case without a cache. Table 1 (second row) summarizes the pros and cons of write-through cache (WTCache). It does not require extra energy for JIT checkpointing, nor additional hardware (beyond a traditional write-through cache). However, all stores have to travel to NVM, so its performance improvement is limited.

	HW cost	**Energy Buf. Requirement**	**NVM Cache Req.(size)**	**Perf. Improve.**
WTCache	None	No	No	Low
NVCache [6, 18]	Low	No	Yes (Large)	Low
NVSRAM(full) [41]	High	Large	Yes (Large)	High
NVSRAM(ideal) [16]	High+	Large	Yes (Large)	High
NVSRAM(practical) [72, 73]	Medium	Medium	Yes (Medium)	Medium
ReplayCache [75]	None	Small	No	Medium
WL-Cache	**Low**	**Small**	**No**	**High**

Table 1: Hardware complexity and performance comparison in prior cache schemes for energy harvesting systems.

2.3.2 Non-volatile Write-back Cache. A write-back cache addresses the performance issue of a write-through cache by holding dirty lines in the cache without having synchronous writes. However, it raises the crash consistency problem since the main memory could be outdated upon power outage. For crash consistency, NVCache [6, 18] is designed as a full non-volatile cache, instead of a traditional SRAM-based volatile one, as illustrated in Figure 1 (c). However, NVCache is inevitably slower and requires more energy than a traditional SRAM-based cache. As summarized in Table 1 (third row), NVCache does not need to reserve extra energy for JIT checkpointing, but it requires a full non-volatile cache design. The performance impact is limited due to long latency and additional energy consumption. Later we use NVCache as baseline for comparison.

2.3.3 NVSRAMCache. NVSRAMCache [16, 41, 42, 72, 73] couples a traditional write-back SRAM cache with an NVM counterpart as shown in Figure 1(d). It achieves crash consistency via JIT checkpointing; it monitors a remaining energy in a capacitor (energy buffer) and copies the SRAM cache states to the NVM counterpart right before a power loss. As listed in Table 1, NVSRAMCache can achieve higher performance improvement as it uses write-back policy and absorb write hits (unlike WTCache) and it uses a SRAM-based cache at runtime (unlike NVCache). Additionally, NVSRAMCache can resume from a warm cache (as in NVCache). Three versions (full, ideal, and practical) of NVSRAMCache designs have been proposed and they differ in how to achieve crash consistency and associated hardware complexity.

The original NVSRAMCache (full) [41] checkpoints the "entire" SRAM cache to the NVM counterpart, while the optimized NVSRAMCache (ideal) [16] reduces checkpointing overhead by magically copying "dirty" SRAM cache states only without requiring any additional support. However, note that because all cache lines could be dirty in the worst case, NVSRAMCache (ideal) still needs to reserve the same amount of large energy, enough to JIT checkpoint the entire cache. Moreover, the NVM part is unnecessarily large and mostly wasted because it is used only for checkpointing; therefore, these two versions are impractical. Table 1 list the high energy buffer requirement and the HW cost as their main downside.

NVSRAMCache (practical) [72, 73] integrates SRAM and NVM cache designs by maintaining SRAM cache lines and NV cache lines in the same cache set. At runtime, it migrates SRAM cache line to NV lines (if available). Upon power failure, NVSRAMCache (practical) uses JIT checkpointing to move the (remaining) dirty SRAM lines to NV lines, *i.e.,* it should ensure that there are enough available NV lines for JIT checkpointing at all times. Therefore, it writes-back dirty NV lines to NVM main memory at runtime, introducing additional traffic to NVM main memory. Besides, in NVSRAMCache (practical), a data may reside in NV lines, yet accessing NV cache lines is slower and consumes more energy than accessing SRAM lines (as in NVCache). Consequently, the performance improvement of NVSRAMCache (practical) is smaller than that the other designs (Table 1). Though NVSRAMCache (practical) is claimed to be more practical, NVSRAMCache (ideal) can achieve better performance, so we later compare our proposed scheme with NVSRAMCache (ideal). We omit "(ideal)" from now on.

3 DESIGN OF WL-CACHE

3.1 Overview

WL-Cache is designed on top of a traditional SRAM-based cache with a write-back policy that holds dirty cache lines and avoids NV main memory accesses on every store. However, unlike a traditional write-back cache, WL-Cache limits the possible number of dirty lines at a moment, so that it can failure-atomically flush the bounded number of dirty lines to NVM before power failure by leveraging the JIT checkpointing with a small energy reservoir. For instance, Figure 1(e) shows the case in which WL-Cache allows up to two dirty lines in a cache. By bounding the maximum number of dirty lines, WL-Cache achieves crash consistency without requiring expensive hardware support such as a large energy buffer or an NVM cache counterpart.

WL-Cache tracks dirty cache lines with a small hardware component, called DirtyQueue. When a cache line becomes dirty, WL-Cache inserts its memory address in DirtyQueue. The data remains in the cache (as dirty) and does not become immediately persisted in NVM. When DirtyQueue is about to be full, WL-Cache selects one of the dirty lines and asynchronously writes it back to NVM—though it may not be in the LRU position WL-Cache does not evict the line but leave the data in the cache as clean. When the asynchronous write-back finishes, WL-Cache removes the entry from DirtyQueue to serve later stores.

To manage DirtyQueue, WL-Cache employs two configurable thresholds, i.e., maxline and waterline. The maxline threshold ($\leq$ |DirtyQueue|) defines the maximum number of dirty cache lines in WL-Cache. When the number of dirty cache lines reaches the maxline, WL-Cache stalls a store instruction until a free slot becomes available. In other words, the maxline determines and bounds the amount of energy that WL-Cache needs to secure for checkpointing dirty cache lines to NVM upon a power failure. WL-Cache is more energy-efficient than an alternative NVSRAMCache (ideal) that should reserve a larger amount of energy enough to flush all dirty cache states, *e.g.,* for the worst case that every line is dirty. Initially, the maxline is set to be reasonable number (*e.g.,* maxline = 4) by considering the energy availability of a given energy buffer.

The waterline threshold ($\leq$ maxline) determines when WL-Cache starts writing back a dirty cache line to NVM during a program execution. When the number of dirty lines exceeds waterline, WL-Cache picks a dirty cache line, based on the DirtyQueue replacement policy (§5.3), and asynchronously writes it back to NVM. Note that WL-Cache does not evict a dirty line from the cache (which is separately done by a conventional cache replacement policy). Instead, the persisted (written-back) cache line remains in the cache in a "clean" state for future references; the address of the clean cache line is just removed from DirtyQueue only. WL-Cache exploits instruction level parallelism (ILP) by overlapping the asynchronous write-back operations with the executions of the following instructions.

The gap between maxline and waterline defines the potential ILP opportunity. A high waterline would keep more dirty cachelines in a cache, and should allow WL-Cache to serve more subsequent write hits without traveling to NVM, saving energy and improving performance. However, at the same time, a higher waterline has a risk to stall the following store instructions in case where WL-Cache cannot effectively hide the write-back latency (*e.g.,* a code region with frequent/dense stores). By default, waterline is set to be maxline − 1. So WL-Cache cleans (persists) one cache line at a time. The default setting attempts to make DirtyQueue at least one slot available so that a new dirty line can be added in DirtyQueue with no stall.

Conceptually, one can view WL-Cache with a cache-size maxline (waterline) as a traditional write-back cache; WL-Cache with a zero maxline (waterline) as a write-through cache. WL-Cache interacts with the runtime system that reconfigures the maxline and waterline threshold and thus simultaneously behave as a write-back, write-through, and somewhere between them, depending on the quality/stability of power sources. WL-Cache in effect provides a tuning knob to make the best of two worlds; this will be discussed in §4.

Figure 2 illustrates a running example of WL-Cache. Supposedly, DirtyQueue's maxline is 2 and waterline is 1 in this example. Assumes that the system has enough energy to execute a given program on the top that consists of three store instructions and two arithmetic instructions in between. The first two store instructions introduces two new dirty cache lines and their addresses (0x10000 and 0x20000) are maintained in DirtyQueue as shown in Figure 2 (a) and (b). After the second store, the number of dirty lines exceeds waterline. In response, WL-Cache picks one of dirty cache lines and asynchronously makes it persisted and clean (without eviction). If WL-Cache relies on FIFO-based DirtyQueue replacement policy, WL-Cache writes back the oldest line (0x10000) in DirtyQueue which is mapped to 0x1 tag address in a cache (Figure 2 (c)). In the meantime, a program can make a progress and execute the next ADD and SUB instructions, exploiting ILP. When the asynchronous write-back operation is completed (with the ACK message), WL-Cache removes the corresponding entry from DirtyQueue (Figure 2 (d)). With an empty slot in DirtyQueue, the third store instruction can be served without a stall. If the number of dirty lines reaches maxline (which did not happen in this example), WL-Cache stalls the store instruction and bounds the total number of dirty lines.

3.2 Crash Consistency with WL-Cache

WL-Cache ensures crash consistency using the following checkpointing and recovery protocols. When a voltage drops below the threshold V_{backup}, the voltage monitor signals the processor to checkpoint volatile registers (as in NVP) and volatile dirty cache lines (in DirtyQueue) to the NVM space. In particular, WL-Cache sets the JIT checkpointing voltage threshold (V_{backup}) accordingly to persist the maxline number of cache lines at each reboot time. While WL-Cache needs to reserve more energy than NVP (without a volatile cache) for dirty cache line checkpointing, WL-Cache's caching benefits are expected to be much higher (as will be shown in our evaluation (§6.3)). During JIT checkpointing, WL-Cache identifies the dirty cache lines based on the memory addresses stored in DirtyQueue using the existing cache lookup control/data path. Then, WL-Cache writes them back to the NVM using the existing cache-memory data path. Note that WL-Cache neither allows more dirty cachelines than it can handle (i.e., a maxline threshold) nor overspends the energy budget for JIT-checkpointing as many dirty lines as the threshold. Once the maxline threshold is (re)configured, WL-Cache adjusts the V_{backup} voltage accordingly at boot time. This backup energy is set aside for JIT-checkpointing, and thus any dirty line hitting the threshold would be flushed—asynchronously—using additional energy,

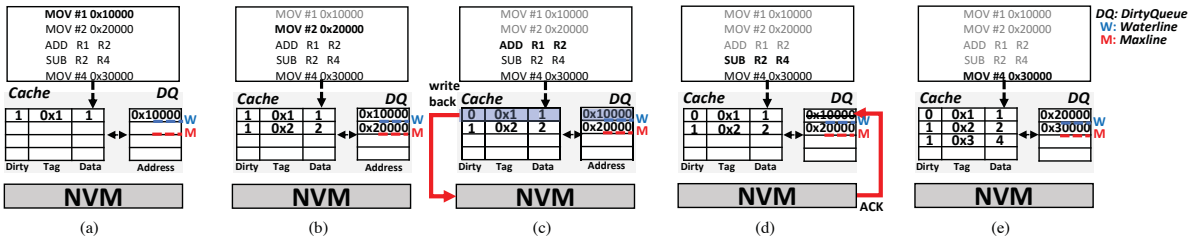


Figure 2: Running example of WL-Cache. WL-Cache holds dirty cache lines and keeps track of their memory addresses in DirtyQueue (DQ). When the number of dirty lines exceeds waterline (blue dashed line), WL-Cache asynchronously writes back a dirty line to NVM while a processor executes the next instructions. When the number of dirty lines reaches maxline (red dashed line), WL-Cache stalls the store instruction, bounding the total number of dirty lines in WL-Cache.

not consuming the backup energy. In this way, whether program runs longer or shorter, WL-Cache always secures a sufficient amount of energy to checkpoint the maxline number of dirty cachelines during the program execution.

The recovery protocol after power becomes available again remains simple and the same as that of existing energy harvesting systems. When the capacitor becomes full, energy harvesting systems restore the register states (including instruction pointer) from NVFF for NVP (or from NVM for QuickRecall [20]).

3.3 Discussion

We found that a WTCache with a large write-back buffer can also behave like WL-Cache. However, the alternative design would be inferior to WL-Cache for three issues. First, the design increases HW cost since the large write-back buffer must be backed with content-addressable memory (CAM) search; one might think of a small buffer to reduce CAM search cost, but it would lead to a worse problem, i.e., frequent NVM writes and pipeline stalls. Second, the design is energy-inefficient since the large buffer requires a significant amount of energy to be secured for crash consistency (failure-atomic write-back before outages). Third, the design would suffer performance degradation by extending the critical path of memory access since the write-back buffer must be consulted before accessing memory, i.e., cache miss latency is lengthened.

The key architectural innovation is that WL-Cache decouples the metadata (which cachelines are dirty) from the actual cacheline data. That way WL-Cache does not increase the critical path of memory access thanks to the lack of metadata (DirtyQueue) lookup, e.g., load miss latency remains the same. This enables WL-Cache to achieve a lightweight cache along with its adaptation to varying energy harvesting conditions. Not only that, the decoupled design makes it possible for WL-Cache to realize the DirtyQueue as a volatile structure without compromising the crash consistency guarantee.

4 ADAPTIVE MAXLINE MANAGEMENT

WL-Cache interacts with a runtime system (a system software) that reconfigures maxline and waterline in DirtyQueue. Energy harvesting systems stores energy in the capacitor so the recharging time (power-off time) directly depends on the quality of energy source. However, the power-off time is hard to measure. Instead, the WL-Cache runtime system measures a "power-on time" using a watch-dog timer to estimate the quality of energy source. Note that energy harvesting systems continuously collect energy as they execute (during power-on as well). Though every on-interval starts from the same level of energy (at the same V_{on} voltage level of the capacitor), when the harvesting condition is good, the system may run longer.

Based on the observation, WL-Cache runtime system estimates the energy source quality from the past power-on times, and adjusts the maxline and waterline thresholds at each boot time. Once set, they remain the same during execution (until energy drains). Changing the thresholds while running could be dangerous as energy harvesting systems may not be able to guarantee JIT checkpointing.

For adaptive management, WL-Cache compares the power-on times of the last two intervals (T_{n-2} and T_{n-1}) to determine the maxline and waterline thresholds of the next interval (maxline$_n$ and waterline$_n$). If the measured power-on time increases significantly, it implies that the energy source quality is good, and thus the system adaptively raises maxline and waterline. With the higher maxline, WL-Cache attempts to take more advantage of locality like a write-back cache. In contrast, if the power-on time decreases, implying a poor energy source condition, the system lowers maxline and waterline because it is better to avoid a large voltage margin. Otherwise, the two thresholds remain the same. Once the maxline is determined, the runtime system also need to adjust the voltage margin V_{backup} large enough to JIT-checkpoint the maxline number of dirty cache lines at boot time.

Figure 3 describes an example execution in which the maxline and waterline are reconfigured. Supposedly, at the beginning of a program execution, the maxline and waterline in DirtyQueue are 3 and 2, respectively. When the capacitor voltage level reaches V_{on}, the system boots on (the first boot) and runs. When the voltage level becomes below V_{backup}, the system initiates JIT checkpointing of volatile registers and (maxline) dirty cache lines. The system also stores its power-on time for the first interval (T_1). Suppose now the energy source condition becomes much better. The system recharges energy quickly and it makes more forward progress. Upon the third boot, the system detects that the power-on time of the second interval (T_2) becomes much longer than that of the first interval (T_1). It increases the maxline and waterline thresholds (4 and 3, respectively) as well as the voltage margin V_{backup} accordingly. The reconfiguration allows the system to hold more dirty cache lines during the execution of the third interval, providing more opportunity to exploit locality. There is no big change in the power-on times of

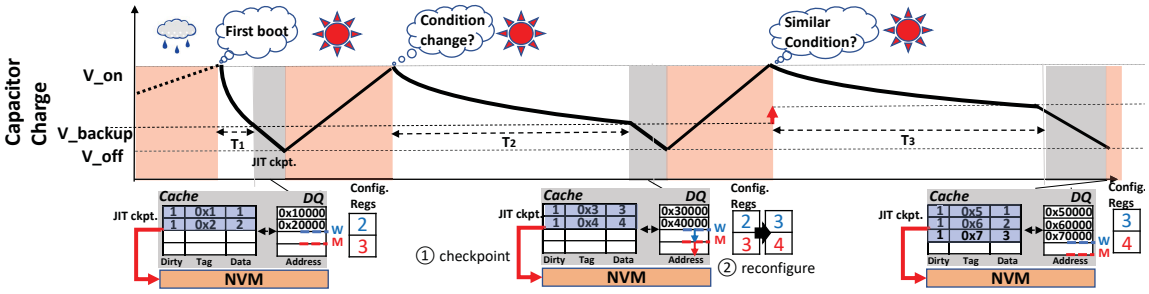


Figure 3: An example execution with adaptive maxline, waterline, and V_{backup}. The red and white intervals represents power-off and power-on periods, respectively. The system boots and runs when the charge reaches V_{on}, and starts JIT-checkpointing (gray interval) when it becomes below V_{backup}. T_n represents the power-on time of n-th interval.

the second and third intervals (T_2 and T_3), so the thresholds remain the same. When the power-on time drops later, the system may adaptively decrease the thresholds accordingly.

Dynamic adaptation: When the energy source is strong (e.g., solar or thermal) without causing frequent power failure, WL-Cache may not fully exploit the adaptive maxline management since the adaptation is done statically at boot time—though we target RF-based energy harvesting systems as in other NVP proposals [16, 41, 71–73, 75]. To address this issue, WL-Cache can dynamically raise V_{backup} as the number of dirty cache lines increases. Note that WL-Cache performs such dynamic adaptation opportunistically, i.e., if the capacitor energy cannot afford a new line, we would rather write back one of the dirty lines tracked by DirtyQueue than stall the pipeline to charge the capacitor. For example, when the number of dirty lines reaches the maxline, WL-Cache checks the residual energy in the capacitor. If the energy is sufficient to JIT-checkpoint another line, WL-Cache increases the maxline by 1 while raising V_{backup} accordingly. We found that the dynamic adaptation works better than the static adaptation when the energy source is strong; this will be discussed in §6.

5 WL-CACHE POLICY AND PROTOCOLS

5.1 DirtyQueue Insertion Protocol

When a cache line becomes dirty,WL-Cache first checks if there is an empty space in DirtyQueue by comparing the maxline and the number of dirty lines (a tail of the circular queue). If available, the corresponding store address is appended in DirtyQueue. Otherwise, the store instruction stalls until an empty slot becomes available. The subsequent store instructions to the same dirty line (*i.e.,* no dirty state change) does not trigger an interaction between a cache and DirtyQueue. Note that, thanks to the waterline constraint, WL-Cache leaves an at least one space empty (§4); therefore, the stall rarely occurs.

5.2 DirtyQueue Replacement Policy

When the number of total dirty cache lines exceeds the waterline, WL-Cache selects a dirty line to write back in NVM. We refer to the decision process as a *"DirtyQueue replacement policy"* to differentiate it from a traditional cache replacement policy since WL-Cache does not evict it but cleans it. WL-Cache supports FIFO and LRU replacement policies.

5.3 DirtyQueue Replacement Protocol

After a dirty line is selected according to the above DirtyQueue replacement policy, WL-Cache asynchronously write-backs the dirty cache line using the following four steps. (1) WL-Cache marks the cache line clean without eviction, (2) WL-Cache sends an asynchronous write-back request, (3) WL-Cache waits for the delivery of an ACK message (acknowledging the completion of the write-back operation). In the meantime, a processor executes the subsequent instructions, and (4) upon ACK, WL-Cache removes the associated slot from DirtyQueue. For FIFO policy, it will be always the head. The LRU-based scheme requires search.

Figure 2 (c) illustrates the first and the second steps in which the cache line of Tag 0x1 (Address 0x10000) becomes clean then persisted in NVM. At the moment, the processor executes ADD instruction (exploiting ILP). Figure 2 (d) shows the next two steps where the ACK message is delivered and the DQ entry (head, Address 0x10000) is removed.

Removing the associated entry in DirtyQueue last (Step 4) ensures that if a power fails any step before Step 4, WL-Cache can still safely find the entry in DirtyQueue and perform JIT-checkpointing of the dirty cache line, regardless of whether the write-back request (Step 2) has been fulfilled or not. If completed, WL-Cache may (redundantly) write back the cache line again, yet there is no correctness issue. If write-back is not done, WL-Cache will persist the cache line, making the NVM state consistent across a power failure.

Marking the cache line clean first (Step 1) ensures the correctness even when the store instruction to the same cache line is executed while the cache line is being written back asynchronously (Step 3). Suppose we have two stores to the same memory location X, say $WX = 1$ and $WX = 2$, and some other instructions in between. The first store $WX = 1$ makes the cache line $X = 1$ and dirty. Let's say DirtyQueue becomes full (by other store instructions) and the cache line X is selected to be cleaned. The protocol first marks the cache line clean (Step 1) and then start a write-back operation (Step 2). Suppose at the moment, the second store $WX = 2$ performs. Now we demonstrate what could go wrong if the cache line is not marked clean (not doing Step 1 first).

If the cache line remains dirty, the second store will not find a state transition to dirty, so it will not update DirtyQueue. The second store will make the cache line $X = 2$ and dirty. Suppose the write-back operation of the first store finishes (NVM now has $X = 1$), and the address X is removed from DirtyQueue (Step 3

and 4). Then, the power is out while the cache has $X = 2$. This is problematic as the cache has $X = 2$ but NVM has $X = 1$ (inconsistent) and DirtyQueue does not have the recent X, losing the cache state $X = 2$. To avoid the problem, WL-Cache marks the cache line clean first (Step 1), so that the second store will also add the address X in DirtyQueue. As this may happen rarely, WL-Cache allows DirtyQueue to temporarily hold redundant X (wasting the slots in DirtyQueue), instead of actively searching the redundant entry in DirtyQueue with additional hardware logic. A redundant entry in DirtyQueue does not affect the correctness as it may only cause redundant write-back operations.

5.4 Cache Replacement Policy

WL-Cache can rely on a traditional LRU cache placement policy that determines which cache line to evict on a cache miss. Yet, the cache replacement policy may conflict with the DirtyQueue replacement policy. When the cache replacement policy decides to evict a dirty cache line whose address is in DirtyQueue, the cache line becomes invalid after a write-back operation (*i.e.,* after it becomes persisted). Thus, it would be ideal to remove the associated entry in DirtyQueue upon an eviction so that DirtyQueue can become more available to other stores. However, this eager cleanup requires searching DirtyQueue to find the entry on each eviction, increasing the latency and hardware complexity. To avoid search, WL-Cache instead chooses not to remove it eagerly and allows an outdated slot to reside in DirtyQueue temporarily. When an DirtyQueue entry is selected for replacement or JIT-checkpointing, WL-Cache can find the cache line invalid (or does not exist) and safely ignore it. Moreover, we empirically found that the traditional LRU cache replacement policy is energy-inefficient for energy harvesting systems. Since it tracks LRU/MRU list at every memory access, it consumes more power and increases more latency than FIFO cache replacement policy; this will be discussed in §6.6.

5.5 DirtyQueue Threshold Management

WL-Cache introduces a small hardware space (1 byte each) to hold the maxline and waterline thresholds of DirtyQueue. As they have to be alive across a power failure, WL-Cache introduces NVFF-based backup and performs JIT checkpointing (similar to volatile registers). For adaptive threshold management (§4), WL-Cache adds a watchdog timer and two NVFF-based non-volatile storage (2 bytes each) to keep the last two past power-on times. When the power backs on, all these values are restored from NVFF.

At boot time, given maxline, WL-Cache also needs to adjust the voltage margin V_{backup} to ensure the failure-atomic JIT checkpointing of (1) registers, (2) (up to maxline) dirty cache lines, and (3) maxline, waterline, and power-on timer values. To reconfigure V_{backup}, WL-Cache assumes existing hardware support in current commodity micro-controller such as TI-MSP430 [2] that already support different voltage selections. WL-Cache sets V_{backup} by choosing an associated voltage divider with a reference voltage.

6 EVALUATION

6.1 Experimental Settings

We implemented WL-Cache on gem5 [7] with ARM ISA, modeling a single core in-order processor. For power failure simulation, we used NVPsim [16] with the same core model. We compared WL-Cache to (1) non-volatile write-back cache (NVCache-WB) [6, 18], (2) volatile write-through SRAM-based cache (VCache-WT), (3) VCache-WB with ReplayCache compiler (ReplayCache) [75], and (4) the state-of-the-art NVSRAM cache that uses ReRAM as the SRAM cache backup storage, with a write-back policy (NVSRAM-WB) [65]; we set this as an ideal design that can checkpoint only dirty lines from SRAM to ReRAM at power-off point. We used Mediabench [31] and MiBench [17] compiled with -O3 optimization.

Processor (1.0GHz, 1 core)	
L1 I/D Cache	8kB, 2-way, 64B block
Cache Latencies (hit/miss)	NVRAM(1.6ns/1.5ns), SRAM(0.3ns/0.1ns)
NVM (ReRAM) Latency (ns)	0.94/7.5/18/15/7.5/150/30 (tCK/tBURST/tRCD/tCL/tWTR/tWR/tXAW)
Energy buffer (capacitor)	1uF [16, 45, 69]
Vbackup/restore	NV(2.9/3.3), NVSRAM(3.1/3.5), WL(2.95∼3.1/3.3∼3.5)
Vmin/max	NV(2.8/3.5), NVSRAM(2.8/3.5), WL(2.8/3.5)

Table 2: Simulation configuration.

As a default configuration, we use volatile L1 instruction and data caches as with the prior work [41, 75] (see Table 2). For WL-Cache, we use the FIFO for DirtyQueue replacement policy and LRU for cache replacement policy as default; we varied the policies for sensitivity analysis (§6.5). Also, we set the DirtyQueue size to 8 and the maxline to 6 as default (*i.e.,* the waterline is 5), then we enable the adaptive threshold management (§4) to reconfigure maxline and waterline, accordingly.

To evaluate WL-Cache in realistic energy harvesting situations, we used the same two power traces, *i.e.,* Trace 1 and Trace 2, of the NVPsim which were collected from real RF sources [16, 75]; Trace 1 and 2 are from home and office, respectively. Trace 2 is relatively less stable than Trace 1.

6.2 Hardware Cost

We analyzed the hardware cost of WL-Cache by using CACTI [62] with 90 nm technology. WL-Cache requires at most 0.005 mm^2 area and 0.0008 nJ (dynamic access). Furthermore, the total leakage power of WL-Cache (DirtyQueue with a logic) causes only 0.1mW in total, which is only 9% of NV cache leakage [16, 73, 75].

6.3 Performance Analysis

Performance Analysis without Power Outages: For performance analysis, we set the baseline to be NVSRAM-WB [65] and compared with NVCache-WB [6, 18], VCache-WT, ReplayCache with VCache-WB [75], and WL-Cache. Figure 4 shows the speedup when there is no power failure. Overall, WL-Cache shows a similar speedup to NVSRAM-WB, achieving a 3.1x speedup on average.

The baseline NVSRAM-WB is the fastest cache design for Non-volatile processor (NVP) [75]. On the other hand, NVCache-WB is the slowest as it has to pay long latency for every nonvolatile cache access. VCache-WT could take advantage of (SRAM-based) fast cache hits. ReplayCache improves the performance since it overlaps the next instruction execution with NVM stores of the same program region without waiting the corresponded ACK like ILP; it persists all stores at region-level granularity. Thanks to the region-level persistence with ILP execution, it can achieve almost 60% speedup compared to VCache-WT. On the other hand, NVSRAM-WB shows the best performance among all designs, demonstrating

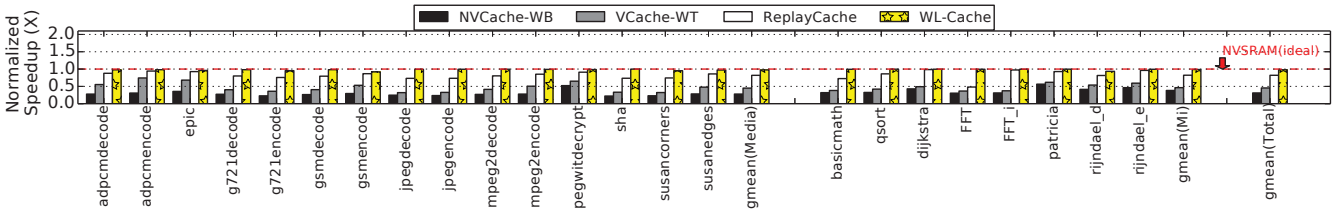


Figure 4: Normalized speedup of each cache design compared to NVSRAM(ideal) with no power failure.

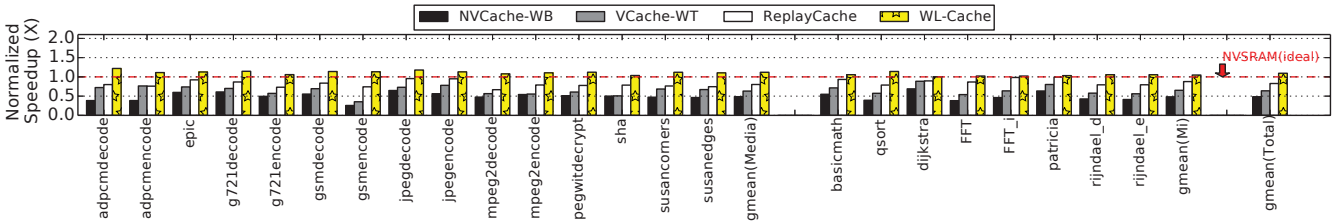


Figure 5: Normalized speedup of each cache design compared to NVSRAM(ideal) in Power Trace 1.

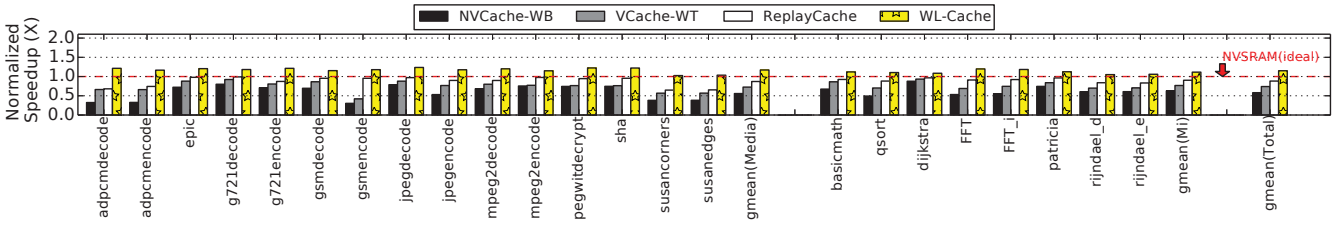


Figure 6: Normalized speedup of each cache design compared to NVSRAM(ideal) in Power Trace 2.

the benefit of write-back. WL-Cache was slower than NVSRAM-WB, implying that WL-Cache could hold enough dirty cache lines. With waterline-based write-back, WL-Cache did not suffer much from potential stalls. WL-Cache also effectively hide the cost of asynchronous write-back operation, exploiting ILP.

Performance Analysis with Power Outages: Figure 5 and 6 show the speedup of each cache design using Power Trace 1 and 2, respectively. We took into account both power-on and power-off periods in this experiment.

For all applications, WL-Cache shows the best performance among all designs. WL-Cache achieves on average about 1.09x and 1.12x speedup compared to the baseline in Trace 1 and 2, respectively. WL-Cache is 225% and 198% faster than NVCache-WB, 71% and 55% faster than VCache-WT, and 32% and 30% faster than Replay-Cache in Trace 1 and 2, respectively, demonstrating the benefits of holding dirty cache lines and exploiting cache locality. Upon a power failure, WL-Cache needs to persist a bounded number of dirty cache lines, whereas NVCache-WB should reserve more energy to support cache backup. With less energy reserved for JIT checkpointing, WL-Cache could efficiently use the energy to compute and make a further forward progress.

Write Traffic with Power Outages: Figure 7 shows the write traffic overhead of WL-Cache compared to NVSRAM-WB cache using Trace 1. The result demonstrates that WL-Cache slightly increases

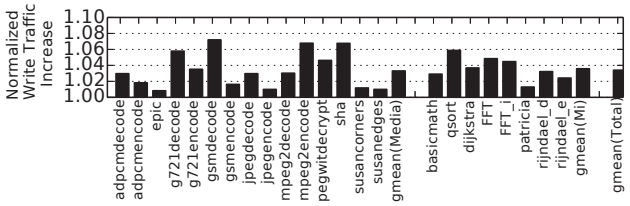
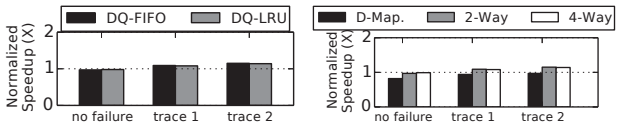


Figure 7: Normalized write traffic increase compared to NVSRAM(ideal) in Power Trace 1.



(a) DirtyQueue replacement. **(b) Cache set associativity**

Figure 8: Normalized speedup of WL-Cache with different DirtyQueue replacement and different cache set associativity compared to NVSRAM(ideal) on average.

the write traffic; however, it can be paid off by enabling asynchronous write back and adaptive execution as proven in Figure 5.

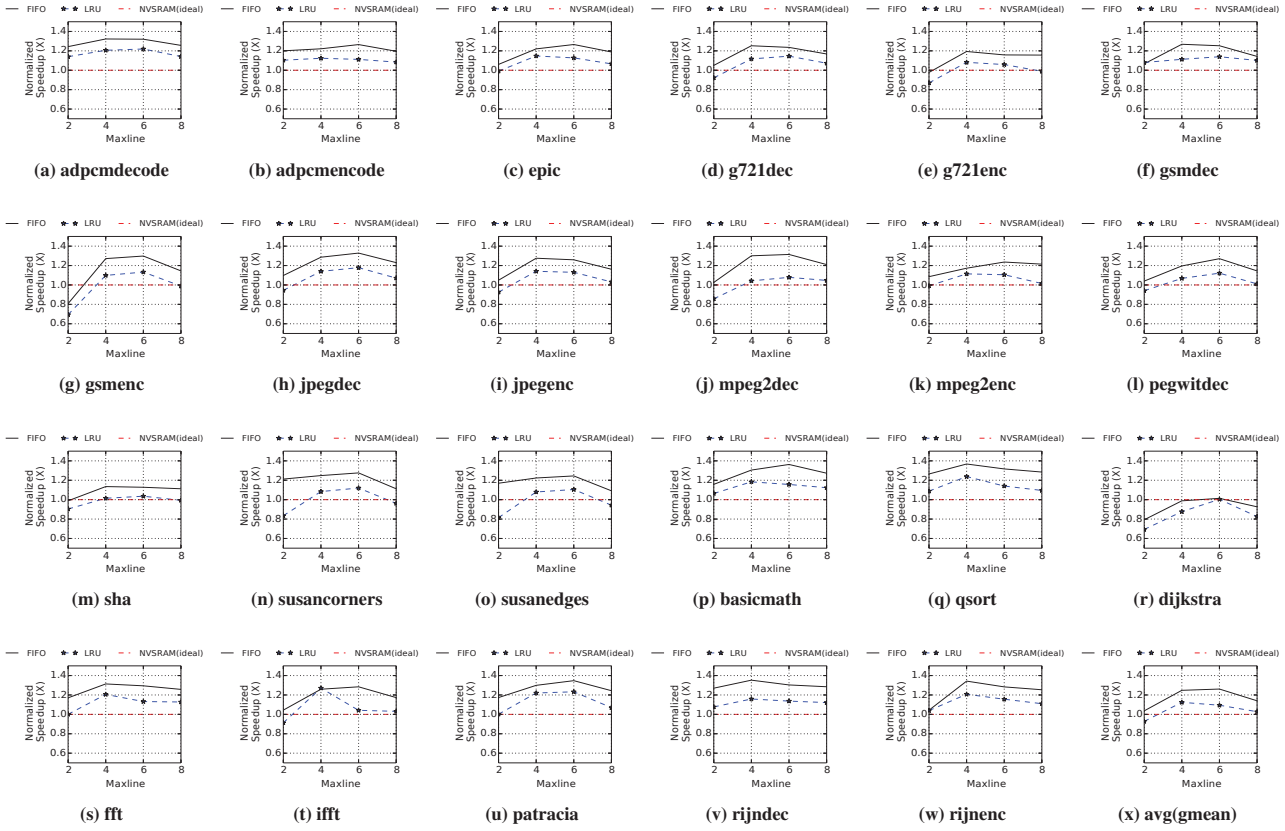
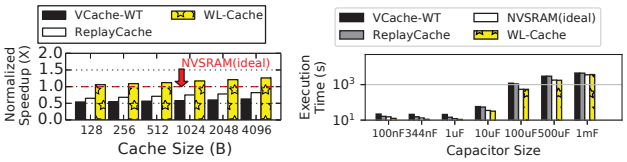


Figure 9: Sensitivity analysis on applications varying maxline sizes and cache replacement policies in Power Trace 1.



(a) Cache size variation. **(b) Capacitor size variation.**

Figure 10: Normalized speedup of each cache design in Power Trace 1 varying (a) each cache size and (b) capacitor size

6.4 DirtyQueue Replacement Policy

We measured performance of WL-Cache by varying the DirtyQueue replacement policies. WL-Cache with DirtyQueue-FIFO (DQ-FIFO) shows slightly higher performance than WL-Cache with DirtyQueue-LRU under power failures as shown in Figure 8(a). Because WL-Cache does not evict the cache line and keeps it in the cache as clean regardless of DirtyQueue replacement policy, most subsequent memory references would show similar cache hit/miss trends. In general, LRU is known to be better than FIFO but it turns out that the additional power consumption for the LRU lookup logic often offsets the potential benefits of LRU, and the cache is often not warm enough to see the visible merits of LRU in energy harvesting systems. In particular, the LRU cleaning policy is slightly slower than FIFO in Trace 1 and 2, due to additional power consumption for that LRU lookup logic. WL-Cache uses the FIFO-based DirtyQueue replacement policy by default.

6.5 Sensitivity Analysis

We conducted sensitivity analysis on WL-Cache by varying the maxline size, cache replacement policy (LRU and FIFO), set associativity, cache size, capacitor size, power trace, and adaptive mechanism. Notably, we explore cache replacement policy in this section (with FIFO-based DirtyQueue replacement policy). For analysis, we used the same baseline (i.e., NVSRAM-WB) and measured the performance of each cache design using Trace 1. Figure 9 shows the results.

Set-associative Cache Variation: We varied cache associativity and measured the execution time of each design. Figure 8(b) shows that the direct-mapped cache is the slowest design; it is slower than the baseline. On the other hand, 2-way and 4-way set-associative cache design show similar performance. In particular, the 4-way set-associative cache is slightly slower than 2-way set-associative cache in Trace 1 and 2, due to additional power consumption. WL-Cache uses the 2-way set-associative cache by default.

Maxline Variation: We investigate how its performance varies when the maxline changes from 2 to 8 along with different replacement policies. WL-Cache shows good performance with maxline 4 or 6. The performance differences between maxline 4 and 6 are not

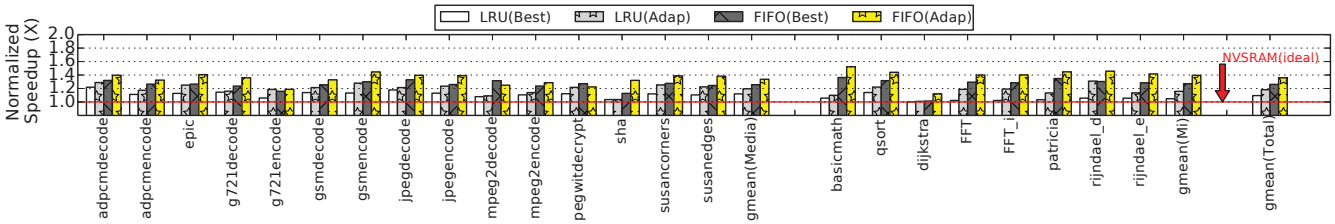


Figure 11: Normalized speedup of WL-Cache with adaptive management compared to NVSRAM(ideal) in Power Trace 1.

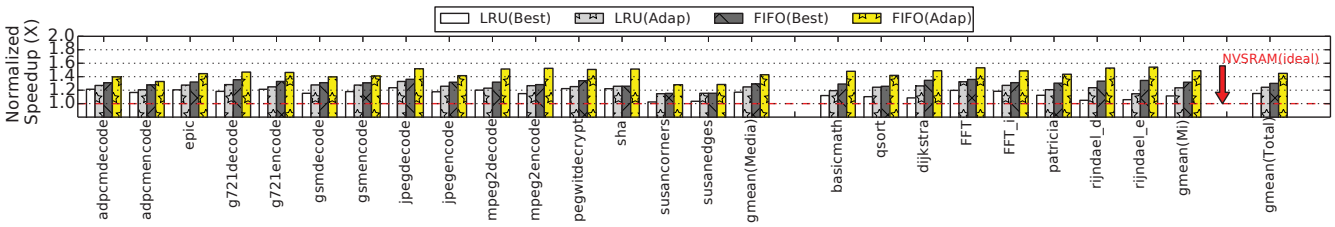
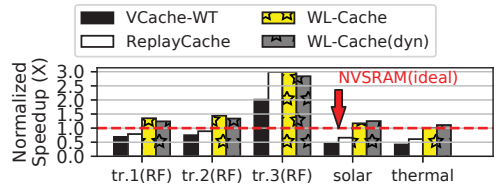
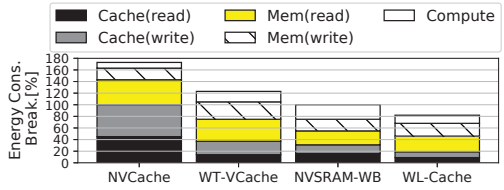


Figure 12: Normalized speedup of WL-Cache with adaptive management compared to NVSRAM(ideal) in Power Trace 2.



(a) Performance comparison across power traces.



(b) Energy consumption breakdown with Power Trace 1.

Figure 13: Performance analysis across power traces (a) and energy consumption breakdown using Power Trace 1 (b).

significant. With too large maxline (*e.g.,* 8), performance degrades as WL-Cache has to reserve more energy for JIT checkpointing. With too small maxline (*e.g.,* 2), performance also degrades as WL-Cache does not take advantage of dirty cache lines and locality.

Cache Replacement Variation: Figure 9 shows that FIFO-based WL-Cache (black line) outperforms LRU-based WL-Cache (blue dotted line) for cache replacement policies. When compared to NVSRAM-WB (red dotted line), FIFO-based WL-Cache performs better for almost all applications in all the maxline settings. For general purpose systems, LRU normally performs better compared to FIFO. However, for energy harvesting systems with frequent power outages, we found FIFO is always faster than LRU. Further investigation reveals two main reasons for this surprising result.

First, the impact of cache replacement policy on cache miss rate is limited. We found that both policies cause almost the same cache miss rates. Energy harvesting systems experience frequent power failures, and they wake up with a "cold" cache—that has no data—across power failure. Therefore, WL-Cache is likely to cause compulsory (cold) misses across power failure regardless of the replacement policy. Furthermore, the systems run applications for a short amount of time. Within the limited time, the room for a smart cache replacement policy to address other conflict misses is simply small.

Second, the FIFO policy is energy-efficient without requiring additional cost to track LRU/MRU list at every memory access unlike LRU, that takes more latency and consumes more power possibly causing more power outages; we found LRU caused more power outages for most of applications.

Cache Size Variation: Figure 10(a) illustrates that the normalized performance speedup of alternative cache schemes with a different cache size from 128B to 4KB using Power Trace 1. The results indicate that the performance gap between WL-Cache and NVSRAM-WB is decreased when the cache size is decreased, and vice versa; the speedup is also increased as the cache size increases.

Capacitor Size Variation: For capacitor size sensitivity analysis, we used NVSRAM(ideal) with 1uF as baseline, and measured the performance of alternative cache design with a different capacitor size from 100nF to 1mF using Power Trace 1. Figure 10(b) describes the normalized performance speedup of alternative cache schemes. In particular, all schemes show their best performance when the capacitor size is 1uF. However, when the capacitor size is increased more than 1uF, their performance is exponentially decreased. This is mainly because their charging time is increased when the capacitor size is increased. On the other hand, the performance gap between WL-Cache and NVSRAM-WB is decreased when the capacitor size is increased. Nevertheless, it would be a mistake to take the diminishing gain to mean that such a large capacitor is the norm because the best performance is achieved with 1uF capacitor where WL-Cache

significantly outperforms NVSRAM-WB achieving 1.35x and 1.44x average speedups for Trace 1 and Trace 2, respectively.

6.6 Maxline/Waterline Threshold Adaptation

The above section motivates WL-Cache's adaptive maxline (and waterline) management optimization (§4). For comparison, we measured the performance gain of "adaptive" WL-Cache with FIFO and LRU replacement policies and compared them to "static" WL-Cache. For the fixed settings, we picked the best performing maxline size for each application as we found in Figure 9. Figures 11 and 12 show the results for Trace 1 and 2, respectively. Adaptive WL-Cache outperforms static-Best WL-Cache for both FIFO- and LRU- cache replacement schemes. Overall, for Trace 1, FIFO (Adap) and LRU (Adap) achieve 1.35x and 1.18x speedups, respectively, while FIFO (Best) and LRU (Best) do 1.26x and 1.1x speedups, respectively. For power trace 2, FIFO (Adap/Best) and LRU (Adap/Best) earn speedups of 1.44x/1.3x and 1.24x/1.15x, respectively.

On average, WL-Cache reconfigures the maxline (and waterline) thresholds 11 and 12 times on trace 1 and 2, respectively. The minimum and maximum values of maxline are 2 and 6 for both traces. The energy source prediction accuracy is >98%, and thus the impact of mis-prediction is minimal in both traces. With adaptive threshold management, we also measured the number of dirty lines and the number of write-backs during each power-on period on average, which are 6/3 and 6/2 (dirty-lines/write-backs) in trace 1 and 2, respectively. In addition, the pipeline stall causes a negligible delay, <1% of the total execution time on average in both traces.

Power Trace Sensitivity: We conducted additional experiments with another RF trace (tr.3 [57]) as well as solar and thermal traces [16]. Figure 13(a) shows that WL-Cache outperforms all others significantly for all RF traces yet achieves only 8% and 2% speedups over NVSRAM (ideal) for solar and thermal traces, respectively. This is because NVSRAM works better in stable environment such as solar and thermal (as shown in Figure 4); the numbers of power outages in tr.1/tr.2/tr.3/solar/thermal are 33/45/121/12/9, respectively, on average during the entire program execution. In particular, such frequent outages in tr.3 make NVSRAM perform the worst among all tested cache designs.

Adaptation Sensitivity: We also tested the dynamic adaptation scheme (§4) shown as WL-Cache (dyn) in Figure 13(a); it outperforms WL-Cache by about 5% and 3% for solar and thermal traces, respectively. This is because by raising the maxline during program execution, WL-Cache (dyn) can reduce the number of writebacks—especially when the following stores fall into the lines of DirtyQueue which would otherwise be written back in NVM due to the waterline constraint. However, it is slower than WL-Cache for all RF traces that are less stable than others; it turns out that even if WL-Cache (dyn) manages to increase the maxline with V_{backup} raised, power failure often occurs before the writeback reduction effect kicks in. In the presence of unstable power, such premature V_{backup} raise leaves a less amount energy for computation at run time and delays the booting time without much gain, thus ending up with more failure and performance degradation compared to WL-Cache.

6.7 Energy Consumption Analysis

We measured energy consumption of WL-Cache and compared it to other cache designs with power outages for each part of the system: core (computation), cache, and main memory (NVM). For analysis, we set WL-Cache with FIFO replacement policy and adaptive maxline management optimization. Figure 13(b) shows the breakdown, normalized to the same NVSRAM(ideal) baseline on average, using Trace 1. Overall, WL-Cache reduces the total energy consumption by almost 17% compared to the baseline. In particular, WL-Cache significantly reduces the energy consumption from a cache part, which is less than the other designs.

7 RELATED WORK

	Domain	When to write back?	Cost	WSP
Early-write[33]	Reliability	Eager (timer expiration)	New cache structure (per-line timer)	No
DPO [29] HOPS [52] BBB [4] ASAP [74]	NVM crash consistency	Eager/Write-through (L1 cache to NVM)	New data-path, SW support, Backup battery, Large buffer	No
VWQ [64]	DRAM performance	Eager (scheduled)	Additional hardware for writeback scheduling	No
VIPS/-m [25, 59]	Cache coherence	Eager (shared block) Lazy (private block)	TLB/OS interaction	No

Table 3: Related work comparison.

To improve cache performance, a prior work introduces an eager write-back cache [32] that opportunistically flushes lines when memory bus is idle. Unfortunately, the prior work was designed without considering neither crash consistency nor energy efficiency that are essential for energy harvesting systems. In addition, many prior studies have leveraged the eager write-back cache for various purposes, e.g., reliability enhancement [5, 27, 33, 60], NVM crash consistency guarantee [4, 29, 52, 74], and performance optimization [24, 25, 59, 64], as shown in Table 3. However, they are not applicable to energy harvesting systems that require high energy efficiency as well as whole system persistence (WSP) [23, 53]. Apart from the power-hungry hardware support, they require either persistent programming [29, 52] with a separated (persist) data-path and a large battery [4, 29, 74], customized TLB [25, 59], or additional cache structure [5, 27, 33, 60].

8 CONCLUSION

This paper presents WL-Cache, new cache organization with a hybrid write policy for energy harvesting systems. WL-Cache is neither write-through nor write-back caches. It combines the benefits of write-back caches (efficiency) and write-through caches (persistence) without their downsides. To achieve this, WL-Cache controls the maximum number of dirty cache lines that it manages to just-in-time checkpoint upon power failure; if the number gets bigger (smaller), WL-Cache acts more like a write-back (write-through) cache. For optimization, WL-Cache estimates the underlying energy harvesting quality by analyzing power-on times, and adjusts the number of manageable dirty cache lines correspondingly. Our evaluation highlights that WL-Cache outperforms all existing cache designs significantly.

ACKNOWLEDGMENTS

We thank anonymous reviewers for their comments. At Purdue, this work was supported by NSF grants 1750503 (CAREER) and 2153749. At Stony Brook, this work was supported by NSF grants 2153747, 2135157, and 2214980. At Virginia Tech, this work was supported by NSF grant 2153748.

REFERENCES

[1] 2015. Maximizing Write Speed on the MSP430™ FRAM. http://www.ti.com/mcu/docs/

[2] 2016. MSP430FR5994LaunchPad Development Kit (MSPEXP430FR5994). http://www.ti.com/lit/ug/slau678a/slau678a.pdf

[3] Khakim Akhunov and Kasim Sinan Yildirim. 2022. AdaMICA: Adaptive Multicore Intermittent Computing. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 6, 3 (2022), 1–30.

[4] Mohammad Alshboul, Prakash Ramrakhyani, William Wang, James Tuck, and Yan Solihin. 2021. Bbb: Simplifying persistent programming using battery-backed buffers. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 111–124.

[5] Ghazanfar-Hossein Asadi, Vilas Sridharan, Mehdi B Tahoori, and David Kaeli. 2005. Balancing performance and reliability in the memory hierarchy. In *IEEE International Symposium on Performance Analysis of Systems and Software, 2005. ISPASS 2005.* IEEE, 269–279.

[6] Mary Baker, Satoshi Asami, Etienne Deprit, John Ouseterhout, and Margo Seltzer. 1992. Non-volatile memory for fast, reliable file systems. *ACM SIGPLAN Notices* 27, 9 (1992), 10–22.

[7] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *SIGARCH Comput. Archit. News* 39, 2 (Aug. 2011).

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

[9] Jongouk Choi. 2022. *HIGH-PERFORMANCE AND RELIABLE INTERMITTENT COMPUTATION*. Ph. D. Dissertation. Purdue University Graduate School.

[10] 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* (2022).

[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, Larry Kittinger, Qingrui Liu, and Changhee Jung. 2022. Compiler-Directed High-Performance Intermittent Computation with Power Failure Immunity. In *Real-Time & Embedded Technology and Applications Symposium*.

[13] 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.

[14] Marc A. De Kruijf. 2012. *Compiler Construction of Idempotent Regions and Applications in Architecture Design*. Ph. D. Dissertation. Madison, WI, USA. Advisor(s) Sankaralingam, Karthikeyan.

[15] Jasper de Winkel, Carlo Delle Donne, Kasim Sinan Yildirim, Przemysław Pawełczak, and Josiah Hester. 2020. Reliable timekeeping for intermittent computing. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 53–67.

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

[17] 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 *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*. IEEE, 3–14.

[18] 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.

[19] Harishankar 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 *VLSI Design and 2014 13th International Conference on Embedded Systems, 2014 27th International Conference on*. IEEE, 330–335.

[20] 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.

[21] Jungi Jeong, Jaewan Hong, Seungryoul Maeng, Changhee Jung, and Youngjin Kwon. 2020. Unbounded hardware transactional memory for a hybrid DRAM/NVM memory system. In *International Symposium on Microarchitecture (MICRO)*.

[22] Jungi Jeong and Changhee Jung. 2021. PMEM-Spec: persistent memory speculation (strict persistency can trump relaxed persistency). In *ACM ASPLOS*.

[23] 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.

[24] Rajat Kateja, Anirudh Badam, Sriram Govindan, Bikash Sharma, and Greg Ganger. 2017. Viyojit: Decoupling battery and DRAM capacities for battery-backed DRAM. *ACM SIGARCH Computer Architecture News* 45, 2 (2017), 613–626.

[25] Stefanos Kaxiras and Alberto Ros. 2013. A new perspective for efficient virtual-cache coherence. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*. 535–546.

[26] 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.

[27] Seongwoo Kim and Arun K Somani. 1999. Area efficient architectures for information integrity in cache memories. *ACM SIGARCH Computer Architecture News* 27, 2 (1999), 246–255.

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

[29] Aasheesh Kolli, Jeff Rosen, Stephan Diestelhorst, Ali Saidi, Steven Pelley, Sihang Liu, Peter M Chen, and Thomas F Wenisch. 2016. Delegated persist ordering. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–13.

[30] Vito Kortbeek, Kasim Sinan Yildirim, Abu Bakar, Jacob Sorber, Josiah Hester, and Przemysław Pawełczak. 2020. Time-sensitive intermittent computing meets legacy software. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 85–99.

[31] Chunho Lee et al. 1997. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communicatons Systems. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*.

[32] Hsien-Hsin S Lee, Gary S Tyson, and Matthew K Farrens. 2000. Eager writeback-a technique for improving bandwidth utilization. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*. 11–21.

[33] Lin Li, Vijay Degalahal, Narayanan Vijaykrishnan, Mahmut Kandemir, and Mary Jane Irwin. 2004. Soft error and energy consumption interactions: A data cache perspective. In *Proceedings of the 2004 international symposium on Low power electronics and design*. 132–137.

[34] 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.

[35] 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.

[36] Qingrui Liu and Changhee Jung. 2016. Lightweight Hardware Support for Transparent Consistency-Aware Checkpointing in Intermittent Energy-Harvesting systems. In *Proceedings of the IEEE Non-Volatile Memory Systems and Applications Symposium (NVMSA)*.

[37] Qingrui Liu, Changhee Jung, Dongyoon Lee, and Devesh Tiwari. 2015. Clover: Compiler Directed Lightweight Soft Error Resilience. In *Proceedings of the 16th ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems 2015*.

[38] Qingrui Liu, Changhee Jung, Dongyoon Lee, and Devesh Tiwari. 2016. Compiler-directed lightweight checkpointing for fine-grained guaranteed soft error recovery. In *High Performance Computing, Networking, Storage and Analysis, (SC16)*.

[39] 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, 32.

[40] 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 *49th IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[41] 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.

[42] 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.

[43] Kaisheng Ma, Jinyang Li, Xueqing Li, Yongpan Liu, Yuan Xie, Mahmut Kandemir, Jack Sampson, and Vijaykrishnan Narayanan. 2018. IAA: Incidental approximate architectures for extremely energy-constrained energy harvesting scenarios using IoT nonvolatile processors. *IEEE Micro* 38, 4 (2018), 11–19.

[44] Kaisheng Ma, Xueqing Li, Mahmut Taylan Kandemir, Jack Sampson, Vijaykrishnan Narayanan, Jinyang Li, Tongda Wu, Zhibo Wang, Yongpan Liu, and Yuan Xie. 2018. NEOFog: Nonvolatility-exploiting optimizations for fog computing. In *Proceedings of the Twenty-Third International Conference on Architectural*

Support for Programming Languages and Operating Systems. 782–796.

[45] Kaisheng Ma, Xueqing Li, Jinyang Li, Yongpwan Liu, Yuan Xie, Jack Sampson, Mahmut Taylan Kandemir, and Vijaykrishnan Narayanan. 2017. Incidental computing on IoT nonvolatile processors. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 204–218.

[46] Kaisheng Ma, Xueqing Li, Huichu Liu, Xiao Sheng, Yiqun Wang, Karthik Swaminathan, Yongpan Liu, Yuan Xie, John Sampson, and Vijaykrishnan Narayanan. 2017. Dynamic power and energy management for energy harvesting nonvolatile processor systems. *ACM Transactions on Embedded Computing Systems (TECS)* 16, 4 (2017), 1–23.

[47] Kaisheng Ma, Xueqing Li, Srivatsa Rangachar Srinivasa, Yongpan Liu, John Sampson, Yuan Xie, and Vijaykrishnan Narayanan. 2017. Spendthrift: Machine learning based resource and frequency scaling for ambient energy harvesting nonvolatile processors. In *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 678–683.

[48] Kaisheng Ma, Xueqing Li, Karthik Swaminathan, Yang Zheng, Shuangchen Li, Yongpan Liu, Yuan Xie, John Jack Sampson, and Vijaykrishnan Narayanan. 2016. Nonvolatile processor architectures: Efficient, reliable progress with unstable power. *IEEE Micro* 36, 3 (2016), 72–83.

[49] 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.

[50] Kiwan Maeng and Brandon Lucia. 2019. Supporting peripherals in intermittent systems with just-in-time checkpoints. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 1101–1116.

[51] Kiwan Maeng and Brandon Lucia. 2020. Adaptive low-overhead scheduling for periodic and reactive intermittent execution. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 1005–1021.

[52] Sanketh Nalli, Swapnil Haria, Mark D Hill, Michael M Swift, Haris Volos, and Kimberly Keeton. 2017. An analysis of persistent memory use with WHISPER. *ACM SIGPLAN Notices* 52, 4 (2017), 135–148.

[53] 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.

[54] Shahriar Nirjon. 2018. Lifelong Learning on Harvested Energy. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 500–501.

[55] Ebelechukwu Nwafor, Andre Campbell, David Hill, and Gedare Bloom. 2017. Towards a provenance collection framework for Internet of Things devices. In *2017 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computed, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*. IEEE, 1–6.

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

[57] Benjamin Ransford, Jacob Sorber, and Kevin Fu. 2012. Mementos: System support for long-running computation on RFID-scale devices. *Acm Sigplan Notices* 47, 4 (2012), 159–170.

[58] Alberto Rodriguez Arreola, Domenico Balsamo, Anup K. Das, Alex S. Weddell, Davide Brunelli, Bashir M. Al-Hashimi, and Geoff V. Merrett. 2015. Approaches to Transient Computing for Energy Harvesting Systems: A Quantitative Evaluation. In *Proceedings of the 3rd International Workshop on Energy Harvesting & Energy Neutral Sensing Systems* (Seoul, South Korea) *(ENSsys '15)*. ACM, New York, NY, USA, 3–8. https://doi.org/10.1145/2820645.2820652

[59] Alberto Ros and Stefanos Kaxiras. 2012. Complexity-effective multicore coherence. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*. 241–252.

[60] Abdallah M Saleh, Juan J Serrano, and Janak H Patel. 1990. Reliability of scrubbing recovery-techniques for memory systems. *IEEE transactions on reliability* 39, 1 (1990), 114–122.

[61] 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.

[62] Premkishore Shivakumar and Norman P Jouppi. 2001. Cacti 3.0: An integrated cache timing, power, and area model. (2001).

[63] Emiliano Sisinni, Abusayeed Saifullah, Song Han, Ulf Jennehag, and Mikael Gidlund. 2018. Industrial Internet of Things: Challenges, Opportunities, and Directions. *IEEE Transactions on Industrial Informatics* (2018).

[64] Jeffrey Stuecheli, Dimitris Kaseridis, David Daly, Hillery C Hunter, and Lizy K John. 2010. The virtual write queue: Coordinating DRAM and last-level cache policies. *ACM SIGARCH Computer Architecture News* 38, 3 (2010), 72–82.

[65] Fang Su, Yongpan Liu, Yiqun Wang, and Huazhong Yang. 2017. A Ferroelectric Nonvolatile Processor with 46μ s System-Level Wake-up Time and 14μ s Sleep Time for Energy Harvesting Applications. *IEEE Transactions on Circuits and Systems I: Regular Papers* 64, 3 (2017), 596–607.

[66] Fang Su, Kaisheng Ma, Xueqing Li, Tongda Wu, Yongpan Liu, and Vijaykrishnan Narayanan. 2017. Nonvolatile processors: Why is it trending?. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 966–971.

[67] Sandeep Krishna Thirumala, Arnab Raha, Hrishikesh Jayakumar, Kaisheng Ma, V Narayanan, Vijay Raghunathan, and Sumeet Kumar Gupta. 2018. Dual mode ferroelectric transistor based non-volatile flip-flops for intermittently-powered systems. In *Proceedings of the International Symposium on Low Power Electronics and Design*. 1–6.

[68] Haris Volos, Andres Jaan Tack, and Michael M Swift. 2011. Mnemosyne: Lightweight persistent memory. *ACM SIGARCH Computer Architecture News* 39, 1 (2011), 91–104.

[69] Yiqun Wang et al. [n. d.]. A 3us wake-up time nonvolatile processor based on ferroelectric flip-flops. In *ESSCIRC, 2012 Proceedings of the*.

[70] 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)*.

[71] Mimi Xie, Chen Pan, Jingtong Hu, Chengmo Yang, and Yiran Chen. 2015. Checkpoint-aware instruction scheduling for nonvolatile processor with multiple functional units. In *Design Automation Conference (ASP-DAC), 2015 20th Asia and South Pacific*. IEEE, 316–321.

[72] Mimi Xie, Chen Pan, Youtao Zhang, Jingtong Hu, Yongpan Liu, and Chun Jason Xue. 2018. A novel stt-ram-based hybrid cache for intermittently powered processors in iot devices. *IEEE Micro* 39, 1 (2018), 24–32.

[73] 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 *2016 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*. IEEE, 1–10.

[74] Sujay Yadalam, Nisarg Shah, Xiangyao Yu, and Michael Swift. 2022. ASAP: A speculative approach to persistence. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 892–907.

[75] 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 *International Symposium on Microarchitecture*.

[76] 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.

[77] Yida Zhang and Changhee Jung. 2022. Featherweight Soft Error Resilience for GPUs. In *55th International Symposium on Microarchitecture*. 101–108.