Delft University of Technology

Data Cache for Intermittent Computing Systems with Non-Volatile Main Memory

Mohapatra, Sourav; Kortbeek, Vito; Ahmed, Saad; Broekhoff, Jochem; Ahmed, Saad; Przemysław, Przemysław

DOI
10.1145/3676641.3715989

Licence
CC BY

Publication date
2025

Document Version
Final published version

Published in
ASPLOS '25

Citation (APA)
Mohapatra, S., Kortbeek, V., Ahmed, S., Broekhoff, J., Ahmed, S., & Przemysław, P. (2025). Data Cache for Intermittent Computing Systems with Non-Volatile Main Memory. In *ASPLOS '25: Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (pp. 227-243). ACM. https://doi.org/10.1145/3676641.3715989

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Copyright
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy
Please contact us and provide details if you believe this document breaches copyrights.
We will remove access to the work immediately and investigate your claim.

This work is downloaded from Delft University of Technology.
For technical reasons the number of authors shown on this cover page is limited to a maximum of 10.

Data Cache for Intermittent Computing Systems with Non-Volatile Main Memory

Sourav Mohapatra[*]
ARM Ltd.
Cambridge, United Kingdom
sm.sourav.mohapatra@gmail.com

Vito Kortbeek[*]
Synopsys
Enschede, The Netherlands
kortbeek@synopsys.com

Marco Antonio van Eerden
TU Delft
Delft, The Netherlands
m.a.vaneerden@student.tudelft.nl

Jochem Broekhoff[†]
TU Delft
Delft, The Netherlands
j.h.broekhoff@student.tudelft.nl

Saad Ahmed
Georgia Institute of Technology
Atlanta, GA, USA
sahmed@gatech.edu

Przemysław Pawełczak
TU Delft
Delft, The Netherlands
p.pawelczak@tudelft.nl

Abstract

Intermittently-operating embedded computing platforms powered by energy harvesting must frequently checkpoint their computation state. Using non-volatile memory reduces checkpoint size by eliminating the need to checkpoint volatile memory but increases checkpoint frequency to cover Write After Read (WAR) dependencies. Additionally, non-volatile memory is significantly slower to access—while consuming more energy than its volatile counterpart—suggesting the use of a data cache. Unfortunately, existing data cache solutions do not fit the challenges of intermittent computing and often require additional hardware or software to detect WARs. In this paper, we extend the data cache by integrating it with WAR detection—dropping the need for an additional memory tracker. This idea forms the basis of NACHO: a data cache tailored to intermittent computing. NACHO, on average, reduces intermittent computing runtime overhead by 54% compared to state of the art cache-based systems. It also reduces the number of non-volatile memory writes by 82% compared to a data cache-less system, and 18% on average compared to multiple state of the art cache-based systems.

CCS Concepts: **• Computer systems organization → Embedded systems**.

Keywords: intermittent computing, battery-free, cache, embedded system

[*]This research was performed when the author was affiliated with TU Delft, Delft, The Netherlands.
[†]Jochem Broekhoff is also affiliated with DSP Innovation B.V., Middelburg, The Netherlands.

This work is licensed under a Creative Commons Attribution 4.0 International License.
ASPLOS '25, Rotterdam, Netherlands
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1079-7/25/03
https://doi.org/10.1145/3676641.3715989

ACM Reference Format:
Sourav Mohapatra, Vito Kortbeek, Marco Antonio van Eerden, Jochem Broekhoff, Saad Ahmed, and Przemysław Pawełczak. 2025. Data Cache for Intermittent Computing Systems with Non-Volatile Main Memory. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3676641.3715989

1 Introduction

The growth of the Internet of Things (IoT) sector is causing a surge in number of embedded devices globally [7], raising sustainability concerns [12, 69]. Batteries powering these devices require proper disposal [23], frequent replacement [19], and pose fire hazards [40]. These issues can be mitigated by using sustainable (super-)capacitors [62, 67] and powering devices via ambient energy sources like sunlight, electromagnetic radiation, or vibrations [1, 3, 57].

Using harvested energy comes with the challenges of (i) the ambient's energy inherent unreliability [60] and (ii) the low output energy due to IoT devices' size constraints—limiting energy storage element's size. These factors force random power failures [59], [14, Sec 8.1], causing the device to completely lose power and reboot only when sufficient energy is available again[1].

To make forward progress, i.e., to continue a program amidst intermittent power failures, the system must recover the program state saved before the power failure and continue execution after a reboot [37, 61, 66], resulting in *intermittent execution of the program* [26, 42]. Despite these difficulties, successful battery-free intermittently-powered systems have emerged, including general-purpose [16, 20] and dedicated [15] wireless sensors, cell phones [65], electronic prototyping [39], and gaming platforms [17].

[1]Consider a system [64, Section 3A] with 1 μJ energy consumption, an energy harvesting source of 20 μW/cm^2 and a 46 μF capacitor. It takes 22 seconds to charge a capacitor to perform computations, after which the system turns off and recharges again.

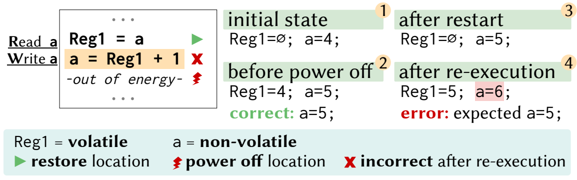

Figure 1. An example of a memory sequence in which a *read* operation (**R**) is followed by a *write* (**W**) can result in an inconsistent memory after a power failure.

Saving the intermediate state of the program executed intermittently can be done in Non-Volatile Memory (NVM) [44, 64]. Magnetoresistive Random Access Memory (MRAM) or Ferroelectric Random Access Memory (FRAM) are well suited for intermittent computing due to their byte-addressability and lower power consumption compared to traditional NVM technologies such as FLASH. The persistence of NVM enables it to checkpoint program states before power failures, supporting actions like backups and rollbacks of volatile components. However, copying and restoring volatile states is inefficient and drains the already limited harvested energy. A recent approach [68] addresses this with a replay-and-bypass method to recover without user-managed checkpoints. Alternatively, the NVM can replace the volatile main memory [37, 38, 46, 66], reducing the volatile state of the complete system (in most cases) only to the registers. However, substituting Static Random Access Memory (SRAM) with MRAM as the main memory in embedded devices is not straightforward. This transition introduces memory consistency challenges, particularly due to the re-execution of WAR accesses after a reboot, which can lead to potential memory corruption if not properly managed (see Figure 1).

One way to mitigate the problem caused by re-execution is to create a *checkpoint* of the reigsters between the read and write of all WAR dependencies. Checkpoints can be inserted statically at compile time [37, 66], or dynamically during execution by using dedicated hardware to track memory accesses, triggering checkpoint insertion upon detecting a WAR [27]. Unfortunately, using NVM as the main memory in intermittently-powered devices introduces downsides that degrade performance. WAR dependencies, common during program execution, require numerous checkpoints to safeguard against power failures—*significantly more* than needed to ensure forward progress. Moreover, any intermittent computing framework must be *incorruptible* [37, 38, 46, 66, 71], ensuring correct continuation from the last completed checkpoint, even if power fails during checkpoint creation. Looking at the results of a recent software-based approach [37, Figure 4], even the most optimized solution has double the execution time compared to native unmodified binaries without checkpoints (while still using NVM as the main memory). Using hardware detection of power failure, such as [27, 74], results in less overhead, as the program does not require to be over-instrumented with checkpoints by the compiler. However, NVMs such as FRAM and MRAM are still considerably slower and require more energy to access than their volatile counterpart SRAM [22, Section 2], [30, Section 8.4]. Hence, intermittent systems would *benefit from finding a balance between using volatile and non-volatile memory.*

System Assumptions: Intermittent computing targets resource-constrained microcontrollers that operate under severe power and computational limitations. These systems rely on energy harvesting, leading to frequent power failures which necessitate carefully designed execution models. Unlike high-performance and general-purpose architectures, intermittent systems *generally* operate under the following key constraints:

① Devices use a *simple in-order single-core processors* without advanced features such as out-of-order or speculative execution pipelines.
② The system works *without a full scale OS or preemptive multitasking* which means that there is no context switching or thread scheduling. The applications typically run in a bare-metal environment or in a lightweight runtime.
③ The memory model we use to propose a data cache features *simple cache hierarchies.* Complex cache architectures require energy that we argue is beyond the boundaries of intermittent deployment scenarios.
④ Due to unpredictable power failures being inherent to the context of intermittent systems, *real-time execution guarantees are infeasible.* Any system that requires strict timing constraints must perform some form of timekeeping [16, 34] to track critical sections during power failures, which diverges from our contributions.
⑤ This work assumes that programs are *compiled for simple, embedded execution environments* without advanced runtime mechanisms such as shadow stacks or transactional memory.

Within these constraints, our goal is to integrate a volatile data cache with intermittent computing paradigm to improve performance while ensuring correct execution under power failures. Unlike traditional cache-based designs that assume continuous power availability, our approach explicitly addresses the intermittency in its execution model. We discuss limitations associated with the above assumptions in Section 8.

Problem Statement: Previous works attempted to achieve above balance by reducing the size of checkpoints while still using volatile components, i.e., a mixed-memory model [48]. Another direction is using a volatile *data cache* in combination with non-volatile main memory. Adding a data cache will decrease the cost of using non-volatile main memory by allowing for faster access speeds and less NVM accesses. However, integrating a data cache with intermittent systems is not straightforward. The system still needs to address WAR hazards, which become even more complicated in the presence of cache [11, 73, 75], as the cache delays the actual writeback to NVM. Then, the cache eviction policy that determines which cache block must be evicted to make space for new data (i) must be aware of when the checkpoint will happen and (ii) how to proceed whilst maintaining consistency. Furthermore, since the cache is a volatile entity, it must be written to NVM before a checkpoint. State of the art cache-based systems ReplayCache [73] and SweepCache [75] use a compiler-based approach to ensure this. This custom compiler limits the use of legacy code, and specialized hardware is required to leverage asynchronous write-back of cache lines. This makes this approach complex and costly to implement. Write-Light Cache [11] similarly requires extra hardware for its implementation.

Our Fundamental Insight: We argue that simply applying existing data cache methods [25, Appendix B] to intermittent computing architectures is inefficient. We take a different position and propose modifications to the cache's workings to better align with intermittent computing. We found that by adding just two bits to the data cache lines and using these to detect whether a writeback to the NVM is *safe*, we can directly **use the cache for WAR detection** and mitigation to break up WAR dependencies, instead of relying on additional WAR detection hardware [27, 74] or software systems [37]. Utilizing the cache as WAR detection should reduce the total number of required checkpoints, NVM memory accesses, and execution time.

Our Contributions: We present NACHO, a data cache architecture for intermittent computing systems. Specifically:

① We define the requirements for a safe data cache in an intermittent computing system with non-volatile main memory. These requirements form the basis of NACHO;
② NACHO, by adding *only two extra bits* per cache entry combined with a novel algorithm, is able to detect if a write back to memory is *safe*, i.e., not a read-dominated WAR dependency;
③ NACHO reduces the checkpoint size by tracking the stack of the executing program and avoid writing memory that is no longer valid to the NVM.

Through these contributions, for an example 512 B cache, we *reduce an overhead* introduced by supporting intermittent computing compared to Clank [27], PROWL [28], and ReplayCache [73]—all state of the art solutions relevant to NACHO—by on average *53%*, *65%*, and *55%*, respectively. NACHO will be an *open source project*, see Appendix A.

2 Cache and Intermittency

The size of volatile and non-volatile memory components in an embedded battery-free system greatly impacts the total execution time and consumed energy [28, Table 1]. Finding the right trade-off between the number of volatile and non-volatile components *motivates using cache for fast and energy-efficient intermittent system architectures.*

2.1 Trade-off in Memory Persistency

In the case of completely non-volatile systems such as Non-Volatile Processors (NVPs) [44], everything is in persistent storage, and the need to create checkpoints and restore them in case of power failure disappears. However, the energy consumed to perform memory accesses increases, diminishing the gains resulting from memory non-volatility. As we move towards the opposite spectrum of fully volatile architectures, the energy consumed per cycle decreases, but the size and number of the checkpoints and the cost of re-execution increases. Everything needs to be saved and restored to/from a fully volatile system, again skewing the associated costs. The desired solution is thus a *balance between the volatile and non-volatile system.* That is, we should seek a solution where a volatile SRAM-based data cache provides high access speeds—while being small enough to ensure low checkpointing overheads—and where a persistent non-volatile main memory ensures data retention across power failures.

2.2 Challenges of Intermittency and Cache

Integrating a cache into an intermittent system is not straightforward. Let us look at an example in Figure 2, where we compare a traditional system supporting intermittent operation with a data cache-based system. In Figure 2, case ②, because the checkpoint placement logic depends on when a "write" to NVM is performed, having a cache that buffers the memory accesses delays the write operation to NVM. This is a runtime phenomenon that the compiler cannot predict, thus rendering the checkpoint placement incorrect.

One might ask, why not use a more complex compiler-directed cache-based system, like ReplayCache [73]. With ReplayCache, compiler transformations create idempotent regions in combination with a parallel cache writeback instruction, replaying cache modification after a failure. However, the ReplayCache-based method limits the use of legacy code and adds a significant amount of complexity in addition to a highly customized cache. One could use an extra dedicated hardware, such as the memory tracker in Clank [27], to be deployed *in addition* to a data cache. However, this approach also increases the cost and complexity of the system.

We state that integration between intermittent computing systems using non-volatile main memory and a data cache should not merely utilize an existing cache architecture but rather tightly couple it with detecting WAR violations. Additionally, it should actively attempt to minimize the number of NVM accesses by considering the behavior of the execution flow of a program running on battery-free, intermittently powered systems. Looking at Table 1, which compares the most relevant systems for intermittent computation, *no existing solution addresses all* system requirements.

Table 1. Features of state of the art intermittent computing systems focusing on ones with a data cache.

	Clank [27]	COACH [29]	ReplayCache [73]	SweepCache [75]	WL-Cache [11]	NvMR [9]	PROWL [28]	**NACHO** (this work)
supports data cache	✗	✓	✓	✓	✓	✓	✓	✓
low checkpoint count	✗	✗	✓	✗	✓	✓	✓	✓
low NVM reads/writes	✗	✗	✓	✓	✗	✓	✓	✓
incorruptible	✓	*partially*†	*partially*†	✓	*partially*†	✓	✓	✓
no compiler transformations	✓	✓	✗	✗	✓	✓	✓	✓
cache architecture-agnostic	*n/a*	✓	✗	✓	✓	✓	✗	✓
no extra hardware required	*n/a*	✗	✓	✗	✗	✗	✓	✓
tight data cache integration	*n/a*	✗	✗	✗	✗	✗	✗	✓
considers program execution flow	*n/a*	✗	✗	✗	✗	✗	✗	✓

Yes: ✓, No: ✗ † The work relies on existing checkpointing strategies, thus it can be as incorruptible as the choice of strategy.

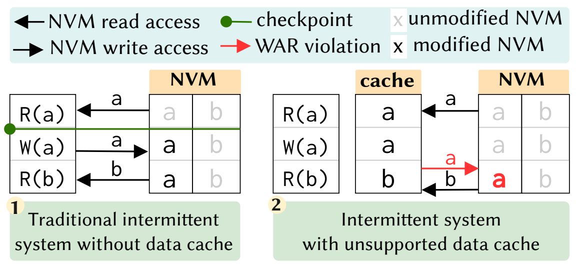
NVM read access
NVM write access
checkpoint
WAR violation
unmodified NVM
modified NVM
NVM
cache
R(a)
W(a)
R(b)
1 Traditional intermittent system without data cache
2 Intermittent system with unsupported data cache

Figure 2. An example program performing memory accesses **R**(x) (a read operation at the memory location x) and **W**(x) (corresponding write operation) on two variables, for two systems. System ① is a cache-less (de facto standard, e.g. [66]), where read and write operations interact with NVM—note a compulsory checkpoint at WAR of variable 'a' inserted at compile time. System ② is based on a regular cache which cannot support WAR tracking by design, i.e., a simple direct mapped write-back [25, Appendix B] cache of two lines. In ②, checkpoints cannot be inserted at compile time because the compiler will not know when the eviction of cache-located variable 'a' back to NVM will take place.

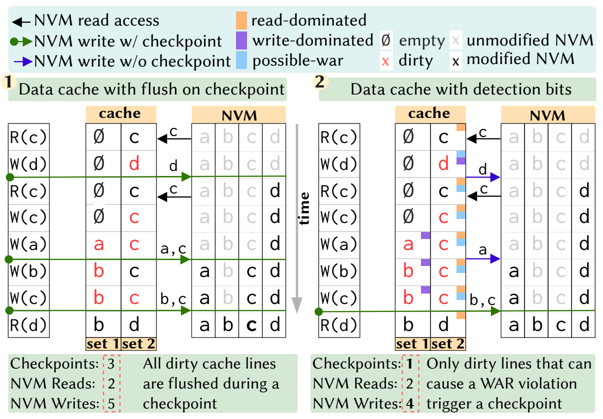

Figure 3. *Cache* and *NVM accesses* for an example program. A NVM access is a *read* from the NVM to the cache or a *cache eviction* of a dirty block from the cache to the NVM. The data cache here is *direct associative* with two sets. In some cases, this can cause a checkpoint signal to be raised, which is sent to the processor. In this case, all dirty cache blocks are evicted to the NVM, but kept in the cache. There are four arbitrary memory blocks *a, b, c, d* that are assigned to two cache sets as follows: a, b ⟶ set 1 and c, d ⟶ set 2.

3 Cache for Intermittent Systems

We propose a fundamentally different approach to overcome the challenge mentioned in Section 2.2. Our *data cache* will be tasked with (i) avoiding WAR (Section 3.1), (ii) optimizing WAR detection (Section 3.2), and (iii) reducing the number of NVM writes (Section 3.3).

3.1 Cache for Avoiding Write After Reads

First, we observe that a data cache *delays* the *write* to NVM, until a cache eviction forces that *write*. A WAR violation can only occur when a *write* is performed after a *read* at an NVM location. Therefore, the cache effectively determines when a WAR can occur by tracking the presence of such access patterns in a given cache line. In other words, we take advantage of the fact that *a WAR violation is only possible when a cache block is written back to the NVM*. We term this event, i.e. a cache line being written back to the NVM, as a *Cache Write Back (CWB)* (similar to Intel x86 CLWB instruction [31, Page 744].) Upon detecting a CWB, the cache generates a checkpoint signal and instructs the processor to create a checkpoint. During the checkpoint, the processor copies the registers the volatile data cache (otherwise, the volatile data would be lost) to the NVM. To this end, all modified, i.e., dirty, memory blocks are copied (i.e., flushed) to the NVM during a checkpoint. By creating a checkpoint in this way, *we ensure that the system remains consistent.* An added advantage is clearing the cache of *all* dirty lines during the checkpoint, which decreases the possibility of future

WARs and thus reduces the number of created checkpoints and NVM accesses. We illustrate this process in Figure 3 ①.

3.2 Cache for Optimizing WAR Detection

As explained above, a CWB *can* lead to WAR violations. However, some CWBs may not. This can be understood more formally as the memory being *read-dominated* or *write-dominated* as introduced in [27, Section 3.1.1]. In a sequence of memory instructions, if the first memory access is a write, then that location is write-dominated. Conversely, if the first memory access in a given sequence of instructions is a read, then that location is read-dominated. An idempotency violation can then be defined as *a write to a read-dominated memory location.* Any other form of access is safe and will not cause a WAR violation. Since any given memory sequence can be read-dominated or write-dominated, this condition bounds all possible idempotency violations. With this understanding, Clank [27] used dedicated hardware to track whether memory accesses are read- or write-dominated. In contrast, we use *the cache to perform this same tracking*, eliminating the need for an additional hardware component. Henceforth, we will denote write-dominated WARs as *safe*, and read-dominated WARs as violations.

To help understand the above, we redefine read-dominated and write-dominated sequences to track a cache line instead of a memory address. A cache line is read-dominated when the first access to the line is a *read* and write-dominated when the first access is a *write*. If a CWB comes from a write-dominated cache line, it does not result in a WAR violation, and therefore does not require special action. We term this write-back as a *safe write*. A CWB can only result in a violation if the associated cache line is read-dominated, which we term as an *unsafe write*. We track these memory sequences to all cache lines during program execution and create a checkpoint only if an *unsafe write* is encountered. The exact functionality of this tracking is discussed in Section 4.

The above process is shown in Figure 3 ②. Compared to Figure 3 ①, we notice a reduction in the number of checkpoints and NVM writes. An important thing to note is that since the cache stores data based on a hash of the memory address, the *distinction between safe-write and unsafe-write is also based on the hashed address.* This implies that the WAR detection is not exact, and although it can never contain false negatives, it does lead to few false positives. This is a trade-off in using the cache (and not a dedicated hardware module, as in e.g. [74]) as a memory tracker. However, we show later (Section 6) that this impact is mostly negligible.

3.3 Reducing Unnecessary NVM Writes

Not all memory in the system is still in use (*live*) during a checkpoint. Consider a program's stack memory, it is allocated/deallocated constantly during execution when entering/leaving functions. However, stack memory that is no longer in use, i.e., has been deallocated, will never be read first during execution but is still marked as *dirty* in the data cache. This insight is based on the fact that unallocated stack is always first written to when allocated. Hence the unallocated stack does not need to be written to NVM at a checkpoint, potentially reducing the number of WAR violations and NVM writes during a checkpoint.

4 System Architecture

We present NACHO: an architecture based on the data cache design presented in Section 3.

4.1 System Requirements

Along with intermittent computing, NACHO also supports the following requirements.

Incorruptibility: NACHO ensures that the program's state is correct. As shown in Table 1, state of the art systems do not always guarantee incorruptibility. Just In Time (JIT) checkpointing used in past and recent systems, e.g. [58], is inherently unsafe in intermittent computing due to its reliance on accurately predicting when power failures will occur [61, Figure 1]. Misjudging the timing can lead to missed checkpoints, resulting in inconsistent system states and potential data loss. Additionally, JIT does not adequately handle sudden power drops, making it difficult to ensure reliable execution, particularly when dealing with critical tasks. Frequent power fluctuations compound this issue, as the system may not have enough time to create a valid checkpoint. By using the data cache as a WAR detector, we guarantee memory consistency without energy prediction to create checkpoints.

Cache Architecture Agnostic: Even though our system incorporates a custom data cache, NACHO is agnostic to the cache architecture (with any placement/replacement policies). The core concept of NACHO relies solely on the addition of two extra bits per cache line to track memory accesses for efficient WAR detection. This mechanism has no inherent dependency on the cache architecture, enabling it to determine when a write-back to non-volatile memory is safe, and thus when a checkpoint is necessary. This makes NACHO a versatile solution that can be incorporated into different types of caches in intermittent computing systems.

4.2 Data Cache Controller

The WAR violation detection mechanism introduced in Section 3.2 glosses over real-world obstacles that prevent it from working in a fully-functioning data cache. In this section, we address the simplifications made and describe the exact workings of the optimized WAR detector.

4.2.1 Cache Line Bits.

In Section 3 we introduced the concept of *read-dominated* and *write-dominated* memory in the cache line in order to detect WAR violations. While we continue tracking both memory patterns, we optimize their representation to minimize the number of additional bits required. Instead of introducing a dedicated *write-dominated*

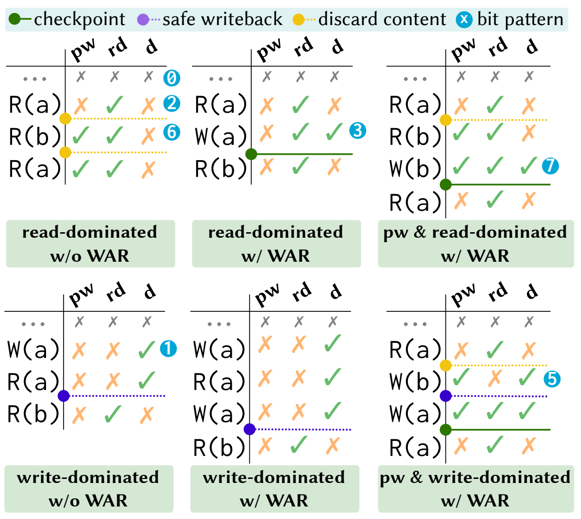

Figure 4. Six memory sequences and their corresponding bit patterns, including their decimal representation (blue circles). Every memory access above maps to the one shown cache line. The data in the cache line is omitted. Only the cache line's three bits of interest are shown: **pw** (possible-war), **rd** (read-dominated), and **d** (dirty). The figure depicts all possible bit patterns. Note that configuration 4 (only the **pw** bit set) is invalid and can never occur.

bit, we infer write domination from the *dirty* bit (which is already present in a standard data cache) and the *read-dominated* bit. A memory access is thus classified as *write-dominated* if the cache line was first *read-dominated* and was later written to, which inherently sets the *dirty* bit. Additionally, we introduce the *possible-war* bit to reduce the conservativeness when detecting WAR violations by incorporating history information into the cache line. In total, this results in **just two additional bits** compared to a standard data cache line, *possible-war* and *read-dominated*. Figure 4 shows all the possible bit patterns and the memory sequences required to reach them.

4.2.2 The Possible WAR Bit. The *possible-war* bit is set when the cache line is *read-dominated* and the data in a cache line is replaced. Multiple memory addresses are mapped to the same cache line when using a cache. While the *read-dominated* and *write-dominated* bits are a good start, they fall short when considering a memory location that is read into the cache, then evicted, and later written on to (scenario "pw & write dominated w/ WAR" in Figure 4). In this scenario, the cache line would not be marked as *read-dominated* if the *possible-war* was not set (scenario "pw & read dominated w/ WAR"). Without the *possible-war* bit, all writes after a read to the data cache must be marked as *read-dominated*, leading to more checkpoints. However, this scenario could never be a WAR violation, as the incoming write **must** be to another memory address to evict the original entry. Thus the *possible-war* bit functions as a one-bit history, recording if there was a *read-dominated* cache entry in the block since the last checkpoint. But since the *possible-war* bit is set last, it will not be taken into consideration during the first transition from a *read-dominated* to a *write-dominated* cache line.

4.2.3 Possible WAR and Cache Associativity. When applying cache bits to track WAR violations, we must consider the data cache associativity, i.e., the number of "ways" in the cache. Assuming a n-way cache, the cache controller can map a memory location to n different cache lines. The placement of memory address in a cache line depends on the cache replacement policy, e.g., *least recently used.* Now, consider a memory read to location m, marking the cache block as *read-dominated.* Next, the line containing m is evicted, removing it from the cache. Finally, memory location m is written to; however, this time, the data is written to another cache block for the same hash (which is possible when n is greater than one, i.e., the cache is not directly mapped). If this final cache line does not have its *possible-war* bit set, it will be marked *write-dominated.* Because the same memory location m was read before—but was not detected because it occurred in a different cache line—this can lead to a WAR violation since the cache line is mismarked as *write-dominated.* To avoid this incorrect *write-domianted* marking, we must slightly change our approach to set the *possible-war* bit. Instead of considering the *possible-war* bit of only the cache line to which data is moved, we must consider all the n cache lines in the set when deciding if it can be marked as *write-dominated.*

4.2.4 Stack Tracking. To both improve the execution time and lower the energy consumption, writing to the NVM should be avoided as much as possible. One situation where data in the cache is written back to NVM without it ever being read again is when a deallocated stack frame, i.e., a stack frame no longer in use because the function completed, is written back to NVM. To avoid writing this data back to NVM, we need to track the stack movement of the program. This can be straightforward, as the top of the stack is constantly being tracked by the Central Processing Unit (CPU)'s stack pointer sp. By also tracking sp_{min}, i.e., the lowest address the stack pointer reaches between two checkpoints (assuming the stack memory grows downwards in memory), we can discard all memory between sp and sp_{min}. By applying this technique, we avoid writing a dirty cache line to NVM during a checkpoint or cache eviction.

4.2.5 Data Cache Controller Algorithm. Algorithm 1 shows the manipulation of the two extra bits introduced by this work: *possible-war* (**pw**) and *read-dominated* (**rd**), in addition to the *dirty* bit (**d**), for each memory request. The other cache line-related bits (e.g., valid) and functionality (e.g., details regarding the ReplacementPolicy) are not shown in Algorithm 1 for the sake of brevity. We will now go through this algorithm, discussing each procedure in detail.

MemoryAccess: During a memory request (Line 1), the default data cache behavior is first to check if the request is a miss (Line 3). If the request results in a miss, an existing cache line must be evicted to make room for the new request. If the request is a hit, the memory location requested already resides in the cache. When a hit occurs, we introduce one special case. If this is the first hit for this cache line after a checkpoint was created, the cache line bits must be updated using the UpdateLine procedure (Line 20). We can identify this is the first hit by checking if all the considered flags are cleared (Line 5). If the cache line has been visited before, no bit changes are needed, and the cache hit continues as usual.

CacheMiss: When a cache miss occurs (Line 8), we use a standard cache replacement policy, e.g., least recently used, to select the line that must be evicted to make room for the new request (Line 9). If the current line is not dirty, i.e., it was only read and never written to (Line 10), we can safely discard the data in the cache line and finish the request by updating the bits (Line 18). Later we update the cache line with data from the NVM (Line 7). If the cache line to be evicted is dirty, we cannot simply write the current content of the line to NVM, as this could cause a WAR violation. Instead, we first check if we can ignore the data because it is in a region of the stack that is no longer live (Line 11), in which case we can reset the cache line (Line 17). After all, the data does not need to be written back to NVM.

If the memory might still be in use, we check whether the memory accesses *could* have been read-dominated by checking the **rd** flag associated with the cache line (Line 12). If the cache line could be *read-dominated*, writing the memory back to NVM *could* cause a WAR violation, so we must create a checkpoint (Line 13). However, if we know for sure that the cache line is *write-dominated*, which **must** be the case if the cache line is *dirty* and **not** *read-dominated*, we can safely write (evict) the data directly to the NVM without creating a checkpoint (Line 15). Finally, we can continue the cache miss as usual by updating the cache line (Line 18) and returning the updated line to the MemoryAccess procedure.

UpdateLine: If the cache line is currently *read-dominated* then the *possible-war* bit must be set after updating the other bits to indicate that the *next* access could be *read-dominated* even if the request is a write (Line 33). If the request is a read (Line 22), only the *read-dominated* flag has to be set. However, if the request is instead a write (Line 24), we consider if any of the lines in the set has their *possible-war* bit set (Line 29), or the size of the requrest is not equal to the size of the cache line (four bytes), in which case we must mark the current line as *read-dominated* (Line 32). Otherwise the *read-dominated* bit is cleared, marking it as *write-dominated*. The size requirement is introduced because a write request smaller than the cache line will first read from the NVM, which could lead to a WAR violation.

Algorithm 1: Data cache controller

```
1  Algorithm MemoryAccess(address, type, value, size) :
2      line, miss ← CacheLine(address)
3      if miss is true then
4          line = CacheMiss(address, type, size)
5      else if (line_pw is false) and (line_rd is false) and (line_dirty is false)
         then
6          UpdateLine(line, type, size)    // 1st hit after checkpt
7      UpdateData(line, value)     // Fill cache line with data

8  Procedure CacheMiss(address, type, size) :
9      line ← ReplacementPolicy(address)      // Evicting line
10     if line_dirty is true then
11         if InUnusedStack(address) is false then
12             if line_rd is true then
13                 Checkpoint()
14             else
15                 Evict(line)    // Writeback without a checkpoint
16         else
17             ResetLine(line)          // No need for a writeback
18     UpdateLine(line, type, size)       // Update new cache line
19     return line

20 Procedure UpdateLine(line, type, size) :
21     was-read-dominated ← line_rd
22     if type is Read then
23         line_rd ← true            // Mark line as read-dominated
24     else if type is Write then
25         possible-WAR ← false
26         Set = GetSet(line)     // Get set associated with line
27         for line in Set do           // For each line in the set
28             possible-WAR ← (possible-WAR or line_pw)
29         if (possibe-WAR is false) and (size is 4) then
30             line_rd ← false        // Mark line as write-dominated
31         else
32             line_rd ← true         // Mark line as read-dominated
33     if was-read-dominated then
34         line_pw ← true              // Mark line as possible WAR

35 Procedure Checkpoint() :
36     for line in Cache do        // For each line in the cache
37         if line_dirty then
38             if InUnusedStack(address) is false then
39                 SafeEvict(line)          // Double buffered evict
40         ResetLine(line)    // Clear all the bits in the line
41     CheckpointRegisters()    // Checkpoint CPU registers
```

Checkpoint: The Checkpoint procedure (Line 35) performs a double-buffered [43, Sec. 4.2 "Checkpointing"] write-back to NVM for all the dirty bits in the cache (abstracted for brevity as SafeEvict, Line 39) that are in use (Line 38). After the checkpoint is completed, the data still resides in the cache, but the WAR detection bits are cleared (Line 40) because the detection must be performed from one checkpoint to the next.

5 Implementation

We employ 32-bit RISC-V [32] as the target Instruction Set Architecture (ISA) of NACHO, due to RISC-V's configurability and open-source nature.

5.1 Processor Emulation

To evaluate NACHO, we use ICEmu [55], a cycle-accurate emulator introduced by WARio [37] and specifically designed for intermittent computing frameworks. ICEmu is built on Unicorn [53] CPU instruction set simulator, which in turn is based on QEMU [52]. ICEmu enables the collection of performance metrics as well as the verification of execution correctness, particularly in detecting WAR hazards, for which neither QEMU nor Gem5 [56] is optimized. This makes ICEmu the ideal choice for our evaluation of NACHO along with other state-of-the-art systems (see Section 6.1.) Furthermore, we extended ICEmu to closely represent the *SiFive E21 standard core* processor [63]—by modeling its pipeline [63, Section 3.3]—as this is a basic 32-bit embedded processor.

We chose emulation instead of hardware-based implementation of NACHO. The emulation enables us to evaluate the correctness of NACHO and other systems (introduced in Section 6.1.2), which will be *hard to accomplish with a Microcontroller Unit (MCU) implementation.* This correctness evaluation is done as follows. As the first safety measure, the emulator duplicates the same access to a shadow memory for every memory access generated by the processor. This way, a correct memory access request handled by NACHO must return the same value as contained in the shadow memory. As the second safety measure, the emulator performs WAR detection to verify the absence of any WAR violation, as done in [49, Section 5.2] and [37, Section 5.1.1] by using read- and write-specific address lists and observing memory access patterns. Additionally, emulation allows us to collect detailed metrics without interfering with the program's execution.

5.2 Memory Access Cost Model

For the purpose of this evaluation, we assume a processor speed of 50 MHz. We also assume an access latency of a common onboard NVM of 125 ns [28, Table 1], [22, Section 2]. Furthermore, we assume that a data cache hit induces a two-cycle latency to the pipleline [63, Section 3.1] and a NVM access induces a six-cycle latency (rounded down). Note that all the above values are chosen conservatively[2], as a higher processor speed results in larger differences between the data cache latency and the NVM latency, leading to an even better performance of NACHO than evaluated (see Section 6).

5.3 Cache Controller

We extend ICEmu with non-volatile main memory, and implement NACHO's cache controller as an ICEmu memory subsystem. Within this subsystem, we implement a data cache with *least recently used* replacement policy and four bytes of data per cache line. Moreover, we enable configuring the data cache size and associativity. The additional bits introduced in Section 4 are implemented together with existing data cache bits to emulate a fully functional cache. On every memory access, the algorithm outlined in Algorithm 1 is executed, and the execution pipeline is updated using the cost model given in Section 5.2 to maintain accurate cycle count. To enable stack tracking (Section 4.2.4) the stack pointer is tracked during execution, storing the minimum address since the last checkpoint.

[2]For example, the current state of the art MCUs targetting ultra-low-power applications often operate at speeds over 100 MHz, e.g. [5].

6 Evaluation

We compare NACHO against existing prior works and show that NACHO reduces the number of NVM and cache accesses, thus ensuring energy-efficient execution of intermittently powered applications. We further dissect NACHO's performance to show that NACHO's energy-efficient design choices incur very low computational overhead.

6.1 Evaluation Setup

We compare NACHO against reference systems using multiple benchmark applications and record various performance metrics to show the benefit of NACHO. We begin with the outline of our setup.

6.1.1 Benchmarks. We use CoreMark [18], an industry-grade benchmark for measuring embedded systems' CPU performance, to evaluate NACHO. Additionally, we use the CRC, SHA, Dijkstra and adpcm benchmarks from the MiBench suite [24], towers [41] benchmark, and a quicksort algorithm [50]. Finally, we use TinyAES [35] and picojpeg [21] to represent two real-life embedded applications. All benchmarks are compiled using version 16.0.2 of the `clang` [54] compiler on the RISC-V ISA with optimization level `-Os`.

6.1.2 Systems. We compare NACHO against intermittent computing systems employing NVM as main memory. We ensure that our choice of systems covers both software and hardware support to solve the challenges arising from NVM main memory, as it would help establish the benefit of our system. All of the following systems use double-buffered checkpointing mechanism similar to that of NACHO, to make the comparison as fair as possible.

▶ **Clank [27]:** A memory-tracking hardware module that detects data inconsistencies during execution time. Our implementation is an *ideal* version of Clank, as it does not utilize any memory buffers that can fill up during the WAR detection [27, Section 3.1], nor does it count any memory access cost to these buffers. We considered implementing a version of Clank with a write-through cache to provide more accurate comparison with NACHO. However, the core assumptions of Clank conflict with this idea. Clank relies on immediate memory updates to accurately track WARs, but a cache obfuscates the timing when writes are actually propagated to main memory, delaying the updates that Clank monitors. The presence of a cache fundamentally disrupts the direct memory access assumptions on which Clank's tracking mechanism is based. Since NACHO also performs WAR detection (but using just the data cache), we use Clank without any cache to compare it with NACHO.

▶ **PROWL [28]:** A cache implementation reducing NVM accesses, which avoids frequent checkpoints due to WARs by employing a custom cache replacement policy that delays the eviction of a dirty cache block. We include PROWL as a reference, as it relates most closely to NACHO, as they both introduce data cache modifications for intermittent systems.

▶ **ReplayCache [73]:** Uses a non-blocking cache to asynchronously write cache lines back to NVM. We include ReplayCache as a reference for its JIT-checkpointing strategy. Unfortunately, ReplayCache source code was not made public and was refused to be shared by its authors. Therefore we have reverse engineered the ReplayCache implementation based on (i) the sparse data provided in the ReplayCache paper and (ii) numerous email conversations with the ReplayCache authors. As a result of our re-implementation efforts we found limiting features of ReplayCache that were not mentioned in the original RepalyCache paper [73]. Firstly, RepalyCache uses *miss status/handler registers for coalescing stores in the write-back queue*. This makes any real-world implementation of ReplayCache more complex since it requires specialized cache hardware. Secondly, although ReplayCache's original implementation is based on Advanced RISC Machines (ARM) architecure, it **cannot be implemented on a real ARM system**. This is because ARM architecure does not allow adding custom flags to instructions, resulting in the original ReplayCache implementation storing all region boundaries in the simulator. In our implementation of ReplayCache—to make a fair comparison between ReplayCache and NACHO—we use RISC-V architecture instead of ARM. As result of our efforts we release the source code of our ReplayCache implementation together with the NACHO source code [50].

▶ **Naive NACHO:** A basic version of NACHO, as described in Section 3.1, that does not have a WAR detector and no stack tracking support. The use of naive NACHO helps us dissect the performance gains achieved by each component of the complete NACHO implementation.

▶ **Oracle NACHO:** An ideal version of NACHO that acts as its theoretical lower bound. The key difference between Oracle NACHO and NACHO lies in the detection of WARs based on the cache line addresses. While NACHO detects WAR using read/write-dominated cache lines, Oracle NACHO makes this detection using exact addresses, thus making it a perfect WAR violation detector. However, implementing such a system increases the hardware cost and complexity, thus making it impractical to implement.

6.1.3 Metrics.

We consider *four* evaluation metrics: (1) *Execution time:* the time required to complete a given benchmark; (2) *Checkpoints:* the number of times the device had to checkpoint its state; (3) *Number of NVM transfers:* the number of bytes read/written from/to NVM during program execution; (4) *Intermittent (re-execution) execution overhead:* the percentage of computational overhead incurred when running on an intermittent energy system compared to a fully volatile system.

6.1.4 Changing parameters.

We consider *two* variable parameters to evaluate the execution time metric:

▶ **Power failures:** periodic power failures at intervals of 5 ms, 10 ms, 50 ms and 100 ms. Power failure experiments are only discussed in Section 6.2.4, all other results do not use intermittent power failures.

▶ **Cache:** 2-way caches of size 256 and 512 bytes (as used in [28]) are considered for the evaluation of execution time. For the cache design space exploration, an additional size of 1024 bytes is used, and all different cache sizes are evaluated using 4-way caches as well.

6.2 Evaluation Results

We now proceed with the evaluation of NACHO. Of note is that, for Figures 6 and 7, the adpcm and quicksort benchmarks are not included because their result is very similar to SHA and CRC respectively, while towers is not included because Clank and Oracle NACHO do not generate any checkpoints in that benchmark.

6.2.1 Execution Time.

Figure 5 shows the execution time of all systems for each of the benchmarks considered for two different cache configurations, normalized to a system with *fully volatile memory* of the respective benchmark. Note that this volatile memory system does not support intermittent computing and assumes the same memory technology for the main memory as used for the data cache. We can see that the mean normalized execution time for NACHO is 76% and 79% lower compared to Clank when using a 256 B and 512 B data cache, respectively. When compared against PROWL, NACHO's execution time is 46% and 38% lower for, respectively, 256 B and 512 B data cache (using the same cache size for both systems). The mean execution time for NACHO compared to ReplayCache is 18% and 28% lower for cache sizes 256 B and 512 B, again using the same cache sizes for both systems. We note that ReplayCache is faster on towers for a 256 B data cache by a small margin, because of fewer accesses to NVM compared to NACHO. Excluding the towers outlier, NACHO is, on average, within 4% and 2% of Oracle NACHO's execution time when using a 256 B and 512 B cache, respectively. The difference between NACHO and Oracle NACHO in the towers benchmark occurs because Oracle NACHO does not generate any checkpoints.

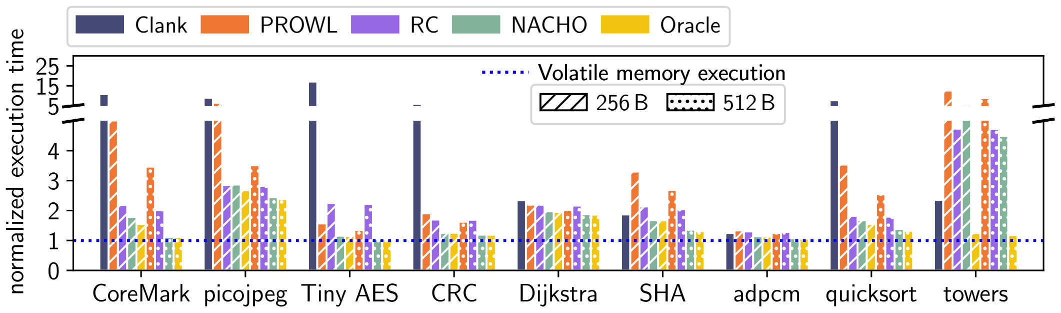

Figure 5. Execution time for all benchmarks for Clank [27], PROWL [28], ReplayCache (RC) [73], NACHO, and Oracle NACHO (Oracle). All results are normalized to the execution time of a system without non-volatile main memory and intermittent computing support. Oracle NACHO is shown as the hypothetical lower bound that NACHO could reach if NACHO utilized perfect memory tracking. The cache configuration used is a 2-way set-associative for two cache sizes: 256 B and 512 B. Note that Clank is a cacheless system and is thus not affected by cache configuration.

If we further dissect the numbers by removing the baseline program execution cost from all benchmarks, the overhead becomes even more apparent. Compared to PROWL, NACHO's average overhead is 65% lower with 512 B cache, with a maximum overhead reduction of 95% (CoreMark). NACHO's average overhead is also 55% lower than ReplayCache with 512 B cache, and a maximum overhead reduction of 96% (TinyAES). Lower execution times for 512 B cache size are due to the cache's ability to retain more addresses, thus delaying the eviction, as explained in Section 6.2.2.

6.2.2 Number of Checkpoints. Figure 6 shows a significant decrease in the number of checkpoints of both PROWL and NACHO compared to Clank. ReplayCache is excluded since it does not generate checkpoints when there are no intermittent power failures. It must be noted that a checkpoint in Clank only consists of the registers, whereas in both PROWL and NACHO, the cache has to be written back to the NVM in a double-buffered manner during a checkpoint, making it significantly more costly. In other words, even though NACHO had a larger update size at the time of checkpoint, NACHO is able to significantly reduce the need for checkpoints due to its efficient detection of idempotence violations. Additionally, we can see a decrease in the number of checkpoints for a larger cache size for all the systems. This is primarily because of the ability of the cache to retain more data, thus reducing the need for eviction and, in turn, the checkpoint. We evaluate this effect further in Section 6.2.6.

6.2.3 Number of NVM Transfers. Figure 7 shows the number of bytes transferred during *read* and *write* operations normalized to the numbers reported for Clank (which exclusively uses NVM). We can observe that NACHO significantly reduces the number of NVM transfers for almost all benchmarks compared to Clank and PROWL, with **99% reduction** for TinyAES being the maximum and the trend holds for most benchmarks. On average, NACHO reduces

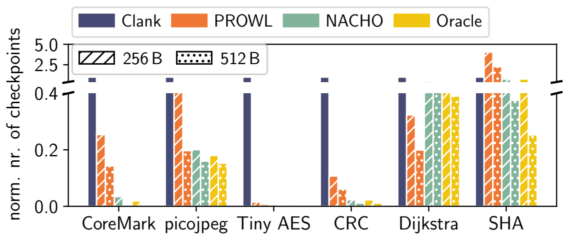

Figure 6. Number of checkpoints created, normalized to Clank with configurations identical to Figure 5. ReplayCache is excluded because it did not generate any checkpoints.

the number of NVM transfers by 82% and 50% compared to Clank and PROWL, respectively. This shows the efficiency gain of NACHO compared to Clank and PROWL, as NVM transfers are typically energy-demanding. While this trend holds for most benchmarks, Dijkstra is an extreme outlier as NACHO has 23% more NVM transfers than PROWL, most likely caused by an unfortunate cache access pattern that causes many checkpoints in NACHO, so benefiting from the PROWL cache relocation strategy. We also observe that the number of transfers in ReplayCache is smaller for picojpeg, Dijkstra and SHA benchmarks compared to NACHO, because NACHO transfers more bytes to NVM due to the checkpoints that it generates.

6.2.4 Re-execution Overhead. An important metric to evaluate for any intermittently powered device is the cost of re-execution, i.e., the cost associated with a power failure. On every power failure, checkpointed system state must be restored whenever energy is available to resume application execution. Resuming execution incurs an additional cost as the data cache loses its content after a power failure, resulting in cache misses. Furthermore, checkpoints are not created precisely before a power failure occurs, so there is a limited amount of *code re-execution* that adds to the computational

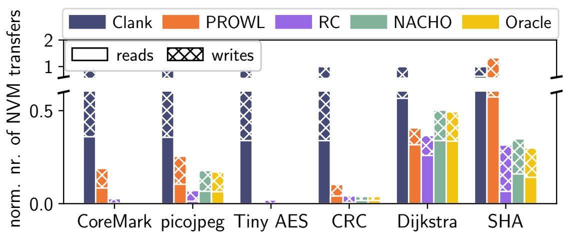

Figure 7. Number of non-volatile memory byte transfers. All results are normalized to Clank. PROWL, ReplayCache and NACHO are configured with a 512 B data cache.

Table 2. Re-execution overhead of NACHO, running at 50 MHz, compared to NACHO without intermittent power failures. The overhead consists of periodic checkpoints with half the period of the on-duration (to guarantee forward progress) and the re-execution cost of power failures. Some benchmarks are excluded because the overhead in those cases is smaller than that of the benchmarks in the table.

On-duration	CoreMark	picojpeg	Tiny AES	SHA	adpcm
5 ms	9.68%	0.95%	16.15%	0.86%	4.35%
10 ms	2.47%	0.42%	5.54%	0.73%	4.35%
50 ms	0.14%	0.11%	1.06%	0.19%	0.99%
100 ms	0.00%	0.07%	0.74%	0.13%	0.17%

overhead. Lastly, to ensure forward progress, the system must introduce periodic checkpoints to guarantee that at least one checkpoint is created during the on-duration.

Table 2 shows the re-execution cost with different power interruption intervals (on-duration). The shorter the power interruption interval, the higher the cost of re-execution to complete the workload. For every on-duration n, we configure a periodic checkpoint to occur every $n/2$ ms to guarantee forward progress. We observe that even with the shortest power interruption interval, the additional cost is less than 4% on average for all benchmarks (including those not shown in Table 2), with CoreMark and TinyAES being the outliers. With more reasonable interruption intervals, such as a power failure every 50 ms, we can see that the average additional cost is less than 1.1%. The low overhead can be attributed to 50 ms being still a considerable amount of clock cycles and memory accesses, masking the cost of some additional periodic checkpoints, refilling the cache, and re-execution.

6.2.5 NACHO's Components Evaluation.

Table 3 shows the percentage reduction achieved by WAR violation detection (PW) and stack-tracking approaches (ST) individually as well as the NACHO (N) overall. We see that for all benchmarks and considered metrics, the overall improvement of NACHO over Naive NACHO is significant, with an average overhead reduction of 18% and an average reduction in the number of NVM writes of 19%. It must be noted that the reduction achieved by each component individually can not be summed to the overall reduction achieved, as both techniques (WAR violation detection and stack-tracking) can target similar memory access patterns.

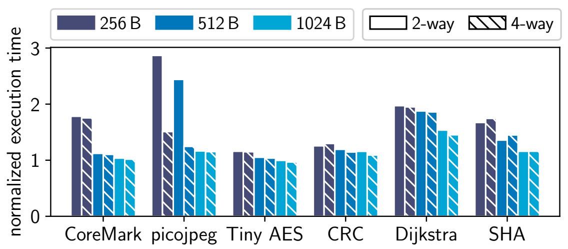

Figure 8. Cache configurations design space exploration of NACHO normalized to a system with only volatile memory.

6.2.6 Design Space Exploration.

Figure 8 shows NACHO's execution time in different cache configurations.

Cache Size: As we have shown, increasing the data cache size improves NACHO performance as a larger data cache can store more dirty blocks, effectively increasing the time between WARs and, therefore, checkpoints. Also, a larger data cache creates smaller mappings between cache lines and program memory, which gives higher accuracy to per-line WAR detection. However, as can be seen in Figure 8, for most considered benchmarks, the jump in performance between a data cache size of 512 B and 1024 B is not as significant as the jump from 256 B to 512 B, suggesting diminishing returns as the cache size increases.

Cache Associativity: A higher cache associativity implies that the cache can store more blocks before it evicts to make space. Similar to cache size, the cache associativity also improves NACHO's performance as increasing the cache associativity decreases the probability of a cache collision. However, the increase in performance due to the increase in associativity is not as significant as it is with the change in size (it even reduced the performance in the case of SHA). This is because NACHO must consider all blocks associated with a hash, reducing the benefits of higher associativity.

We conclude that, with NACHO, a 2-way set associative data cache is preferred over a more complicated 4-way cache implementation. The marginal increase in performance when utilizing a 4-way cache does not outweigh the additional complexity. Hence, during our evaluation, we configured NACHO with a 2-way set associative data cache[3].

[3] Another reason is to aid comparisons against PROWL, which only provides hashing functions for a 2-way set associative data cache.

Table 3. Percental *reduction* of selected metrics compared to *Naive NACHO* for the two individual NACHO components, possible war (**PW**) and stack-tracking (**ST**), and finally the complete system—NACHO (**N**).

	CoreMark			picojpeg			Tiny AES			CRC			Dijkstra			SHA		
Metric	**PW**	**ST**	**N**	**PW**	**ST**	**N**	**PW**	**ST**	**N**	**PW**	**ST**	**N**	**PW**	**ST**	**N**	**PW**	**ST**	**N**
overhead	12%	4%	**16%**	3%	6%	**7%**	0%	9%	**9%**	12%	22%	**22%**	2%	0%	**2%**	39%	33%	**52%**
checkpoints	20%	0%	**20%**	3%	3%	**4%**	0%	0%	**0%**	25%	25%	**25%**	4%	0%	**4%**	41%	21%	**41%**
NVM reads	9%	3%	**12%**	3%	5%	**6%**	0%	4%	**4%**	10%	17%	**17%**	1%	0%	**1%**	34%	28%	**44%**
NVM writes	13%	4%	**17%**	3%	6%	**7%**	0%	5%	**5%**	13%	24%	**24%**	4%	0%	**4%**	42%	36%	**57%**

7 Related Work

Tasks/Threads: A volatile system requires frequent checkpoints during program execution to cross periods of energy unavailability [2, 10, 61]. Checkpointing is an energy-hungry operation and adds to the execution time of an application. An alternative to checkpoints, a task- or thread-based programming model [8, 13, 43, 45, 68, 71, 72], reduces the cost of checkpointing by employing a mixed volatility memory system. It exposes Application Programming Interfaces (APIs), allowing the programmer (or a compiler) to divide a program into a set of atomic tasks and perform lightweight checkpoints at the end of each task to ensure a persistent application state across reboots.

Energy-efficient Program Execution: Work of [4] proposes a dynamic voltage and frequency scaling. This reduces the cost of program execution by allowing an intermittent system to dynamically regulate its operating voltage based on the changing energy conditions.

Static Volatile Memory Mapping: A virtual memory manager maps data either to volatile or non-volatile memory during compilation [48]. However, this approach allows only a limited number of accesses to be made volatile.

8 Discussion and Future Work

Limitations and Mitigations: While NACHO is designed to operate within the constraints of batteryless, energy harvesting intermittent systems, certain limitations arise from the assumptions outlined earlier in Section 1. These limitations define the scope of applicability of NACHO and highlight potential challenges on extending this approach to systems with different deployment constraints and cache configurations. By defining these limitations and potential approaches to address them, we provide a foundation for future research directions in intermittent computing. Firstly, NACHO is cache agnostic in terms of WAR detection, but its implementation assumes a write-back cache model. For write-through caches, the implementation needs to be modified to adjust cache eviction policies to align with checkpointing behavior. Secondly, our assumption of cache sizes and associated energy consumption might falter with the high pace of cache architecture innovation [33]. As cache size increases, the potential checkpoint size also grows, increasing the energy required to persist cache state before power failure. To mitigate this, *adaptive checkpointing policies* [6, 51] could be explored, where a system dynamically decides when to flush based on an upper-bound threshold of dirty cache lines. Such designs could also implement *cache-aware energy budgeting mechanisms*, ensuring that placement of checkpoints account for both cache occupancy and available energy capacity.

Integration with Other Systems: NACHO takes a different approach than, e.g., PROWL [28], ReplayCache [73], SweepCache [75], WL-Cache [11] and NvMR [9]. Even though all these systems utilize a data cache, their cache is not tightly integrated with WAR violation detection. PROWL, SweepCache, WL-Cache and ReplayCache do not use WAR violation detection, and NvMR focuses on renaming NVM accesses to avoid WAR violations as much as possible and uses a detection mechanism similar to Clank [27]. These systems could benefit from incorporating NACHO's WAR detection techniques into the existing data cache, although this is not trivial and requires further research.

Energy and Area Costs: Because NACHO is implemented as a memory subsystem in an emulator, precise and accurate assessments of area and energy overheads are challenging. Nonetheless, we discus the relevant parameters that influence the power, energy, and area costs of NACHO. *Energy Overhead:* While NACHO minimizes non-volatile memory writes, which are the most energy consuming operations, NACHO does introduce extra power consumption and area overheads. The primary energy overhead arises from cache accesses, where the additional two bits per cache line track WAR dependencies. However, this is a minor cost compared to the significant energy savings achieved by avoiding unnecessary NVM writes. *Cache Area:* The area overhead is minimal as the additional bits integrate into existing cache metadata that eliminates the need for separate WAR tracking hardware. Larger caches may lead to higher checkpointing energy requirements, but this can be managed by adjusting capacitor size or implementing adaptive checkpointing strategies as discussed earlier. Finally we recall that our experiments generate additional statistics (see Appendix A.5 and Appendix A.6 for steps to reproduce and evaluate our results) on various metrics such as checkpointing frequency, average size of a checkpoint, and cache utilization. Our evaluation in Section 6.2 already accounts for the differences in memory accesses and can be further extended to these additional metrics to construct a rough energy model.

Energy Prediction: NACHO is incorruptible through double buffering. However, if the system can guarantee that enough energy is available to complete the cache writeback and register checkpoint, double buffering is not needed, halving the number of NVM writes during a checkpoint.

Peripherals: NACHO does not focus on supporting peripherals or other input/output operations needed to communicate with sensors and actuators. Integrating prior works such as [36, 47, 70] into NACHO is a nontrivial topic and requires further research.

9 Conclusions

We presented NACHO system, where a data cache is coupled with the intermittent computing paradigm. NACHO, with the cache as a WAR violation detection entity, removes the need for additional memory tracking. NACHO achieves better performance than the state of the art solutions by reducing both the number of required checkpoints and accesses to slow, non-volatile memory.

10 Acknowledgements

We thank our anonymous reviewers and our shepherd Prof. John (Jack) Sampson for their valuable comments. This research was supported by the Netherlands Organisation for Scientific Research (NWO), partly funded by the Dutch Ministry of Economic Affairs, through TTW Perspective program ZERO (P15-06) within Project P4.

References

[1] Kofi Sarpong Adu-Manu, Nadir Adam, Cristiano Tapparello, Hoda Ayatollahi, and Wendi Heinzelman. 2018. Energy-harvesting Wireless Sensor Networks (EH-WSNs): A Review. *ACM Trans. Sens. Netw.* 14, 2 (July 2018), 10:1–10:50. https://doi.org/10.1145/3183338.

[2] Saad Ahmed, Naveed Anwar Bhatti, Muhammad Hamad Alizai, Junaid Haroon Siddiqui, and Luca Mottola. 2019. Effient Intermittent Computing with Differential Checkpointing. In *Proc. LCTES*. ACM, Phoenix, AZ, USA, 70–81. https://doi.org/10.1145/3316482.3326357.

[3] Saad Ahmed, Bashima Islam, Kasım Sinan Yıldırım, Marco Zimmerling, Przemysław Pawełczak, Muhammad Hamad Alizai, Brandon Lucia, Luca Mottola, Jacob Sorber, and Josiah Hester. 2024. The Internet of Batteryless Things. *Commun. ACM* 67, 3 (2024), 64–73. https://doi.org/10.1145/3624718.

[4] Saad Ahmed, Qurat ul Ain, Junaid Haroon Siddiqui, Luca Mottola, and Muhammad Hamad Alizai. 2020. Intermittent Computing with Dynamic Voltage and Frequency Scaling. In *Proc. EWSN* (Feb. 17–19). ACM, Lyon, France, 97–107. https://doi.org/10.5555/3400306.3400319.

[5] Ambiq Micro Inc. 2022. Apollo4 Ultra-Low Power Microcontroller. https://ambiq.com/apollo4/. Last accessed: Oct. 17, 2022.

[6] Mohsen Ansari, Sepideh Safari, Heba Khdr, Pourya Gohari-Nazari, Jörg Henkel, Alireza Ejlali, and Shaahin Hessabi. 2022. Power-Aware Checkpointing for Multicore Embedded Systems. *IEEE Trans. Parallel Distrib. Syst.* 33, 12 (Dec. 2022), 4410–4424. https://doi.org/10.1109/TPDS.2022.3188568.

[7] Rauf Arif. 2021. With An Economic Potential Of $11 Trillion, Internet Of Things Is Here To Revolutionize Global Economy. Forbes, https://www.forbes.com/sites/raufarif/2021/06/05/with-an-economic-potential-of-11-trillion-internet-of-things-is-here-to-revolutionize-global-economy. Last accessed: Oct. 14, 2022.

[8] Abu Bakar, Alexander G. Ross, Kasım Sinan Yıldırım, and Josiah Hester. 2021. REHASH: A Flexible, Developer Focused, Heuristic Adaptation Platform for Intermittently Powered Computing. *ACM Interact. Mob. Wearable Ubiquitous Technol.* 5, 3 (Sept. 2021), 87:1–87:42. https://doi.org/10.1145/3478077.

[9] Abhishek Bhattacharyya, Abhijith Somashekhar, and Joshua San Miguel. 2022. NvMR: Non-Volatile Memory Renaming for Intermittent Computing. In *Proc. ISCA* (June 18–22). ACM, New York, NY, USA, 1–13. https://doi.org/10.1145/3470496.3527413.

[10] Naveed Anwar Bhatti and Luca Mottola. 2017. HarvOS: Efficient Code Instrumentation for Transiently-Powered Embedded Sensing. In *Proc. IPSN*. ACM/IEEE, Pittsburgh, PA, USA, 209–219. https://doi.org/10.1145/3055031.3055082.

[11] Jongouk Choi, Jianping Zeng, Dongyoon Lee, Changwoo Min, and Changhee Jung. 2023. Write-Light Cache for Energy Harvesting Systems. In *Proc. ISCA*. ACM, Orlando, FL, USA, Article 63, 13 pages. https://doi.org/10.1145/3579371.3589098.

[12] Clare Church and Laurin Wuennenberg. 2019. Sustainability and Second Life: The Case for Cobalt and Lithium Recycling. Retrieved Oct. 14, 2022 from https://www.iisd.org/publications/sustainability-and-second-life-case-cobalt-and-lithium-recycling

[13] Alexei Colin and Brandon Lucia. 2016. Chain: Tasks and Channels for Reliable Intermittent Programs. In *Proc. OOPSLA* (Oct. 30 – Nov. 4). ACM, Amsterdam, The Netherlands, 514–530. https://doi.org/10.1145/2983990.2983995.

[14] Andy Kong Daehwa, Kim Chris, and Harrison. 2024. Power-over-Skin: Full-Body Wearables Powered By Intra-Body RF Energy. In *Proc. UIST*. ACM, Pittsburgh, PA, USA, 3:1–3:13. https://doi.org/10.1145/3654777.3676394.

[15] Tuan Dang, Trung Tran, Khang Nguyen, Tien Pham, Nhat Pham, Tam Vu, and Phuc Nguyen. 2022. IoTree: A Battery-free Wearable System with Biocompatible Sensors for Continuous Tree Health Monitoring. In *Proc. MobiCom* (Oct. 17–21). ACM, Sydney, NSW, Australia, 352–366. https://doi.org/10.1145/3495243.3558749.

[16] Jasper de Winkel, Carlo Delle Donne, Kasım Sinan Yıldırım, Przemysław Pawełczak, and Josiah Hester. 2020. Reliable Timekeeping for Intermittent Computing. In *Proc. ASPLOS* (March 16–20). ACM, Lausanne, Switzerland, 53–67. https://doi.org/10.1145/3373376.3378464.

[17] Jasper de Winkel, Vito Kortbeek, Josiah Hester, and Przemysław Pawełczak. 2020. Battery-Free Game Boy. *ACM Interact. Mob. Wearable Ubiquitous Technol.* 4, 3 (Sept. 2020), 111:1–111:34. https://doi.org/10.1145/3411839.

[18] Embedded Microprocessor Benchmark Consortium. 2018. CoreMark Benchmark. https://github.com/eembc/coremark/releases/tag/v1.01. Last accessed: Oct. 16, 2022.

[19] Laura Marie Feeney, Christian Rohner, Per Gunningberg, Anders Lindgren, and Lars Andersson. 2014. How do the Dynamics of Battery Discharge Affect Sensor Lifetime?. In *Proc. Annual Conference on Wireless On-demand Network Systems and Services* (April 2–4). IEEE, Obergurgl, Austria, 49–56. https://doi.org/10.1109/WONS.2014.6814721.

[20] Kai Geissdoerfer and Marco Zimmerling. 2024. Riotee: An Open-source Hardware and Software Platform for the Battery-free Internet of Things. In *Proc. SenSys*. ACM, Hangzhou, China, 198–210. https://doi.org/10.1145/3666025.3699332.

[21] Rich Geldreich. 2022. Picojpeg. https://github.com/richgel999/picojpeg. Last accessed: Oct. 17, 2022.

[22] William Goh, Andreas Dannenberg, and Johnson He. 2021. Texas Instruments MSP430 FRAM Technology – How To and Best Practices.

https://www.ti.com/lit/an/slaa628b/slaa628b.pdf. Last accessed: Oct. 17, 2022.

[23] Rabeeh Golmohammadzadeh, Fariborz Faraji, Brian Jong, Cristina Pozo-Gonzalo, and Parama Chakraborty Banerjee. 2022. Current Challenges and Future Opportunities Toward Recycling of Spent Lithium-Ion Batteries. *Renewable Sustainable Energy Rev.* 159 (May 2022), 112202:1–112202:24. https://doi.org/10.1016/j.rser.2022.112202.

[24] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. 2001. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *Proc. Workload Characterization Workshop* (Dec. 2). IEEE, Austin, TX, USA, 3–14. https://doi.org/10.1109/wwc.2001.990739.

[25] John L. Hennessy and David A. Patterson. 2012. *Computer Architecture: A Quantitative Approach (Fifth Edition)*. Morgan Kaufman, Burlington, MA, USA.

[26] Josiah Hester and Jacob Sorber. 2017. The Future of Sensing is Battery-less, Intermittent, and Awesome. In *Proc. SenSys* (Nov. 6–8). ACM, Delft, The Netherlands, 21:1–21:6. https://doi.org/10.1145/3131672.3131699.

[27] Matthew Hicks. 2017. Clank: Architectural Support for Intermittent Computation. In *Proc. ISCA* (June 24–28). ACM, Toronto, ON, Canada, 228–240. https://doi.org/10.1145/3140659.3080238.

[28] Ali Hoseinghorban, Mohammad Abbasinia, and Alireza Ejlali. 2020. PROWL: A Cache Replacement Policy for Consistency Aware Renewable Powered Devices. *IEEE Trans. Emerging Top. Comput.* 10, 1 (Jan.-Mar. 2020), 476–487. https://doi.org/10.1109/TETC.2020.3031114.

[29] Ali Hoseinghorban, Amir Mahdi Hosseini Monazzah, Mostafa Bazzaz, Bardia Safaei, and Alireza Ejlali. 2021. COACH: Consistency Aware Check-Pointing for Nonvolatile Processor in Energy Harvesting Systems. *IEEE Trans. Emerging Top. Comput.* 9, 4 (Oct. 2021), 2076–2088. https://doi.org/10.1109/TETC.2019.2961007.

[30] Texas Instruments. 2021. Texas Instruments MSP430FR599x, MSP430FR596x Mixed-Signal Microcontrollers. https://www.ti.com/lit/ds/symlink/msp430fr5994.pdf. Last accessed: Oct. 19, 2022.

[31] Intel Corp. 2022. Intel 64 and IA-32 Architectures Software Developer's Manual; Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4. https://cdrdv2.intel.com/v1/dl/getContent/671200.

[32] RISC-V International. 2022. Official RISC-V Website. https://riscv.org/about/. Last accessed: Oct. 17, 2022.

[33] Ravi Iyer, Vivek De, Ramesh Illikkal, David Koufaty, Bhushan Chitlur, Andrew Herdrich, Muhammad Khellah, Fatih Hamzaoglu, and Eric Karl. 2021. Advances in Microprocessor Cache Architectures Over the Last 25 Years. *IEEE Micro* 41, 6 (Nov.-Dec. 2021), 78–88. https://doi.org/10.1109/MM.2021.3114903.

[34] Mohsen Karimi, Hyunjong Choi, Yidi Wang, Yecheng Xiang, and Hyoseung Kim. 2021. Real-Time Task Scheduling on Intermittently Powered Batteryless Devices. *IEEE Internet Things J.* 8, 17 (Sept. 2021), 13328–13342. https://doi.org/10.1109/JIOT.2021.3065947.

[35] kokke. 2022. Tiny AES in C. https://github.com/kokke/tiny-AES-c. Last accessed: Oct. 17, 2022.

[36] Vito Kortbeek, Abu Bakar, Stefany Cruz Kasım Sinan Yıldırım, Przemysław Pawełczak, and Josiah Hester. 2020. BFree: Enabling Battery-free Sensor Prototyping with Python. *ACM Interact. Mob. Wearable Ubiquitous Technol.* 4, 4 (Dec. 2020), 135:1–111:39. https://doi.org/10.1145/3432191.

[37] Vito Kortbeek, Souradip Ghosh, Josiah Hester, Simone Campanoni, and Przemysław Pawełczak. 2022. WARio: Efficient Code Generation for Intermittent Computing. In *Proc. PLDI* (June 13–17). ACM, San Diego, CA, USA, 777–791. https://doi.org/10.1145/3519939.3523454.

[38] Vito Kortbeek, Kasım Sinan Yıldırım, Abu Bakar, Jacob Sorber, Josiah Hester, and Przemysław Pawełczak. 2020. Time-sensitive Intermittent Computing Meets Legacy Software. In *Proc. ASPLOS* (March 16–20). ACM, Lausanne, Switzerland, 85–99. https://doi.org/10.1145/3373376.3378476.

[39] Christopher Kraemer, William Gelder, and Josiah Hester. 2024. User-directed Assembly Code Transformations Enabling Efficient Battery-less Arduino Applications. *ACM Interact. Mob. Wearable Ubiquitous Technol.* 8, 2 (May 2024), 44:1–44:32. https://doi.org/10.1145/3659590.

[40] Fredrik Larsson and Bengt-Erik Mellander. 2014. Abuse by External Heating, Overcharge and Short Circuiting of Commercial Lithium-Ion Battery Cells. *J. Electrochem. Soc.* 161, 10 (2014), A1611–A1617. https://doi.org/10.1149/2.0311410jes.

[41] Eric Love. 2016. Towers. https://github.com/ucb-bar/riscv-benchmarks/blob/master/towers/towers_main.c. Last accessed: Oct. 17, 2022.

[42] Brandon Lucia, Vignesh Balaji, Alexei Colin, Kiwan Maeng, and Emily Ruppel. 2017. Intermittent Computing: Challenges and Opportunities. In *Proc. SNAPL*. Schloss Dagstuhl, Alisomar, CA, USA, 8:1–8:14. https://drops.dagstuhl.de/opus/volltexte/2017/7131/pdf/LIPIcs-SNAPL-2017-8.pdf.

[43] Brandon Lucia and Benjamin Ransford. 2015. A Simpler, Safer Programming and Execution Model for Intermittent Systems. In *Proc. PLDI* (Aug. 13–17). ACM, Portland, OR, USA, 575–585. https://doi.org/10.1145/2737924.2737978.

[44] Kaisheng Ma, Xueqing Li, Karthik Swaminathan, Yang Zheng, Shuangchen Li, Yongpan Liu, Yuan Xie, John Jack Sampson, and Vijaykrishnan Narayana. 2016. Nonvolatile Processor Architectures: Efficient, Reliable Progress with Unstable Power. *IEEE Micro* 36, 3 (May–Jun. 2016), 72–83. https://doi.org/10.1109/MM.2016.35.

[45] Kiwan Maeng, Alexei Colin, and Brandon Lucia. 2017. Alpaca: Intermittent Execution without Checkpoints. In *Proc. OOPSLA* (Oct. 22–27). ACM, Vancouver, BC, Canada, 96:1–96:30. https://doi.org/10.1145/3133920.

[46] Kiwan Maeng, Alexei Colin, and Brandon Lucia. 2018. Adaptive Dynamic Checkpointing for Safe Efficient Intermittent Computing. In *Proc. OSDI* (Oct. 8–10). USENIX, Carlsbad, CA, USA, 129–144. https://www.usenix.org/system/files/osdi18-maeng.pdf.

[47] Kiwan Maeng and Brandon Lucia. 2019. Supporting Peripherals in Intermittent Systems with Just-in-Time Checkpoints. In *Proc. PLDI* (June 22–26). ACM, Phoenix, AZ, USA, 1101–1116. https://doi.org/10.1145/3314221.3314613.

[48] Andrea Maioli and Luca Mottola. 2021. ALFRED: Virtual Memory for Intermittent Computing. In *Proc. SenSys* (Nov. 15–17). ACM, Coimbra, Portugal, 261–273. https://doi.org/10.1145/3485730.3485949.

[49] Andrea Maioli, Luca Mottola, Muhammad Hamad Alizai, and Junaid Haroon Siddiqui. 2021. Discovering the Hidden Anomalies of Intermittent Computing. In *Proc. EWSN* (Feb. 17–19). ACM, Delft, The Netherlands, 1–12. https://dl.acm.org/doi/10.5555/3451271.3451272.

[50] Sourav Mohapatra, Vito Kortbeek, Marco A. van Eerden, and Jochem Broekhoff. 2022. NACHO Artifact Repository: Software and Documentation. https://github.com/TUDSSL/intermittent-risc-v. (`nacho-artifact-release` branch) Last accessed: Feb. 10, 2025.

[51] Muhammad Mudassar, Yanlong Zhai, and Liao Lejian. 2022. Adaptive Fault-Tolerant Strategy for Latency-Aware IoT Application Executing in Edge Computing Environment. *IEEE Internet Things J.* 9, 15 (Aug. 2022), 13250–13262. https://doi.org/10.1109/JIOT.2022.3144026.

[52] Open Source Community Contributors. 2021. QEMU: a Generic and Open Source Machine and Userspace Emulator and Virtualizer. https://gitlab.com/qemu-project/qemu. Last accessed: Oct. 17, 2022.

[53] Open Source Community Contributors. 2021. Unicorn: a Lightweight, Multi-platform, Multi-architecture CPU Emulator Framework based on QEMU. https://github.com/unicorn-engine/unicorn. Last accessed: Oct. 16, 2022.

[54] Open Source Community Contributors. 2022. Clang: a C language family frontend for LLVM. https://clang.llvm.org/. Last accessed: Oct. 17, 2022.

[55] Open Source Community Contributors. 2022. ICEmu: Intermittent Computing Emulator. https://github.com/tudssl/ICEmu/. Last accessed: Oct. 17, 2022.

[56] Open Source Community Contributors. 2024. Gem5: The gem5 Simulator. https://github.com/gem5/gem5. Last accessed: Oct. 2, 2024.

[57] R. Venkatesha Prasad, Shruti Devasenapathy, Vijay S. Rao, and Javad Vazifehdan. 2014. Reincarnation in the Ambiance: Devices and Networks with Energy Harvesting. *IEEE Commun. Surveys Tuts.* 11, 1 (First Quarter 2014), 195–213. https://doi.org/10.1109/SURV.2013.062613.00235.

[58] Phillip Raffeck, Johannes Maier, and Peter Wägemann. 2024. WoCA: Avoiding Intermittent Execution in Embedded Systems by Worst-Case Analyses with Device States. In *Proc. LCTES*. ACM, Copenhagen, Denmark, 83–94. https://doi.org/10.1145/3652032.3657569.

[59] Vijay Raghunathan, Aman Kansal, Jason Hsu, Jonathan Friedman, and Mani Srivastava. 2005. Design Considerations for Solar Energy Harvesting Wireless Embedded Systems. In *Proc. IPSN*. ACM/IEEE, Boise, ID, USA, 457–462. https://doi.org/10.1109/IPSN.2005.1440973.

[60] Benjamin Ransford. 2011. Traces repository used in 'Mementos: System Support for Long-running Computation on RFID-scale Devices' paper. https://github.com/ransford/mspsim/tree/mementos/traces. Last accessed: Oct. 14, 2022.

[61] Benjamin Ransford, Jacob Sorber, and Kevin Fu. 2011. Mementos: System Support for Long-running Computation on RFID-scale Devices. In *Proc. ASPLOS* (March 5–11). ACM, Newport Beach, CA, USA, 159–170. https://doi.org/10.1145/1950365.1950386.

[62] Esther Shein. 2021. A Battery-Free Internet of Things. *Commun. ACM* 64, 7 (2021), 16–18. https://doi.org/10.1145/3464937.

[63] SiFive, Inc. 2021. SiFive E21 Core Complex Manual 21G3.02.00. https://sifive.cdn.prismic.io/sifive/6f0f1515-1249-4ea0-a5b2-79d4aaf920ae_e21_core_complex_manual_21G3.pdf. Last accessed: Oct. 16, 2022.

[64] Fang Su, Kaisheng Ma, Xueqing Li, Tongda Wu, Yongpan Liu, and Vijaykrishnam Narayanan. 2017. Nonvolatile Processors: Why is it Trending?. In *Proc. DATE*. ACM/IEEE, Lausanne, Switzerland, 966–971. https://doi.org/10.23919/DATE.2017.7927131.

[65] Vamsi Talla, Bryce Kellogg, Shyamnath Gollakota, and Joshua R Smith. 2017. Battery-free Cellphone. *ACM Interact. Mob. Wearable Ubiquitous Technol.* 1, 2 (June 2017), 25:1–25:19. https://doi.org/10.1145/3090090.

[66] Joel Van Der Woude and Matthew Hicks. 2016. Intermittent Computation Without Hardware Support or Programmer Intervention. In *Proc. OSDI* (Nov. 2–4). ACM, Savannah, GA, USA, 17–32. https://www.usenix.org/system/files/conference/osdi16/osdi16-van-der-woude.pdf.

[67] Harrison Williams and Matthew Hicks. 2024. Energy-Adaptive Buffering for Efficient, Responsive, and Persistent Batteryless Systems. In *Proc. ASPLOS*. ACM, La Jolla, CA, USA, 268–282. https://doi.org/10.1145/3620666.3651370.

[68] Yilun Wu, Byounguk Min, Mohannad Ismail, Wenjie Xiong, Changhee Jung, and Dongyoon Lee. 2024. IntOS: Persistent Embedded Operating System and Language Support for Multi-threaded Intermittent Computing. In *Proc. OSDI*. USENIX, Santa Clara, CA, USA, 425–443. https://www.usenix.org/system/files/osdi24-wu-yilun.pdf.

[69] Yue Yang, Emenike G. Okonkwo, Guoyong Huang, Shengming Xu, Wei Sun, and Yinghe He. 2021. On the Sustainability of Lithium Ion Battery Industry – A Review and Perspective. *Energy Storage Mater.* 36 (April 2021), 186–212. https://doi.org/10.1016/j.ensm.2020.12.019.

[70] Kasım Sinan Yıldırım, Amjad Yousef Majid, Dimitris Patoukas, Koen Schaper, Przemysław Pawełczak, and Josiah Hester. 2018. InK: Reactive Kernel for Tiny Batteryless Sensors. In *Proc. SenSys* (Nov. 4–7). ACM, Shenzhen, China, 41–53. https://doi.org/10.1145/3274783.3274837.

[71] Eren Yıldız, Saad Ahmed, Bashima Islam, Josiah Hester, and Kasım Sinan Yıldırım. 2023. Efficient and Safe I/O Operations for Intermittent Systems. In *Proc. EuroSys*. ACM, Rome, Italy, 35–50. https://doi.org/10.1145/3552326.3587435.

[72] Eren Yıldız, Lijun Chen, and Kasım Sinan Yıldırım. 2022. Immortal Threads: Multithreaded Event-driven Intermittent Computing on Ultra-Low-Power Microcontrollers. In *Proc. OSDI* (July 11–13). USENIX, Carlsbad, CA, USA, 339–355. https://www.usenix.org/system/files/osdi22-yildiz.pdf.

[73] Jianping Zeng, Jongouk Choi, Xinwei Fu, Ajay Paddayuru Shreepathi, Dongyoon Lee, Changwoo Min, and Changhee Jung. 2021. ReplayCache: Enabling Volatile Caches for Energy Harvesting Systems. In *Proc. MICRO* (Oct. 18–22). ACM, Virtual Event, 170–182. https://doi.org/10.1145/3466752.3480102.

[74] Jianping Zeng, Jungi Jeong, and Changhee Jung. 2023. Persistent Processor Architecture. In *Proc. MICRO*. ACM, Toronto, ON, Canada, 1075–1091. https://doi.org/10.1145/3613424.3623772.

[75] Yuchen Zhou, Jianping Zeng, Jungi Jeong, Jongouk Choi, and Changhee Jung. 2023. SweepCache: Intermittence-Aware Cache on the Cheap. In *Proc. MICRO*. ACM, Toronto, ON, Canada, 1059–1074. https://doi.org/10.1145/3613424.3623781.

A Artifact Appendix

A.1 Abstract

The NACHO artifact provides a comprehensive and automated framework for evaluating the proposed data cache integration into intermittent computing systems. It leverages the ICEmu Emulator (https://github.com/tudssl/ICEmu/), which supports customizable plugins to simulate various caching mechanisms, as well as perform comparisons. The artifact integrates a specific version of LLVM (16.0.2) for compiling benchmarks tailored to the NACHO architecture. The benchmarking suite includes diverse workloads designed to evaluate the performance of NACHO across multiple configurations. To streamline the user experience, the artifact is encapsulated in a Docker container that bundles all dependencies, tools, and configurations required to reproduce the results.

A.2 Artifact Check-List (Meta-Information)

- **Algorithm:** Cache controller algorithm proposed in Algorithm 1 of the main paper.
- **Program:** CoreMark, MiBench, towers benchmark, a quicksort algorithm, TinyAES, and picojpeg.
- **Compilation:** LLVM, ICEmu, ReplayCache, benchmarks
- **Output:** Comparison graph
- **Experiments:** Bash scripts and manual actions
- **How much disk space required (approx)?:** 20 GB
- **How much time is needed to prepare workflow (approx.)?:** 1 hour
- **How much time is needed to complete experiments (approx.)?:** 1 hour
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?: MIT License**
- **Archived (provide DOI)?: [To be added]**

A.3 Description

A.3.1 How to Access. The code for conducting the experiment and evaluating the results presented in the paper is hosted publicly at

`https://github.com/TUDSSL/intermittent-risc-v/`

at `nacho-artifact-release` branch.

A.3.2 Hardware Dependencies. No hardware dependencies are present as we use an emulated setup.

A.3.3 Software Dependencies. NACHO is set up and executed within a Docker container based on Ubuntu 22.04. The container provides a fully configured environment that includes all required dependencies:

- **ICEmu Emulator:** NACHO relies on the ICEmu emulator, an extension of Unicorn (https://www.unicorn-engine.org), to simulate execution with custom plugins that support intermittent computing experiments.
- **LLVM (version 16.0.2):** The system uses a customized LLVM version for compilation, integrating necessary transformations and optimizations specific to implementing ReplayCache's (system that we compare against) execution model.
- **ICEmu Plugins:** A set of plugins extends ICEmu's capabilities, enabling additional cache simulation, memory access tracking, and execution analysis.
- **Benchmarking Suite:** NACHO includes a collection of benchmarks designed to evaluate different configurations of intermittent execution and cache behavior.
- **Jupyter Notebook for Plotting:** The framework provides Jupyter notebooks to analyze and visualize experimental results.
- **Additional Dependencies:** The Docker image preinstalls all necessary libraries, avoiding manual configuration.

We recommend a minimum of 20 GB of free storage to hold all the Docker container images, toolchains and evaluation results. We also recommend that a computer executing the NACHO artifacts should have a minimum of 16 GB of main memory. This is needed as building NACHO from scratch and running all the experiments is time- and memory-consuming. That is, the complete process from building NACHO to getting all the results can take up to two hours.

A.4 Installation

The following steps define the basic flow of experimentation. This flow ensures that everything is compiled, the emulator is set up, the experiment is run, the results are collected, and the final plots are generated, allowing for the evaluation of results presented in the paper. While this is the recommended approach, users can customize the workflow based on their requirements. For instance, to verify different cache configurations, Section A.5 provides an example of running a single benchmark with specific settings.

1. **Clone the repository:** This retrieves the project source code, including all required configurations and scripts.

```
git clone git@github.com:TUDSSL/
    intermittent-risc-v.git -b nacho
    -artifact-release
```

   Note that we are using a specific branch customized for ease of artifact evaluation.

2. **Source environment setup scripts:** These scripts set up the necessary environment variables and paths required for proper compilation and execution.

```
source setup.sh
source sourceme.sh
```

3. **Build the Docker image:** The docker build script builds the Docker container with all necessary dependencies pre-installed. This ensures a consistent environment for running the experiments.

```
cd docker/development
./build.sh
```

4. **Start the Docker container:** Running `docker-start` launches the container, setting up the environment for executing the experiments.

```
docker-start
```

5. **Attach to the running container:** Using `docker-attach`, you get a shell prompt inside the container, allowing you to interact with the environment.

```
docker-attach
```

6. **Setup docker environment:** The docker environment needs to be setup before performing the build.

```
cd intermittent-risc-v
source setup.sh
source sourceme.sh
```

7. **Run the full build** Inside the Docker container, executing `make all` builds all necessary components, including LLVM, ICEmu, and its plugins. A fresh new compilation can take upwards of one hour to complete.

```
make all
```

8. **Run the experiments:** After a full build is done, the experiments can be run from inside the benchmarks directory. This runs all the experiments and generates the results (please refer to Section 6).

```
cd benchmarks
make run
```

Please note that the docker container and the storage associated with it are not cleaned up automatically and need manual cleaning after evaluation is done.

A.5 Experiment Workflow

The steps given in Section A.4 takes care of everything and should be enough for evaluation purpose. In this section we provide methods to run specific aspects of the experiment and show how to customize various aspect of our proposal.

Users can run specific benchmarks using a custom ICEmu plugin for a specific system. The script `benchmark.sh` in the repository automates the execution of benchmarks for the selected system.

A.5.1 Running Experiment. The `benchmark.sh` script allows users to execute a specific benchmark with a chosen system and optimization level. This script assumes that all required components, including LLVM, ICEmu, and necessary plugins, have already been built.

Usage:

```
./benchmark.sh <benchmark> <
    optimization_level> <system>
```

A.5.2 Available Benchmarks. The following benchmarks are available in the `benchmarks` directory:

```
adpcm, aes, coremark, crc, dijkstra,
    picojpeg, quicksort, sha, towers
```

A.5.3 Available Systems.

- **NACHO (nacho):** Proposed system
- **ReplayCache (replaycache):** Comparison system

A.5.4 Customization. Along with the command line arguements passed to the script `benchmark.sh`, it can also be modified to test different cache configurations by changing the following parameters within the script:

- **cache-size**: Cache size in bytes (must be power of 2).
- **cache-lines**: Number of cache lines (2, 4, or 8).
- **on-duration**: Period for the simulated power trace.

A.5.5 Example workflow. To run NACHO with benchmark AES with optimization level O1

```
./benchmark.sh aes O1 nacho
```

To run ReplayCache with benchmark coremark with optimization level O0

```
./benchmark.sh coremark O0 replaycache
```

Each experiment generates results in CSV format, stored in the `benchmarks/logs` directory.

A.6 Evaluation and Expected Results

Once `make all` or `benchmark.sh` is run, logs are generated and stored in the `benchmarks/logs` directory. Each log file is named based on the benchmark, system configuration, and cache parameters. Users can inspect the results manually or use the provided plotting tools for analysis.

A.6.1 Generating Plots. To visualize the results, Jupyter notebooks located in the `plotting` directory can be used. The key notebooks are:

- **BenchmarkPlots**: Generates performance comparison graphs across different benchmarks.
- **CacheVariation**: Analyzes the effect of varying cache configurations.
- **PowerFailures**: Investigates the impact of power failures on execution.

Plots generated from these notebooks are stored in:

```
plotting/plots/
```

By running these notebooks, users can obtain graphical insights into the experimental results and compare the performance of NACHO with ReplayCache across different configurations. The plots generated here are used in the paper for our evaluation.

A.7 Experiment Customization

The experiment can be customized by passing different arguments to the cache plugin to change the cache design space and enable/disable certain features as mentioned in Section A.5.4.