NvMR: Non-Volatile Memory Renaming for Intermittent Computing

Abhishek Bhattacharyya
University of Wisconsin-Madison
Madison, WI, USA
abhattacha24@wisc.edu

Abhijith Somashekhar*
Apple Inc.
San Francisco, CA, USA
asomashekhar@apple.com

Joshua San Miguel
University of Wisconsin-Madison
Madison, WI, USA
jsanmiguel@wisc.edu

ABSTRACT

Intermittent systems on energy-harvesting devices have to frequently back up data because of an unreliable energy supply to make forward progress. These devices come with non-volatile memories like Flash/FRAM on board that are used to back up the system state. However, quite paradoxically, writing to a non-volatile memory consumes a lot of energy that makes backups expensive. Idempotency violations inherent to intermittent programs are major contributors to the problem, as they render system state inconsistent and force backups to occur even when plenty of energy is available. In this work, we first characterize the complex persist dependencies that are unique to intermittent computing. Based on these insights, we propose NvMR, an intermittent architecture that eliminates idempotency violations in the program by renaming non-volatile memory addresses. This can reduce the number of backups to their theoretical minimum and decouple the decision of when to perform backups from the memory access constraints imposed by the program. Our evaluations show that compared to a state-of-the-art intermittent architecture, NvMR can save about 20% energy on average when running common embedded applications.

CCS CONCEPTS

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

KEYWORDS

energy-harvesting, intermittent computing, idempotency

ACM Reference Format:
Abhishek Bhattacharyya, Abhijith Somashekhar, and Joshua San Miguel. 2022. NvMR: Non-Volatile Memory Renaming for Intermittent Computing. In *The 49th Annual International Symposium on Computer Architecture (ISCA '22), June 18–22, 2022, New York, NY, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3470496.3527413

1 INTRODUCTION

With cutting-edge connectivity and computing resources, there has been a surge in the number of IoT devices in recent years. These devices, in spite of their small form factor, can support several sophisticated algorithms that can run on a normal desktop or a laptop. Most of the time, these devices draw their energy for computation (e.g. smartphones, smartwatches, etc.) from a battery that needs to be charged periodically. This makes batteries the bulkiest components in these devices. However, in many cases these devices come in small form factors or are installed in remote locations and thus, may not be tethered to a battery. Hence, there is a push towards batteryless devices that harvest energy from ambient sources around them (Solar, RF, Wind, or Piezoelectric devices) [11, 12, 15, 29, 32, 35] to run applications until battery technologies evolve. In general, energy is stored in the form of charge across a capacitor or current passing through a resistor.

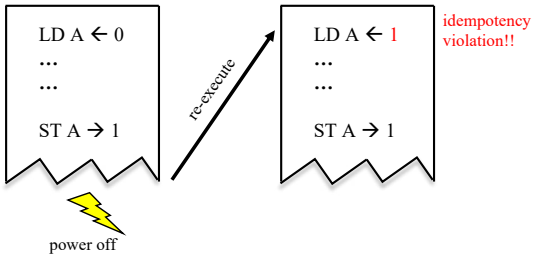


Figure 1: A sudden power loss can cause an idempotency violation in intermittent program execution

These batteryless devices come with their own set of challenges that require both software and hardware solutions. Since these devices depend on ambient sources for energy, it makes the power supply unpredictable. From an architectural perspective, the key challenge is to perform useful computation before the device runs out of energy. This type of computing, also known as *intermittent computing*, often needs to back up application and processor state to a non-volatile memory (NVM) before the charge in the capacitor drains out. The system state is restored and computation is resumed once there is enough charge in the capacitor again. As writing to NVMs consumes a lot of energy, backups tend to be quite expensive. So, it is important to minimize the energy overhead for backups so that the remaining energy can be utilized for making forward progress.

1.1 Challenges in Correctness and Efficiency

Previous works have suggested various ways to perform backups in an energy-efficient way. Early works use just-in-time checkpoints at a predetermined threshold voltage [2, 3, 17] to backup when absolutely necessary. However, recent works have shown that minimizing backup overheads is not enough. An unexpected

*This work was done while the author was at UW-Madison.

ISCA '22, June 18–22, 2022, New York, NY, USA

ACM ISBN 978-1-4503-8610-4/22/06…$15.00
https://doi.org/10.1145/3470496.3527413

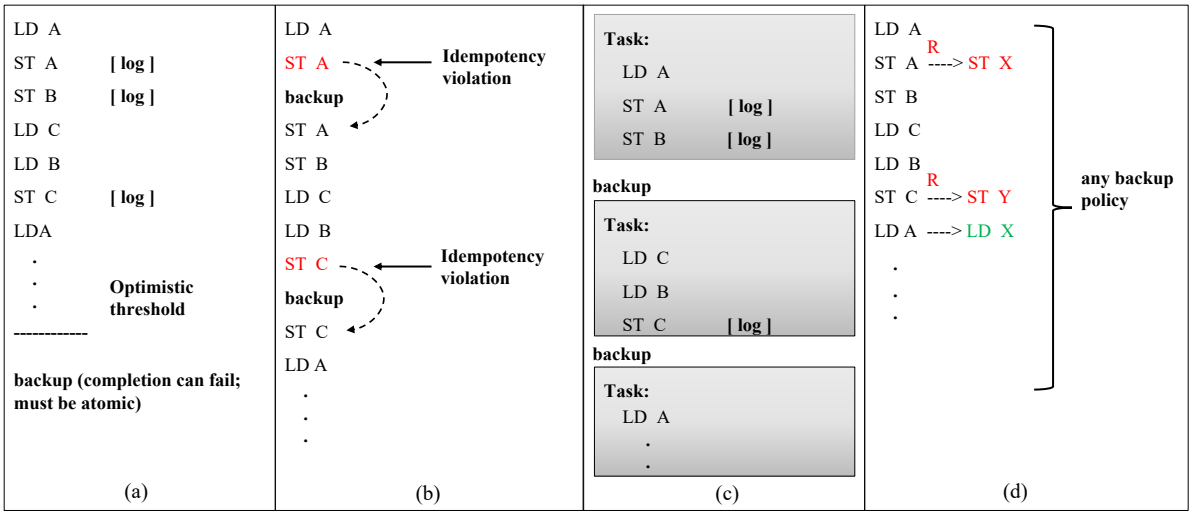


Figure 2: (a) Backup policy with an optimistic threshold (backup must be atomic), (b) Clank (backup when there is an idempotency violation), (c) DINO (backup at the boundary of pre-defined atomic tasks), (d) NvMR (any backup policy of choice); R stands for renaming. NVM addresses A and B are renamed to X and Y.

power loss in the middle of a running application can lead to an incorrect processor state. For instance, consider the program in Figure 1. `LD A` returns a value `0` since address `A` is initialized to `0` in NVM. The following `ST A` writes a value `1` to address `A` in NVM. However, an abrupt power outage after the store leads to re-execution of the program. But this time `LD A` reads from NVM and returns a value `1` which is incorrect. This discrepancy is known as an *idempotency violation* [27, 34]. The software solution, proposed in [7, 22, 26], divides an application into programmer-defined idempotent tasks and performs backups at task boundaries. But this approach puts the burden of determining execution-safe regions of code on the programmer. In [16], the hardware detects idempotency violations and performs a backup upon detection. But if an application has too many idempotency violations the device will have to back up often even though there is sufficient energy to sustain further execution of instructions. This presents a conundrum where an intermittent architecture has to backup on idempotent violations but also make forward progress. An ideal solution will be an intermittent architecture that guarantees program correctness in the most energy-efficient manner.

1.2 Our Approach: NvMR

To address the problems discussed above, we first characterize the complex persist dependencies that must be enforced for correctness and are unique to intermittent computing. Upon careful examination we find that renaming NVM addresses can eliminate most of these dependencies and reduce backup costs to their theoretical minimum. We propose *Non-volatile Memory Renaming* (*NvMR*) – an intermittent architecture that bypasses idempotency violations by renaming addresses of stores to NVM. This achieves two key goals: (1) it minimizes the number of backups that need to be invoked, and (2) it decouples the decision of when to perform a backup from the program execution. Ideally, a backup scheme should be flexible and solely dependent on environmental conditions. However, due to the presence of idempotency violations, the program is more often the dominant decider of when to perform backups in state-of-the-art intermittent systems. Our experiments on a previous intermittent architecture (Clank [16]) with popular embedded benchmarks show that backups due to idempotency violations account for more than *30%* of the total energy consumption on average.

2 BACKGROUND

Previous studies have explored different techniques to checkpoint data in order to make forward progress. In this section, we explain some of the backup schemes using the same example program. These backups schemes can be broadly classified as hardware-based or software-based.

2.1 Hardware-based Checkpointing

[2, 3, 17] propose backup schemes with an optimistic threshold by recording updates to NVM in logs and atomically committing them on a backup (stores to addresses A, B and C in Figure 2a. They use an analog-to-digital converter (ADC) to detect if the supply voltage dips below a predetermined threshold and back up once just before power failure. However, the backup needs to be atomic to guarantee program correctness. In such systems data updates are stored in a redo log in NVM. Such systems do not need to be idempotency violation-aware as the store updates are re-executed from the log during restore [7, 48] and loads are never re-executed if the backup were successful. HOOP [6] is a log-based architecture for persistent memory that we adjust to semantically resemble a log-based intermittent architecture. HOOP does out-of-place (OOP) updates to a statically allocated OOP Region in NVM. A mapping table renames the addresses of dirty cache blocks evicted from a data cache to addresses from OOP Region. To reduce NVM writes, a volatile OOP Buffer packs word updates into slices that are finally

written to OOP Region during a backup. HOOP invokes a backup when the OOP Buffer or the mapping table gets full, or based on the backup policy. During restore or if the OOP Region gets full, HOOP has to perform garbage collection. During restore, HOOP does garbage collection to update the original addresses with the new data from the OOP Region.

Clank [16] is a hardware solution that divides the addresses of a program into two categories: read-first and write-first, in order to detect idempotency violations. If the first access to an address after a backup is a load, the address is placed in a read-first buffer. Thus, addresses A and C in the example program will be inserted into read-first buffer. On the other hand, an address is put in the write-first buffer if the first access to it is a store (e.g. address B). An address can be present in only one of these buffers at any time. If there is a subsequent write to an address which is in the read-first buffer, the hardware flags an idempotency violation and triggers a backup. ST A and ST C are the violating stores that cause backups. They are highlighted in red text and shown again with dotted curved arrows in Figure 2b to indicate that an idempotency violation is detected first, followed by a backup and then the violating instruction is executed. If the address is in the write-first buffer instead, there is no action and normal execution resumes. If any of the buffers gets full, the device does a backup when a new entry is needed. Though this approach ensures correct intermittent execution, it imposes numerous backups on the system than actually needed, since there is plenty of energy available to continue execution most often.

2.2 Software-based Checkpointing

[7, 22, 26] suggest decomposing long-running executions into user-defined atomic tasks. Checkpoints are inserted at task boundaries, guaranteeing application state to be consistent. Such a backup scheme is shown in Figure 2c. To keep a task idempotent, [25] introduce a privatization buffer that stores updates to variables shared between tasks until the task commits atomically. Similar to Clank, this approach imposes more backups on the system than actually needed, since tasks are sized much smaller than the available energy supply. Furthermore, as highlighted in prior work [16], task-based solutions can be challenging for programmers since task decomposition is static and often needs detailed knowledge of the intermittent hardware to make appropriate task decomposition.

NvMR renames the addresses of the idempotency violating stores in our toy program, ST A and ST C, to X and Y respectively (shown with a red R above a dotted arrow in Figure 2d)[1]. The subsequent load to address A will read the updated data from the most recent mapping X. This way NvMR decouples the backup scheme from the underlying hardware execution. So, any backup policy based on the operating conditions can be deployed.

In the following sections, we first model the fundamental persist dependencies that are unique to intermittent computing and analyze them in detail. We then present our NvMR architecture in Section 4, inspired from concepts (and limitations) of these prior works. Section 5 then talks about the tools and techniques used for our simulation and performance evaluation. Section 6 provides a thorough analysis of both performance and energy results from our simulation. In Section 7, we present a brief discussion of literature from related work before summing up with a conclusion in Section 8.

[1]Addresses are renamed after the execute stage of the pipeline when all addresses are known

3 INTERMITTENT PERSIST DEPENDENCIES

Several works have built models to characterize the efficiency of intermittent systems [3, 5, 29]. However, none have built generalized models that describe the necessary persistency constraints for correct intermittent execution. These persist dependencies must be upheld for a program to execute correctly, as if it had run in a continuously-powered system. Memory persistency models have been explored [20, 33] though they address fundamentally different challenges; these works instead focus on interactions between memory persistency and consistency for shared-memory multiprocessors. According to our knowledge, this is the first work that provides a persist dependency (and ordering) model for intermittent systems. Although prior works have suggested correctness models for intermittent execution based on data dependencies, they bypass the discussion on persist dependencies by assuming an architecture with no buffering of dirty data (e.g. a write-through cache) [41, 42]. We consider an architecture where all load and store instructions access NVM addresses, with a volatile write-back cache acting as an intermediary. As shown by [48], a write-back cache enables reuse of dirty data and reduces the number of expensive writes to NVM. Backups save the contents of the volatile register file (including the program counter) into NVM. After a power loss, processor state is restored to the last persisted backup. Any instructions that were executed after the backup must be re-executed. Our goal is to provide a theoretical understanding of which persists depend on each other and why, in the context of intermittent execution.

3.1 Requirements for Correct Intermittent Execution

The first observation we make is that there are three main requirements that must be enforced in traditional intermittent architectures for correct program execution:

(1) *Code Progress:* Stores to the same address must be persisted in program order and backups must be persisted in the order that they are invoked. This ensures that the program eventually completes, as long as the device's energy supply is large enough for at least one backup.
(2) *Data Progress:* The value of a store to NVM address X that precedes a backup (in program order) must be persisted prior to persisting the backup. This ensures that program data is always kept up-to-date across power losses.
(3) *Idempotency:* If the first access to NVM address X after a backup is a load, the value of any subsequent store to the same address that succeeds the backup (in program order) must not be persisted until the next backup is persisted, in case power is lost before the next backup. This ensures that program data is not corrupted by instructions that were not backed up before the last power loss.

3.2 Code Sections in Intermittent Execution

A non-volatile memory location X is *read-dominated* if the first load from X after the most recent backup precedes the first store to X in

program order. Conversely, X is *write-dominated* if the first store precedes the first load. Thus, in an intermittent code section an address X can be either read-dominated or write-dominated[2]. This distinction is vital to characterizing the correctness requirements for persists. Recent work [16, 46] recognizes that Requirement 3 only needs to be enforced if X is read-dominated. If X is write-dominated, the store's value can be persisted prior to the next backup. Though X would be corrupted upon a power loss, upon re-execution, X will first be overwritten before being read, nullifying the corruption.

3.3 Modeling Intermittent Persist Dependencies

Understanding when backups may be persisted relative to when stores are persisted is important. Thus, we aim to characterize all the persist orderings including the ones between persist backups and persist stores for correct program execution in an intermittent setting. We will show that current intermittent systems have only scratched the surface in understanding the complex dependencies between persists and, thus, enforce conservative (inefficient) constraints on backups and stores.

What are the minimum ordering constraints for correct persists? We formulate the problem statement as: for a given program and set of backups invoked during its execution, what dependencies must be respected among persists? Each backup records the contents of the volatile register file (including the program counter) at some instance in the program execution and persists them into NVM. Similar to store instructions, persisting the backup can be decoupled from its execution.[3] That is, writing the register contents into NVM can occur after (or even before) the backup is invoked, as long as the persisted register contents match what they would be at the position in program order at which the backup was invoked. Thus, the processor may invoke a backup and continue executing instructions after it; however, if power is lost before the backup is persisted, the processor must revert its state to an earlier backup.

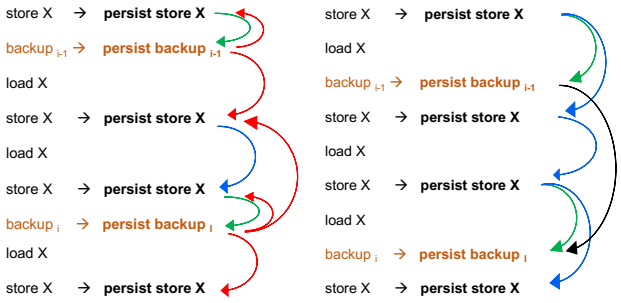
We offer a formal description of the dependencies among persists in an intermittent execution. With this purpose, we model the happens-before persist orderings as relations between a pair of persist operations *{p1,rel,p2}*, where *p1* and *p2* are two persist operations in program order and the direction associated with *rel* represents the persist order between the pair. For example, *{st,* $\overset{spo}{\longrightarrow}$*, st}* is the relation between a pair of persist stores to the same address, representing the necessary ordering between them to maintain program order (Requirement 1). We color code the various relations that exist in an intermittent execution and list them alongside the requirements that they satisfy in Table 1. Note that the ordering, *{st,* $\overset{spo}{\longrightarrow}$*, st}*, is needed for correct program execution in continuous execution too, while the other orderings are specific to intermittent execution. In general, all backup schemes for intermittent systems can be broadly classified as *single-backup* [3, 23] or *multi-backup* [7, 15, 16] schemes based on the number of backups invoked before a power loss [39]. *{backup,* $\overset{bpo}{\longrightarrow}$*, backup}* ordering is necessary in multi-backup schemes only.

[2]Address X is symbolic and representative of any address

[3]Note that the persist backup itself must be performed atomically (i.e., all registers' contents and program data must be backed up together), usually via double buffering.

Requirement	Persist Ordering
Code Progress	ST X → ST X (spo)
Code Progress	backup → backup (bpo)
Data Progress	ST X → backup (rfpo)
Idempotency	ST X → backup (irpo)

Table 1: Persist happens-before orderings (*spo*: store persist order, *bpo*: backup persist order, *rfpo*: read from persist order and *irpo*: idempotent re-execution enforced persist order) in an intermittent execution along with the correctness requirements that they fulfill, shown with color-coded arrows



(a) read-dominated **(b) write-dominated**

Figure 3: Happens-before persist orderings for correct intermittent execution. Cyclic sequence of orderings represent atomicity constraints

Figure 3 shows the necessary persist dependencies for any general program execution, characterized by read-dominance and write-dominance, focusing on the symbolic NVM location X. We ignore the dependencies in instruction execution and assume that they respect program order. We also ignore loads and assume the most up-to-date values are readily available in the write-back cache, except immediately after a power loss; in that case, the load must read the value in NVM. Backups can be invoked at any arbitrary point in the program, but when invoked, they introduce a set of happens-before ordering and atomicity constraints with surrounding stores. The backup point also decides if the next intermittent code section is read-dominated or write-dominated for address X.

3.4 Read-Dominance

Figure 3a models the dependency pattern when all stores access read-dominated addresses. To ensure data progress (Requirement 2), each store must persist before the next backup persists. Otherwise, if the next backup were persisted first and power is lost, the most up-to-date value of X would be lost. Simultaneously, to ensure idempotent execution (Requirement 3), each store must not persist until the next backup persists. Otherwise, if the store were persisted first and power is lost, the program would re-execute from the prior

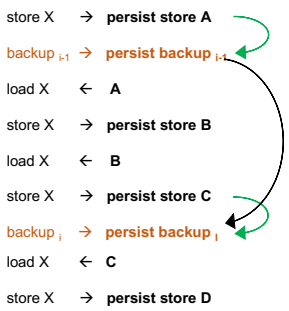


Figure 4: Happens-before persist orderings with proposed NVM renaming. By renaming X, all stores access only write-dominated memory locations. Only the stores that immediately precede backups must be persisted.

backup and incorrectly load the later store's value of X. These constraints combined imply that all stores to a read-dominated address must atomically persist with the next backup. This is a unique, fundamental challenge in intermittent computing systems.

3.5 Write-Dominance

Figure 3b shows that for the same program, with a simple shift of when backups are invoked, X becomes write-dominated. As a result, the dependency pattern changes dramatically. Now only Requirements 1 and 2 need to be enforced, which is done by persisting backups in program order and ensuring that stores are persisted before the next backup. As discussed earlier, idempotent execution is guaranteed when stores access write-dominated addresses. This implies that stores no longer have to wait until the next backup to persist, eliminating the atomicity constraint. In fact, stores after a backup may even be persisted before the backup persists. Since X is always write-dominated, whenever power is lost, regardless to which backup we revert, X will always be overwritten first, nullifying the last persisted store.

3.6 Motivating Non-Volatile Memory Renaming

Idempotency is a requirement for correct intermittent execution and can only be violated by false (write-after-read) dependencies on NVM locations. This principle will guide the creation of our novel intermittent architecture that dynamically *renames* read-dominated NVM addresses. Renaming is a standard technique for breaking false register dependencies (i.e., write-after-read, write-after-write) in out-of-order processors. This allows falsely-dependent instructions to bypass each other during execution. Memory renaming has also been referred to in the context of store buffers and memory disambiguation in out-of-order processors [18, 44]. Though inspired from these approaches, our concept of *NVM renaming* is fundamentally different. We do not aim to enable out-of-order execution but rather translate NVM addresses to guarantee idempotent execution (Requirement 3) regardless of when backups are invoked. Our key insight is: *by renaming NVM locations, all addresses become write-dominated.* As evident from Figure 4, renaming eliminates the orderings $\{st, \xrightarrow{spo}, st\}$, $\{backup, \xrightarrow{irpo}, st\}$ and $\{st, \xleftarrow{irpo}, backup\}$

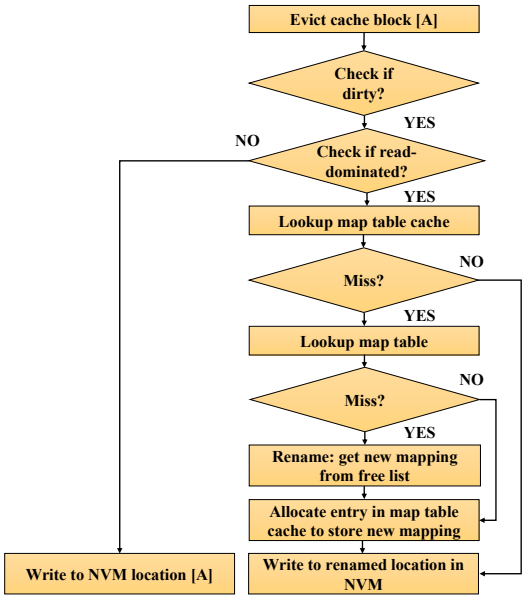


Figure 5: Flowchart showing the handling of idempotency violations in NvMR

present in Figure 3a. Only the stores that immediately precede backups must be persisted to ensure data progress (Requirement 2) as shown in Figure 4; all other stores do not need to be persisted at all, achieving the theoretical maximum efficiency. Backups are still required to be persisted in the order that they are invoked to maintain code progress (Requirement 1). In other words, rendering all addresses as write-dominated minimizes ordering constraints to their theoretical minimum, thus decoupling the decision of when to perform a backup from the correctness constraints of the program.

4 NvMR ARCHITECTURE

Based on the discussion in the previous section, we propose *Non-volatile Memory Renaming* (*NvMR*) intermittent architecture that detects idempotency violation in hardware and uses renaming of NVM address to eliminate the relevant perist orderings.

4.1 Overview

Figure 5 explains the whole process of renaming in NvMR in a concise, high-level flowchart. NvMR renames the address of a persist store to a read-dominated address. NvMR uses a write-back data cache to take advantage of an application's data locality deriving motivation from previous works [10, 43, 48]. Besides, NvMR has two other structures apart from the write-back, write-allocate (WBWA) data cache to keep track of read-dominated and write-dominated addresses and detect idempotency violations: *global bloom filter (GBF)* and *local bloom filter (LBF)*. GBF keeps track of read-dominated cache blocks and LBF keeps track of read-dominated or write-dominated words in a cache block.

NvMR renames the address of a read-dominated cache block if it is dirty when it is evicted from the cache. NvMR identifies such a cache block by reading the value of its dirty bit and the composite

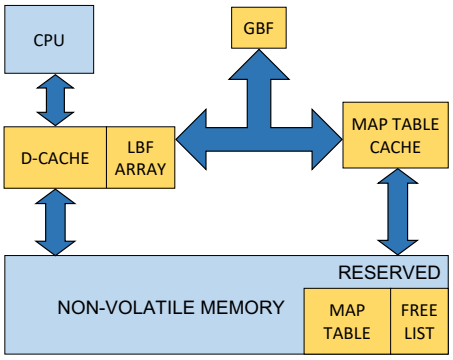


Figure 6: Block diagram of NvMR architecture (blocks shown in gold are added)

state calculated from its LBF[4]. GBF logs if a cache block is read-dominated or not when they get evicted from the cache using their address for the membership hash function. NvMR renames the NVM address of such a cache block with the aid of a *map table* and a *free list*, allocated in NVM, to a new address from a region in NVM that has been reserved by the compiler. The data from the cache block is written to this new mapping in NVM, guaranteeing that the contents of the original address are left untouched. In this manner, NvMR bears some resemblance to copy-on-write (COW) mechanisms used in operating systems to reduce storage overhead associated with forking multiple processes [38, 47]. However, in operating systems, copy-on-write is done to prevent sharing of modified resources between two processes, while in NvMR, a new location of the modified cache block is created to avoid idempotency violations. Since NVM accesses are expensive, NvMR has an SRAM write-back *map table cache* on-chip to reduce the number of accesses to the map table. The map table and the map table cache keep a record of the old and the new mappings, while the free list is a queue of available mappings from the compiler-reserved region of NVM that can be used for renaming. Figure 6 presents an overview of our NvMR architecture with a block-level representation of all the hardware components described so far.

4.2 Intermittent Execution

In Figure 7 we can see the map table cache. Each entry in the map table cache contains five fields:

- *valid*: set to 1 to allocate a new entry.
- *dirty*: set to 1 to indicate that this entry has the latest mapping of a program address and not the map table.
- *tag*: derived from the program address and used to look up the map table cache.
- *old*: original address or most recently backed up mapping of a program address.
- *new*: new mapping assigned during renaming.

[4]Each LBF stores the individual state of the constituent words of a cache block (Unknown=00, Read-dominated=01 and Write-dominated=10) and is tightly coupled with the data cache. The composite state of a cache block is the OR of the LSBs of the LBF states of all the constituent words. The composite state can be either Unknown=0 or Read-dominated=1

Each entry in the map table holds only three of the above fields: valid, tag and old.On a backup, the old field of the map table is updated with the latest mapping from the new field of a dirty map table cache entry. Memory addresses are shown at cache block boundaries in the Application Region to simplify the discussion (Figure 7). In order to understand the microarchitecture of NvMR better, we divide intermittent execution into separate events and explain them individually in detail with an example.

4.3 Idempotency Violation.

Figure 7 shows a cache block X in the write-back cache with a tag, X_{tag}. This cache block was fetched from NVM because of a read-dominated access to the word [C]. Later word [A] of the same cache block was written with [A_{new}]. Thus, its dirty bit in the data cache is set and the corresponding LBF states of [C] and [A_{new}] are set as R and, W respectively. In NvMR, upon a dirty cache block eviction, the composite state of the cache block is calculated from the individual LBF states of its constituent words. If the composite state is 0, there is no idempotency violation. However, block X is dirty and its composite state is 1. On eviction of this block, NvMR logs the composite state in GBF and flags an idempotency violation A1.

4.4 Renaming

Write back of a dirty cache block triggers a lookup in the map table cache to find its latest mapping. The block X incurs a miss as it does not have any entry in the map table cache A2. This, in turn, triggers a lookup in the map table, which also results in a miss A3. A new mapping is popped from the head of the free list and the read pointer is moved to the next location of the free list as shown in Figure 7 A4. NvMR logs 200 and 710 as old and new mapping of the block X respectively in a new entry in the map table cache A5. Thus, address 200 of block X is renamed to address 710. This entry is marked valid, and dirty. It is important to note that NvMR can allocate a new map table cache entry only if there is at least one empty entry in the map table. The contents of the block X are written to the address 710 in NVM using the new mapping from the map table cache, A6. If a lookup is a hit in the map table, the old mapping is fetched into the old field of the map table cache before renaming. It is noteworthy to mention here that if the lookup in the map table cache returns a hit, an evicted, dirty cache block is persisted at the location pointed to by the new mapping and no renaming is necessary.

4.5 Cache Miss

As shown in Figure 8, there is a subsequent store request to the cache block X. It wants to write a word, [B_{new}]. But since this block was recently evicted from the data cache, the lookup causes a cache miss B1. To fetch the cache block data from its latest mapping in NVM, the cache controller invokes a lookup in the map table cache. Since address 200 has been renamed to address 710 and the entry with this information still exists in the map table cache, the lookup is a hit B2. The cache block will be fetched from the new mapping 710 in NVM, which has the latest data. The word [B_{new}] is updated in the cache block, and it is marked dirty B3. To update the states of the associated LBF, NvMR performs a lookup in GBF. GBF stores

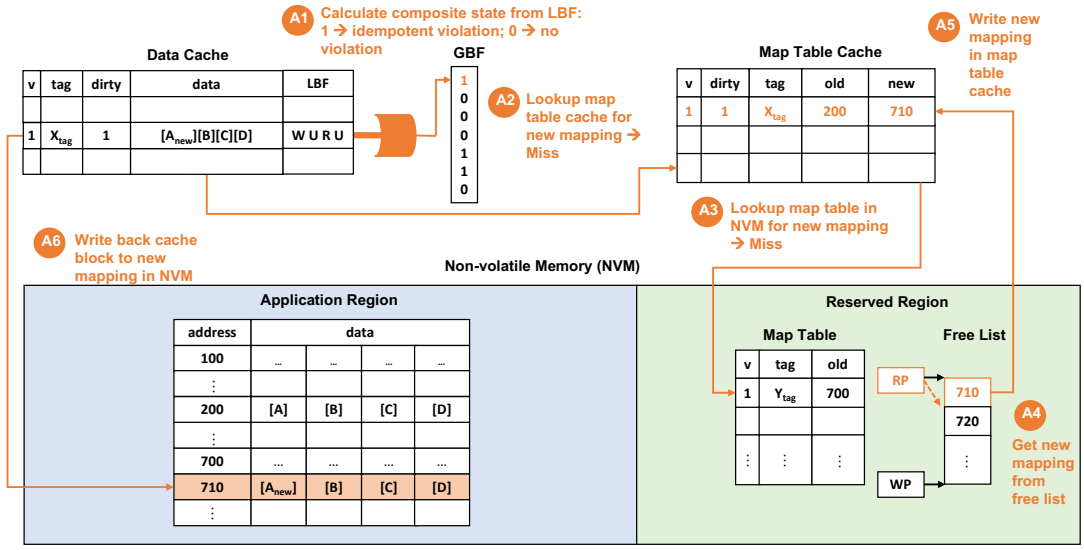


Figure 7: Eviction of a cache block with tag X_{tag} causes an idempotency violation. Cache block address `200` is renamed to address `710`. The dirty cache block is stored at the new mapping `710` instead of `200`.

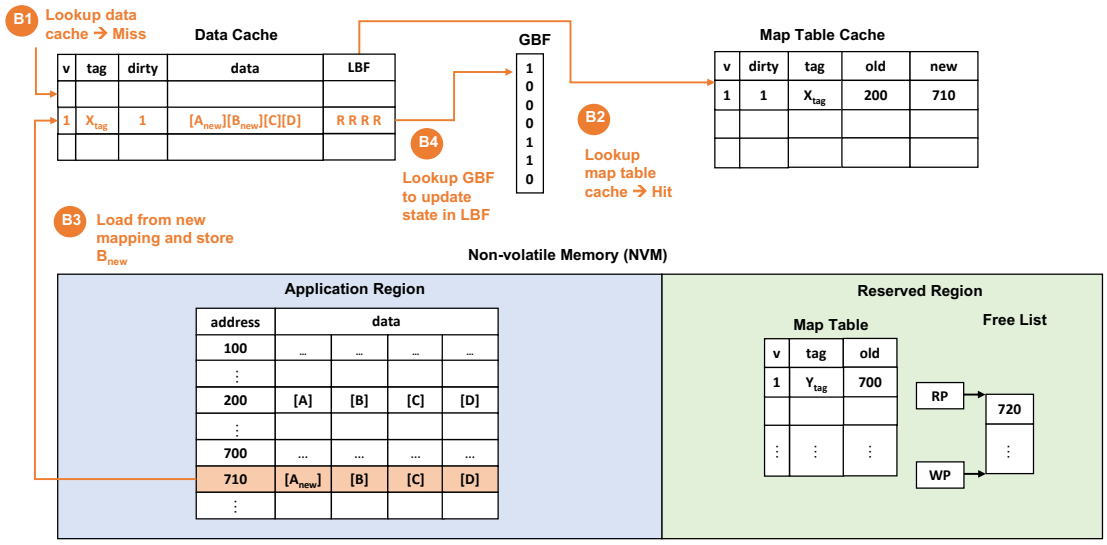


Figure 8: A store miss causes the cache block to be fetched from the new mapping `710`. NvMR keeps a log of the new mapping in the map table cache.

the composite state of cache block address, which indicates if it is read-dominated or not. As the composite state of the block `X` in GBF is 1, it is read-dominated. All the LBF states are conservatively marked as `R` to reflect this (B4). If the composite state of a block is U, the corresponding LBF state of the word is marked as either R or W based on whether it is a read or write request, respectively.

4.6 Backup

NvMR can back up on three occasions. First, NvMR performs a backup when a dirty entry is evicted from the map table cache. This is to ensure that the map table in NVM always contains the mappings from the most recent backup. In addition, NvMR can also perform a backup based on the backup policy implemented by the user to ensure forward progress. Lastly, NvMR invokes a backup if there is an idempotency violation after either the map table is full or the free list is empty, as renaming is not possible in both the circumstances. Like all other intermittent architectures, NvMR too takes a snapshot of its general-purpose registers and special registers like the program counter. Hence, it is not shown in Figure 9. Since NvMR has a write-back data cache, it persists all the dirty blocks on a backup. However, NvMR must store the

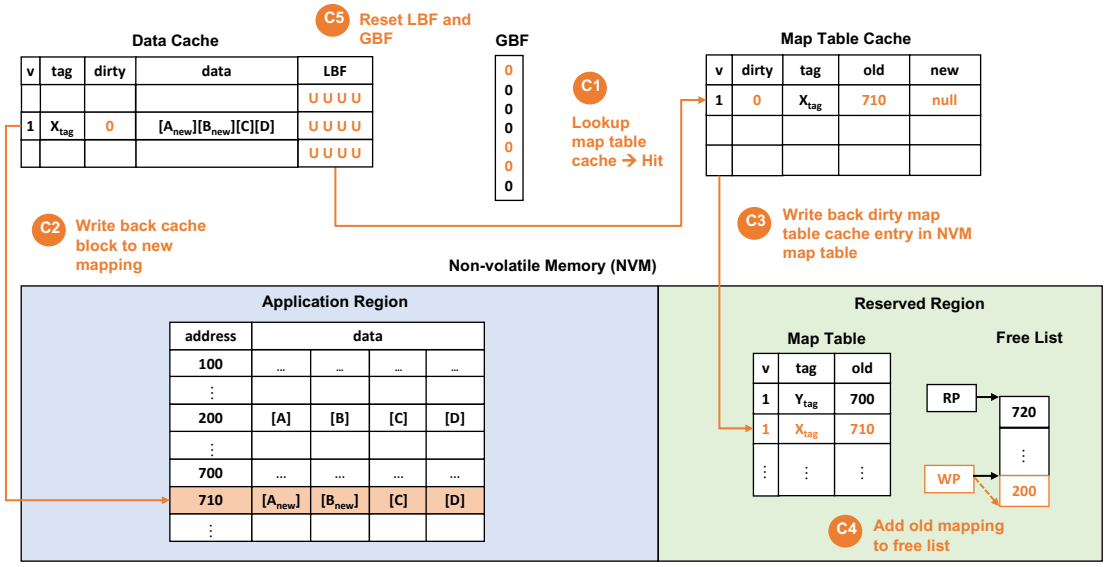


Figure 9: Backup will cause the data cache to persist its dirty blocks. The old mappings from the map table cache are pushed into the free list, and the new mappings are written to the map table. All states in LBF and GBF are reset.

dirty data to the latest mapping if the block has been renamed at some point of the program execution to guarantee correctness. So, NvMR looks up the map table cache to find the latest mapping for every dirty cache block. Otherwise, the latest mapping has to be fetched from the map table. If the lookup in the map table is a miss, the data is stored to the program address of the cache block.

Fortunately, for the block `X`, the lookup is a hit in the map table cache (C1). The new mapping `710` is where the dirty cache block data is finally persisted (C2). The dirty bit of block `X` is set to `0` to indicate that the write-back completed successfully. After the dirty cache blocks are persisted, the new mappings from the dirty map table cache entries are written back to the map table in NVM. The old mappings from the dirty map table cache entries are added to the tail of the free list. In our example, NvMR writes back the new mapping `710` for the block `X` in the map table (C3). It also adds the old mapping `200` to the tail of the free list (shown with the dotted orange arrow to indicate the change in position of the write pointer) (C4). The dirty bit of the map table cache entry is set to `0` upon completion of the write-back, and the new mapping is copied to the old mapping in the map table cache. Thus, `710` is copied to the old field and the dirty bit is cleared in Figure 9. The read and write pointers of the free list are also saved at this point (not shown in figure). Finally, GBF and LBF array are reset (C5).

4.7 Restore

For an intermittent system, power loses are frequent. The system state has to be restored to the previous checkpoint when there is enough power to resume execution. In NvMR, restore is pretty simple. The program counter and the other registers are restored from the previous snapshot saved in NVM. The read and write pointers of the free list from the most recent checkpoint are also restored.

4.8 Reclamation

The map table is a limited hardware resource and can get full pretty quickly for applications that incur a lot of idempotency violations. Besides, most of the energy-harvesting devices have small non-volatile memories on board and can't afford to have a large map table. To mitigate this issue, we introduce a technique called *reclaiming*. The idea is quite simple and elegant. The *original mapping* of a cache block is the cache block address of the memory request from the program. After a backup due to an idempotency violation and a full map table, NvMR reclaims a map table entry based on a replacement policy like the one used in the caches. Reclaiming has to atomically push the old mapping from the map table to the tail of the free list, invalidate the entry and copy the contents of the cache block from the freed mapping to the original mapping. Thus, original mapping of the address is *reclaimed*. Besides, NvMR also does a lookup in the map table cache and invalidates the corresponding entry on a hit. Thus, reclaiming reduces the number of backups after the map table becomes full. But there is a caveat with using reclaiming. The original mappings cannot be added to the free list which effectively reduces the size of the free list. Thus, the free list should have a higher number of mappings in order to compensate and avoid backups due to shortage of available mappings.

5 METHODOLOGY

5.1 Simulation Infrastructure

For our evaluations, we extend a cycle-accurate C++ simulator from prior works [16, 39] to model an intermittent execution platform with an ARM Cortex M0+ [1] (3-stage in-order pipeline) that executes instructions from ARM Thumb ISA. Our power model samples voltage traces collected in a previous work at 1kHz frequency to calculate the charge across the supply supercapacitor [28][5]. We use

[5] All our results presented in Section 6 are averaged across 10 different voltage traces.

Parameter	Value
Processor	ARM Cortex M0+ (Thumb ISA), 8MHz
Data Cache	256B, 8-way associative, 16B block, LRU, 1 cycle hit latency
GBF	8 one-bit entries
LBF	4 two-bit entries per cache line
Map Table Cache	512 entries, 8-way associative, LRU
Map Table	4096 entries, LRU
Free List	4096 + 512 + 1 = 4609 mappings
Flash	2MB
Supercapacitor	100mF, 2.4V max. voltage

Table 2: System configuration used in evaluation

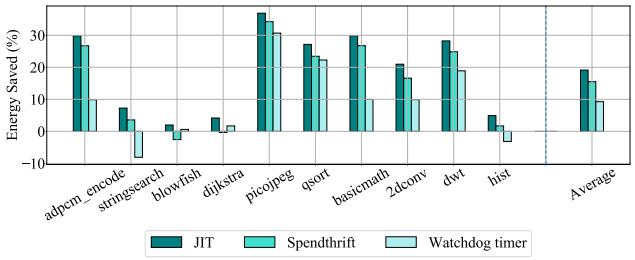


Figure 10: Energy saved in NvMR with different backup schemes compared to Clank

CACTI [31] to model dynamic and leakage power of the hardware structures added - data cache, global bloom filter, local bloom filter, map table cache. The energy numbers for NVM accesses are from the data sheet for a STM32L011K4 MCU board [40]. We compare NvMR with our version of Clank [16]. Clank is a hardware solution proposed to tackle backups in intermittent systems without causing idempotency violations. We modify the original Clank to implement our version of Clank. Our version of Clank has a GBF and an LBF instead of buffers to track read-dominated and write-dominated addresses. We also replace the write-back buffer with a write-back data cache, following suit from recent works that have shown the benefits of caches in state-of-the-art intermittent architectures [10, 48].

We use the configuration listed in Table 2 for the results presented in Section 6.1, 6.2, 6.4 and 6.5. Our experiments yield that the configuration (data cache, GBF and LBF) in Table 2 works the best for our version of Clank[6]. For NvMR, we allocate a free list with the worst-case size *(#available mappings = #map table entries + #map table cache entries + 1)* in order to eliminate any backups due to an idempotency violation when the free list is empty.

5.2 Backup Schemes

Unless stated otherwise, the results presented in Section 6 are for a *Just-In-Time* (JIT) backup scheme. The JIT scheme accurately estimates when a power loss will happen and triggers a backup just before it. Apart from it, we use two other schemes in Section 6.1 – *spendthrift* and *watchdog timer*. The spendthrift backup scheme uses a lightweight neural network to predict when to back up [24], representative of JIT schemes deployed commercially. The neural network takes in two input parameters, current environment and battery voltages. The model is implemented using PyTorch v1.8 and was trained offline with the selected benchmarks. We train 2 models, one on baseline and one on NvMR with the system configurations as shown in Table 2 using output collected from running with oracle backup scheme on 7 different voltage traces and tested on 3 different voltage traces. It is about *97%* accurate. The trained models are deployed to work in conjunction with the simulator using libtorch library for C++. The watchdog timer backup scheme invokes a backup every *8000* cycles [16]. It is the most conservative policy out of the three.

5.3 Benchmarks

We select seven benchmarks (*adpcm_encode, basicmath, blowfish, dijkstra, picojpeg, qsort* and *stringsearch*) from a benchmark suite for IoT applications, MiBench [13] for evaluation. These benchmarks have higher number of idempotency violations than the others in that suite (Table 3)[7]. We also port three benchmarks, 2D Convolution *(2dconv)*, histogram equalization *(hist)* and discrete wavelet transform *(dwt)* from PERFECT benchmark suite [4] to our simulation infrastructure and tune them to have similar memory requirement as that of applications generally supported in ultra-low power platforms.

6 EVALUATION

In this section, we compare the results of NvMR against our version of Clank. In Section 6.1, we study the results for energy saved when we execute the selected benchmarks in NvMR with different backup schemes. In Section 6.2, we offer a comparison to a transaction-based system, HOOP. In Section 6.3, we analyze the sensitivity of energy saved in NvMR when we vary the configurations of the hardware components. Since both NvMR and our version of Clank has a GBF, an LBF and a write-back cache, we don't present sensitivity studies for them. We also vary the supercapacitor size and analyze its impact on the results. In Section 6.4, we present the results of NvMR with and without reclaiming. In Section 6.5, we briefly discuss the overheads associated with NvMR.

6.1 NvMR vs. Clank

Figure 10 presents the percentage of energy saved in NvMR compared to Clank using the three backup schemes mentioned in Section 5.

6.1.1 Just-In-Time. With this backup technique, NvMR saves on average about *20%* energy compared to Clank. Out of the benchmarks studied, *picojpeg* performs the best on NvMR. The energy savings can range from *2%* to *37%* based on the application. NvMR shows the highest improvement in energy savings with this scheme, as it is the most aggressive out of the three.

[6]For the same on-chip data storage, our version of Clank saves 11% more energy than the original Clank

[7]These numbers are obtained by simulations with an ideal architecture where backups occur due to the JIT scheme and not because of any structural hazards.

adpcm_encode	basicmath	blowfish	dijkstra	picojpeg	qsort	stringsearch	2dconv	dwt	hist
$9.71x10^3$	$2.87x10^6$	$4.71x10^4$	$1.29x10^5$	$1.52x10^5$	$7.21x10^5$	$4.19x10^5$	$3.37x10^5$	$1.36x10^4$	$2.61x10^3$

Table 3: Number of idempotent violations per benchmark

6.1.2 Spendthrift. With the neural network-based scheme, NvMR saves about *15.6%* energy on average. *Blowfish* and *dijkstra* are the benchmarks that marginally consume more energy in NvMR. Both these benchmarks spend the least amount of energy on backups due to idempotency violations. Without an optimistic backup scheme it is difficult to extract energy savings from these benchmarks.

6.1.3 Watchdog Timer. On average, the percentage of energy saved with this backup policy is just above *9%*. *stringsearch* and *hist*, perform worse than in Clank because the timer period is not profiled well enough for them. Since it is not a JIT backup scheme, there is dead energy spent on re-execution. The percentage of dead energy for these two benchmarks is between *6.5%-7%*, while for the rest it is below *2%*. Since it is the most naive backup scheme out of the three discussed, NvMR saves the least amount of energy with this scheme.

6.1.4 Break-down of energy consumed. The total energy consumption for a benchmark in an intermittent architecture can be split into four components according to previous work - *forward progress, backup, restore* and *dead* energy. For a detailed explanation of these energy components, we refer the reader to the previous work [39]. In addition, NvMR has overhead versions of forward progress and restore energy due to the dynamic and leakage energy spent for the map table cache and additional NVM accesses to the map table and the free list. Also, there is an energy component related to reclamation.

Figure 11 presents the break-down of normalized energy consumption of Clank and NvMR. Due to the JIT backup scheme that invokes a backup just before a power failure, there is no dead energy. Restore energy and its overhead version for NvMR are also small, since restore only consists of retrieving the previously backed up copy of the CPU registers. Thus, none of these are shown in Figure 11. *stringsearch* is the worst performing benchmark. It has little opportunity for reducing backup energy costs as almost *90%* of its energy is consumed in making forward progress because it involves searching a list of strings in a file which does not incur a lot of idempotency violations. Note that there are forward progress and backup overheads in NvMR which add to the total energy consumption of a benchmark. There is an additional energy component when reclaiming is employed. The forward progress component in NvMR is higher than in Clank for a few benchmarks. It is because we include the energy spent on persisting data to a renamed address in forward progress. In Clank, this would be part of backup energy. Since reclaiming is carried out when an idemptency violation occurs and the map table is full, only a few benchmarks actually reclaim map table entries enough times to incur a considerable overhead. Thus, energy spent on reclaim is significant only for a few benchmarks like *blowfish, dijkstra* and *qsort*. Overall, the energy overhead of renaming and reclaiming in NvMR amounts to just *3%* of its total energy consumption; a small price to pay for a reduction of *185X* in the number of backups on average.

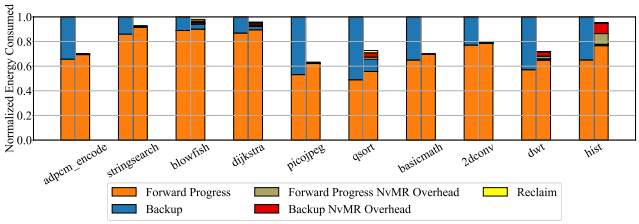


Figure 11: Breakdown of normalized energy consumption of Clank (left bars) and NvMR (right bars)

Structure	Mapping Table	OOP Buffer	OOP Region
Size	Infinite	128	2048

Table 4: System configuration of a simplified version of HOOP. The infinitely large map table has no energy and area overhead

6.2 NvMR vs HOOP

So far we have limited our discussion to intermittent systems that are idempotency violation-aware and backup data to avoid inconsistent program execution. However, there are transaction-based intermittent systems that store data updates to NVM in a redo log and re-execute the updates from the log during restore. In this section we compare NvMR against such a system, HOOP [6] mentioned in Section 2.

Table 4 lists the HOOP configuration used for producing the results presented in Figure 12. We run HOOP with the JIT and watchdog timer backup schemes mentioned in Section 5. The components, except for the mapping table (the infinite mapping table is an ideal scenario where the mapping does not consume any energy), are carefully sized to match the on-chip area and memory footprint overheads of NvMR with the configuration listed in Table 2. In case of JIT backup, NvMR saves on average *40%* more energy. Only with *stringsearch, picojpeg and basicmath* HOOP saves more energy than NvMR. These three benchmarks have high store locality (stores to the same cache block) which gives better opportunity to the OOP Buffer to pack data without filling up completely and reduce NVM writes. In case of watchdog timer backup, NvMR saves about *19.4%* more energy on average than HOOP. Since watchdog timer backup is a naive technique the energy saved is less than that in case of JIT backup.

6.3 Sensitivity Analysis

Energy efficiency of NvMR can vary based on the characteristics of the structures like map table cache, map table and free list. In order

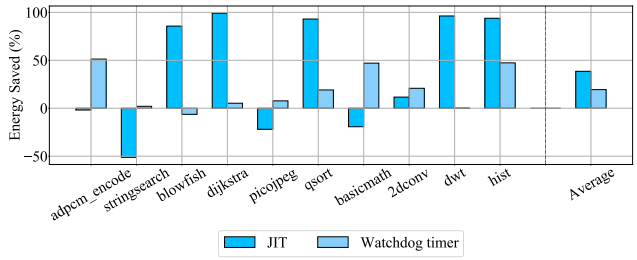


Figure 12: Energy saved in NvMR compared to HOOP

to simplify the discussion, we keep the number of entries of the free list fixed as per the worst-case scenario mentioned in Section 5 and vary the configuration of the rest. We also vary the supercapacitor size to show the benefits of NvMR at different supply capacities.

6.3.1 Map Table Cache. We use results for the JIT backup policy of all sensitivity study experiments. For the results shown in Figure 13a and Figure 13b, we use a map table with 4096 entries. Figure 13a shows how energy saved varies with the number of entries in the map table cache. The associativity is kept fixed at 2. We can observe that there is a steady increase in energy savings with a larger map table cache because the number of backups due to eviction of a dirty map table cache entry decreases as the number of entries increase. Figure 13b shows the trend when the associativity of map table cache varies. We use a small map table cache with 32 entries because as the map table cache size increases, the impact of associativity on the percentage of energy saved diminishes. Percentage of energy saved does not vary a lot with associativity. In fact, after an associativity of 4 only about 0.2% energy is saved until a full associativity (indicated as '0' on the x-axis of Figure 13b).

6.3.2 Map Table. For Figure 13c , we chose a map table cache with 512 entries and associativity of 8. In Figure 13c, we can see that there is a small jump of about 1% when the number of map table entries increased 8-fold from 1024 to 8192. These savings are coming from the reduction in the number of backups due to an idempotency violation when the map table is full, which occurs in the case of only a few of the benchmarks - *blowfish, dijkstra, qsort* and *dwt*. Since the amount of Flash needed for a small percentage increase in energy savings is quite high, we use a map table that has 4096 entries for our results in Section 6.1, a reasonable choice considering the limited amount of Flash on energy-harvesting devices.

6.3.3 Capacitor Size. We vary the size of the supercapacitor to analyze the benefit of NvMR with different supply capacities. The default capacitor size for our observations so far has been 100mF as was used in [8]. We run benchmarks on NvMR, configured according to Table 2, with smaller capacitor sizes 7.5mF [8] and 500µF [9][8]. As shown in Figure 13d, there is an increase of about *7%* when the capacitor size increases from 500µF to 7.5mF. However, the growth in energy savings slows down (*2%*) as the capacitor size increases to 100mF from 7.5mF. The primary reason behind this trend is that as the capacitor sizes get bigger, active periods get longer[9]. It results in an increase in the number of idempotency violations until a capacitor size of 7.5mF, after which the growth slows down. In fact, the number of idempotency violations increases by about *14%* from 500µF to 7.5mF and *3%* from 7.5mF to 100mF. NvMR uses this opportunity to reduce the number of backups due to idempotency violations and save more energy than Clank.

6.4 Reclaim vs. No Reclaim

Reclaiming is used to reduce the number of backups due to an idempotency violation when the map table is full. However, out of the benchmarks studied, only a few face this issue with the configuration in Table 2. As we can see from Figure 14, on average, reclaiming yields only about 1% better energy savings compared to the "no reclaim" case. However, it does not eat into the energy savings obtained from NvMR. Only *qsort* (9%) and *dwt* (1%) benefit from reclaiming. Only in the case of *dijkstra*, *blowfish* and *2dconv*, reclaiming results in a little higher energy consumption. Reclaiming is useful when the MCU board at hand does not come with a large NVM so that allocating a large map table and free list becomes out of question. Our experiments show a map table with 1024 entries, reclaiming saves around *9%* more energy than the "no reclaim" case.

6.5 NvMR Overhead

The sizes of the software and hardware structures in NvMR can be configured to fit the requirements of the target application, only to be constrained by the available memory on board (both volatile and non-volatile). The memory footprint of the reserved region is about *6%* of the Flash on-board (2MB). A Flash of such size falls within the range of those found in commercially available MCU boards [37]. It is worth mentioning that renaming in NvMR reduces NVM wear-out. Our experiments show that the maximum number of writes to a single NVM location reduces by 80.8% (averaged across all benchmarks) in NvMR compared to Clank.

The map table and the free list are software structures in NVM. Thus, the hardware cost of NvMR only consists of the map table cache. We use McPAT [21] to find an estimate of the hardware overhead of NvMR compared to our version of Clank. For the configuration listed in Table 2 on-chip area overhead of the map table cache in NvMR is about *6%*.

7 RELATED WORK

In this section, we briefly discuss a few of the previous works which are related to our work. We note the similarities and point out the key differences between these works and the work proposed in this paper.

7.1 Memory Renaming

Memory renaming has been suggested in earlier works mainly to address the problem of memory disambiguation and mitigate performance deterioration due to it [14, 19, 30, 36]. For instance, [45] renamed the memory location of a load instruction in the rename stage of the pipeline by using a memory dependence prediction table and a value file. This work tries to mitigate the performance

[8]The capacitor sizes are in mF - µF range because we are using Flash which is the most commonly found NVM on commercial micro-controller(MCU) boards. FRAM consumes three orders of magnitude less energy and can operate with a capacitance in nF range.

[9]Active period = Cycles between power failures during which an intermittent system wakes up, executes a given program and backs up data

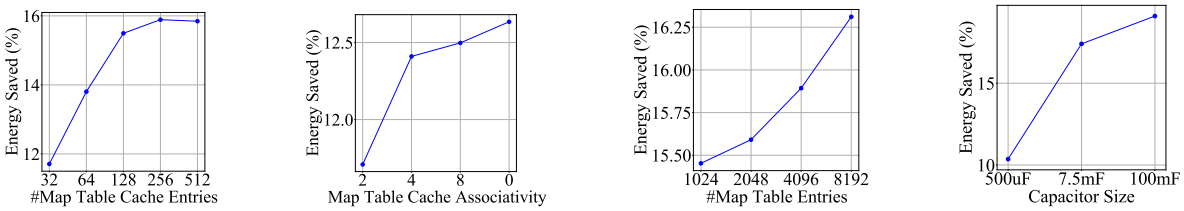


Figure 13: Energy saved with different map table cache and map table configurations and capacitor sizes

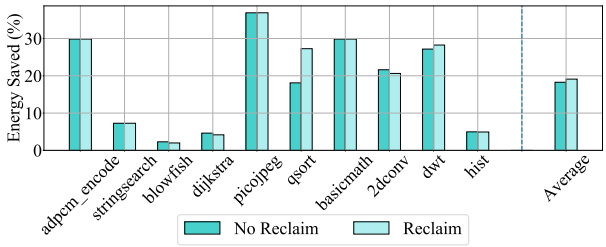


Figure 14: Energy saved in NvMR with and without reclaim

degradation due to RAW dependency between a store and a load in an out-of-order processor. In contrast, NvMR eliminates WAR and WAW persist dependencies in an intermittent system.

7.2 Intermittent Computing

[7, 22] put the onus of correct intermittent execution on the programmer. It is the duty of the programmer to ensure correctness by breaking down the program statically into small tasks that are atomic and potentially maintain idempotency [25]. Backups are allowed only at the boundary between two tasks. In contrast, NvMR does not require a programmer to modify the application code. NvMR also detects idempotent violations in hardware similar to [16], but it bypasses the backups deemed necessary in that work by renaming NVM addresses. [48] also has a write-back cache to take advantage of data locality. However, it needs on-chip non-volatile storage (e.g. NVFF) to log dirty data and has to do a backup when there is a register spill. In NvMR, if the map table gets full, NvMR can reclaim mappings to continue renaming.

8 CONCLUSION

Although prior works have proposed efficient backup schemes so that much of the energy harvested is used to do useful work in intermittent systems, most of these works need to do backups due to idempotency violations in order to maintain consistent system state. We argue that backups in energy harvesting systems should be due to energy or hardware constraints and not because of idempotency violations. We characterize the unique persist dependencies in intermittent program execution and show that backups due to idempotency violations can be eliminated by renaming the program addresses of the violating read-dominated stores. Based on these insights, we present a hardware architecture, NvMR, that implements renaming in an energy-efficient manner. As a result, NvMR gets more useful work done with the harvested energy than state-of-the-art intermittent architectures and decouples the correct execution of a program from the backup scheme.

ACKNOWLEDGEMENTS

We thank reviewers for their invaluable feedback. This work is supported by the Wisconsin Alumni Research Foundation and NSF/Intel under award No. 2010830.

REFERENCES

[1] ARM Ltd. 2012. *Cortex-M0+ Technical Reference Manual – Arm Developer.* ARM Ltd.

[2] Domenico Balsamo, Alex S. Weddell, Anup Das, Alberto Rodriguez Arreola, Davide Brunelli, Bashir M. Al-Hashimi, Geoff V. Merrett, and Luca Benini. 2016. Hibernus++: A Self-Calibrating and Adaptive System for Transiently-Powered Embedded Devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2016). https://doi.org/10.1109/TCAD.2016.2547919

[3] Domenico Balsamo, Alex S. Weddell, Geoff V. Merrett, Bashir M. Al-Hashimi, Davide Brunelli, and Luca Benini. 2015. Hibernus: Sustaining Computation During Intermittent Supply for Energy-Harvesting Systems. *IEEE Embedded Systems Letters* (2015). https://doi.org/10.1109/LES.2014.2371494

[4] Kevin Barker, Thomas Benson, Dan Campbell, David Ediger, Roberto Gioiosa, Adolfy Hoisie, Darren Kerbyson, Joseph Manzano, Andres Marquez, Leon Song, Nathan Tallent, and Antonino Tumeo. 2013. *PERFECT (Power Efficiency Revolution For Embedded Computing Technologies) Benchmark Suite Manual.* Pacific Northwest National Laboratory and Georgia Tech Research Institute. http://hpc.pnnl.gov/projects/PERFECT/.

[5] P. Bogdan, M. Pajic, P. P. Pande, and V. Raghunathan. 2016. Making the internet-of-things a reality: From smart models, sensing and actuation to energy-efficient architectures. In *IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis.*

[6] Miao Cai, Chance C. Coats, and Jian Huang. 2020. HOOP: Efficient Hardware-Assisted Out-of-Place Update for Non-Volatile Memory. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA).* 584–596. https://doi.org/10.1109/ISCA45697.2020.00055

[7] Alexei Colin and Brandon Lucia. 2016. Chain: Tasks and Channels for Reliable Intermittent Programs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Amsterdam, Netherlands) *(OOPSLA 2016).* ACM, New York, NY, USA, 514–530. https://doi.org/10.1145/2983990.2983995

[8] Alexei Colin, Emily Ruppel, and Brandon Lucia. 2018. A Reconfigurable Energy Storage Architecture for Energy-harvesting Devices. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Williamsburg, VA, USA) *(ASPLOS '18).* ACM, New York, NY, USA, 767–781. https://doi.org/10.1145/3173162.3173210

[9] Samuel DeBruin, Bradford Campbell, and Prabal Dutta. 2013. Monjolo: An Energy-Harvesting Energy Meter Architecture. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems* (Roma, Italy) *(SenSys '13).* Association for Computing Machinery, New York, NY, USA, Article 18, 14 pages. https://doi.org/10.1145/2517351.2517363

[10] Graham Gobieski, Amolak Nagi, Nathan Serafin, Mehmet Meric Isgenc, Nathan Beckmann, and Brandon Lucia. 2019. MANIC: A vector-dataflow architecture for ultra-low-power embedded systems. In *Proceedings of the Annual International Symposium on Microarchitecture, MICRO.* IEEE Computer Society, New York, NY, USA, 670–684. https://doi.org/10.1145/3352460.3358277

[11] Shyamnath Gollakota, Matthew S. Reynolds, Joshua R. Smith, and David J. Wetherall. 2014. The emergence of rf-powered computing. *Computer* 47, 1 (jan 2014),

32–39. https://doi.org/10.1109/MC.2013.404

[12] K. Gudan, S. Chemishkian, J. J. Hull, M. S. Reynolds, and S. Thomas. 2012. Feasibility of wireless sensors using ambient 2.4GHz RF energy. In *SENSORS, 2012 IEEE*. 1–4.

[13] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trever Mudge, and Richard B. Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE International Workshop on Workload Characterization (IISWC)*. 3–14.

[14] Lance Hammond, Benedict A. Hubbert, Michael Siu, Manohar K. Prabhu, Michael Chen, and Kunle Olukotun. 2000. The Stanford Hydra CMP. *IEEE Micro* 20, 2 (March 2000), 71–84. https://doi.org/10.1109/40.848474

[15] Josiah Hester, Kevin Storer, and Jacob Sorber. 2017. Timely Execution on Intermittently Powered Batteryless Sensors. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems* (Delft, Netherlands) *(SenSys '17)*. Association for Computing Machinery, New York, NY, USA, Article 17, 13 pages. https://doi.org/10.1145/3131672.3131673

[16] Matthew Hicks. 2017. Clank: Architectural Support for Intermittent Computation. In *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*. 228–240.

[17] Hrishikesh Jayakumar, Arnab Raha, Woo Suk Lee, and Vijay Raghunathan. 2015. Quick Recall. *ACM Journal on Emerging Technologies in Computing Systems* 12, 1 (2015), 1–19. https://doi.org/10.1145/2700249

[18] Stephen Jourdan, Ronny Ronen, Michael Bekerman, Bishara Shomar, and Adi Yoaz. 1998. A Novel Renaming Scheme to Exploit Value Temporal Locality Through Physical Register Reuse and Unification. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture* (Dallas, Texas, USA) *(MICRO 31)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 216–225. http://dl.acm.org/citation.cfm?id=290940.290985

[19] Stephen Jourdan, Ronny Ronen, Michael Bekerman, Bishara Shomar, and Adi Yoaz. 1998. A Novel Renaming Scheme to Exploit Value Temporal Locality through Physical Register Reuse and Unification. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture* (Dallas, Texas, USA) *(MICRO 31)*. IEEE Computer Society Press, Washington, DC, USA, 216–225.

[20] Aasheesh Kolli, Vaibhav Gogte, Ali Saidi, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. 2017. Language-Level Persistency. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada) *(ISCA '17)*. Association for Computing Machinery, New York, NY, USA, 481–493. https://doi.org/10.1145/3079856.3080229

[21] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. 469–480.

[22] Brandon Lucia and Benjamin Ransford. 2015. A Simpler, Safer Programming and Execution Model for Intermittent Systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. ACM, New York, NY, USA, 575–585. https://doi.org/10.1145/2737924.2737978

[23] Kaisheng Ma, Xueqing Li, Srivatsa Rangachar Srinivasa, Yongpan Liu, John Sampson, Yuan Xie, and Vijaykrishnan Narayanan. 2017. Spendthrift: Machine learning based resource and frequency scaling for ambient energy harvesting nonvolatile processors. In *Proceedings of the Asia and South Pacific Design Automation Conference, ASP-DAC*. Institute of Electrical and Electronics Engineers Inc., 678–683. https://doi.org/10.1109/ASPDAC.2017.7858402

[24] K. Ma, X. Li, S. R. Srinivasa, Y. Liu, J. Sampson, Y. Xie, and V. Narayanan. 2017. Spendthrift: Machine learning based resource and frequency scaling for ambient energy harvesting nonvolatile processors. In *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*. 678–683. https://doi.org/10.1109/ASPDAC.2017.7858402

[25] Kiwan Maeng, Alexei Colin, and Brandon Lucia. 2017. Alpaca: Intermittent Execution without Checkpoints. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 96 (Oct. 2017), 30 pages. https://doi.org/10.1145/3133920

[26] Kiwan Maeng and Brandon Lucia. 2007. Adaptive dynamic checkpointing for safe efficient intermittent computing. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018*. 129–144.

[27] Scott A. Mahlke, William Y. Chen, Roger A. Bringmann, Richard E. Hank, Wen Mei W. Hwu, B. Ramakrishna Rau, and Michael S. Schlansker. 1993. Sentinel Scheduling: A Model for Compiler-Controlled Speculative Execution. *ACM Transactions on Computer Systems (TOCS)* 11, 4 (jan 1993), 376–408. https://doi.org/10.1145/161541.159765

[28] Josiah Hester Matthew Furlong. 2017. *PERSISTLab/BatterylessSim*. https://github.com/PERSISTLab/BatterylessSim

[29] Geoff V. Merrett and Bashir M. Al-Hashimi. 2017. Energy-driven computing: Rethinking the design of energy harvesting systems. In *Proceedings of the 2017 Design, Automation and Test in Europe, DATE 2017*. https://doi.org/10.23919/DATE.2017.7927130

[30] Andreas Moshovos and Gurindar S. Sohi. 1997. Streamlining Inter-operation Memory Communication via Data Dependence Prediction. In *In Proceedings of the 30th International Symposium on Microarchitecture*.

[31] Naveen Muralimanohar, Rajeev Balasubramonian, and Norm Jouppi. 2007. Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0. In *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*. 3–14. https://doi.org/10.1109/MICRO.2007.33

[32] A. N. Parks, A. P. Sample, Y. Zhao, and J. R. Smith. 2013. A wireless sensing platform utilizing ambient RF energy. In *2013 IEEE Topical Conference on Biomedical Wireless Technologies, Networks, and Sensing Systems*. 154–156.

[33] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. 2014. Memory Persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture* (Minneapolis, Minnesota, USA) *(ISCA '14)*. IEEE Press, 265–276.

[34] Benjamin Ransford and Brandon Lucia. 2014. NVM is a broken Time Machine. In *Proceedings of the 2014 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*.

[35] Benjamin Ransford, Jacob Sorber, and Kevin Fu. 2011. Mementos: System Support for Long-Running Computation on RFID-Scale Devices. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Newport Beach, California, USA) *(ASPLOS XVI)*. Association for Computing Machinery, New York, NY, USA, 159–170. https://doi.org/10.1145/1950365.1950386

[36] Glenn Reinman, Brad Calder, Dean Tullsen, Gary Tyson, and Todd Austin. 1999. Classifying Load and Store Instructions for Memory Renaming. In *Proceedings of the 13th International Conference on Supercomputing* (Rhodes, Greece) *(ICS '99)*. Association for Computing Machinery, New York, NY, USA, 399–407. https://doi.org/10.1145/305138.305218

[37] Renesas Microcomputers 2020. *Arm® Cortex®-M0+ Ultra-low Power MCU Based on SOTB™ Process Technology*. Renesas Microcomputers. Rev. 1.10.

[38] Ohad Rodeh. 2008. B-Trees, Shadowing, and Clones. *ACM Trans. Storage* 3, 4, Article 2 (Feb. 2008), 27 pages. https://doi.org/10.1145/1326542.1326544

[39] Joshua San Miguel, Karthik Ganesan, Mario Badr, Steven Xia, Rose Li, Hsuan Hsiao, and Natalie Enright Jerger. 2018. The EH Model: Early Design Space Exploration of Intermittent Processor Architectures. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*.

[40] STMicroelectronics 2017. *Access line ultra-low-power 32-bit MCU Arm®-based Cortex®-M0+, up to 16KB Flash, 2KB SRAM, 512B EEPROM, ADC*. STMicroelectronics. https://www.st.com/resource/en/datasheet/stm32l011k4.pdf Rev. 5.

[41] Milijana Surbatovich, Limin Jia, and Brandon Lucia. 2021. Automatically Enforcing Fresh and Consistent Inputs in Intermittent Systems. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 851–866. https://doi.org/10.1145/3453483.3454081

[42] Milijana Surbatovich, Brandon Lucia, and Limin Jia. 2020. Towards a Formal Foundation of Intermittent Computing. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 163 (Nov. 2020), 31 pages. https://doi.org/10.1145/3428231

[43] Texas Instruments Inc. 2021. *MSP430FR599x, MSP430FR596x Mixed-Signal Microcontrollers datasheet*. Texas Instruments Inc.

[44] Gary S. Tyson and Todd M. Austin. 1997. Improving the Accuracy and Performance of Memory Communication Through Renaming. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture* (Research Triangle Park, North Carolina, USA) *(MICRO 30)*. IEEE Computer Society, Washington, DC, USA, 218–227. http://dl.acm.org/citation.cfm?id=266800.266821

[45] Gary S Tyson and Todd M. Austin. 1997. Improving the accuracy and performance of memory communication through renaming. In *Proceedings of the Annual International Symposium on Microarchitecture*. 218–227. https://doi.org/10.1109/MICRO.1997.645812

[46] Joel Van Der Woude and Matthew Hicks. 2016. Intermittent Computation Without Hardware Support or Programmer Intervention. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, Berkeley, CA, USA, 17–32. http://dl.acm.org/citation.cfm?id=3026877.3026880

[47] Xingbo Wu, Wenguang Wang, and Song Jiang. 2015. TotalCOW: Unleash the Power of Copy-On-Write for Thin-Provisioned Containers. In *Proceedings of the 6th Asia-Pacific Workshop on Systems* (Tokyo, Japan) *(APSys '15)*. Association for Computing Machinery, New York, NY, USA, Article 15, 7 pages. https://doi.org/10.1145/2797022.2797024

[48] Jianping Zeng, Jongouk Choi, Xinwei Fu, Ajay Paddayuru Shreepathi, Dongyoon Lee, Changwoo Min, and Changhee Jung. 2021. ReplayCache: Enabling Volatile Cachesfor Energy Harvesting Systems. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 170–182. https://doi.org/10.1145/3466752.3480102