WARio

efficient code generation for intermittent computing

Kortbeek, Vito; Ghosh, Souradip; Hester, Josiah; Campanoni, Simone; Pawełczak, Przemysław

DOI
10.1145/3519939.3523454

Publication date
2022

Document Version
Final published version

Published in
PLDI 2022 - Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation

Citation (APA)
Kortbeek, V., Ghosh, S., Hester, J., Campanoni, S., & Pawełczak, P. (2022). WARio: efficient code generation for intermittent computing. In R. Jhala, & I. Dillig (Eds.), *PLDI 2022 - Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (pp. 777-791). (Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)). Association for Computing Machinery (ACM). https://doi.org/10.1145/3519939.3523454

WARio: Efficient Code Generation for Intermittent Computing

Vito Kortbeek
Delft University of Technology
Delft, The Netherlands
v.kortbeek-1@tudelft.nl

Souradip Ghosh
Carnegie Mellon University
Pittsburgh, PA, USA
souradip@cmu.edu

Josiah Hester
Northwestern University
Evanston, IL, USA
josiah@northwestern.edu

Simone Campanoni
Northwestern University
Evanston, IL, USA
simonec@eecs.northwestern.edu

Przemysław Pawełczak
Delft University of Technology
Delft, The Netherlands
p.pawelczak@tudelft.nl

Abstract

Intermittently operating embedded computing platforms powered by energy harvesting require software frameworks to protect from errors caused by Write After Read (WAR) dependencies. A powerful method of code protection for systems with non-volatile main memory utilizes compiler analysis to insert a checkpoint inside each WAR violation in the code. However, such software frameworks are oblivious to the code structure—and therefore, inefficient—when many consecutive WAR violations exist. Our insight is that by transforming the input code, i.e., moving individual write operations from unique WARs close to each other, we can significantly reduce the number of checkpoints. This idea is the foundation for WARio: a set of compiler transformations for efficient code generation for intermittent computing. WARio, on average, reduces checkpoint overhead by 58%, and up to 88%, compared to the state of the art across various benchmarks.

CCS Concepts: • **Software and its engineering → Checkpoint / restart**; **Compilers**; **General programming languages**; • **Computer systems organization → Embedded systems**.

Keywords: intermittent computing, battery-free, compiler, embedded system, code transformation, optimization

ACM Reference Format:
Vito Kortbeek, Souradip Ghosh, Josiah Hester, Simone Campanoni, and Przemysław Pawełczak. 2022. WARio: Efficient Code Generation for Intermittent Computing. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '22), June 13–17, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3519939.3523454

PLDI '22, June 13–17, 2022, San Diego, CA, USA

ACM ISBN 978-1-4503-9265-5/22/06.
https://doi.org/10.1145/3519939.3523454

1 Introduction

We have a long way to go to make Internet of Things (IoT) self-sustainable, self-recyclable, and carbon-neutral. Fortunately, the first steps to remove polluting components from IoT devices have already been taken, with batteries being the principal focus [15, 61]. The resulting battery-free systems [20, 51] are embedded devices powered by ambient energy (such as solar radiation or vibrations) where the energy is stored in small (super-) capacitors. So far many battery-free devices have been demonstrated, ranging from experimental handheld gaming platforms [13], networks for long-term, hard-to-reach infrastructure monitoring [1] to commercial wireless IoT tags [58]. One should expect more complex battery-free devices in the coming years.

Problem Statement. Capacitors hold orders of magnitude less energy than batteries, which means that their energy supply is intermittent as they must recharge. Therefore, power failures are common, causing computational intermittency [29]. Intermittent operation causes the computational state to be lost unless explicitly saved in Non-Volatile Memory before a power failure and restored afterward.

The obvious solution to maintain forward progress is to store the volatile computation state at predefined intervals (by number of clock cycles or by number of instructions) through *checkpoints*. However, copying all volatile memory regions to NV memory is inefficient (as unmodified memory regions would also be re-saved). Another option is to use a fully non-volatile processor architecture [30, 52], with all components realized with non-volatile logic gates. Sadly, fully non-volatile microcontroller architectures are disadvantageous compared to their volatile counterparts, as it is difficult to optimize cost, access speed, number of read/write cycles, and density for these gates. Instead, processor architectures for intermittently-powered systems often rely on

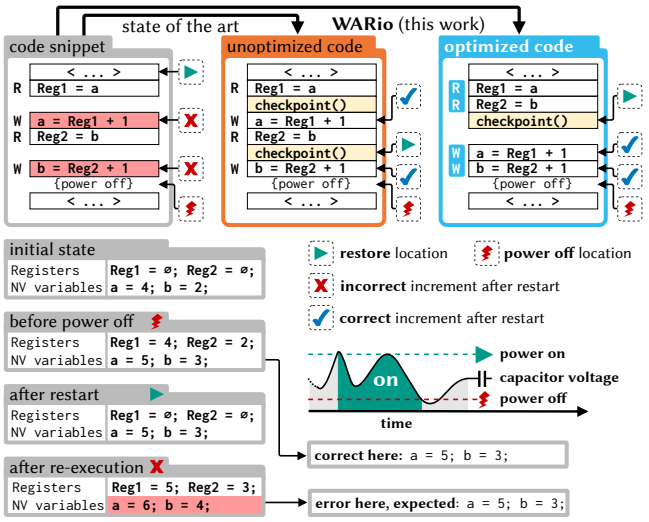


Figure 1. Three versions of the same code snippet demonstrating Non-Volatile Memory corruption, its mitigation, and our optimization. A checkpoint records only the registers (Reg1 and Reg2). The variables (a and b) are in Non-Volatile Memory (NVM) and *not* restored after a power failure. The unprotected code (left) executes until the power failure, reading from and writing to the Non-Volatile (NV) variables. A restart does not undo any modifications to NVM, resulting in incorrect re-execution caused by a Write After Read to the NV variables. By placing a checkpoint of the registers between the *read* (R) and *write* (W) of a WAR, state-of-the-art systems such as Ratchet [57] (center figure, unoptimized code) avoid this memory corruption caused by re-execution. WARio (our work) aims to reduce the number of required checkpoints by *clustering* writes to NVM, reducing the number of required checkpoints (right figure, optimized code).

mixed volatility memory [26, 32, 57][1]. The processor houses an on-board, byte-addressable Non-Volatile Memory, often in the form of Ferroelectric Random Access Memory (FRAM) or Magnetoresistive Random Access Memory (MRAM). These mixed-memory architectures are still uncommon, but are actively being developed [3, 55]. In these architectures, the main memory is non-volatile, and only the CPU registers and configurations (e.g., peripheral settings and operating frequency) are volatile. This way, only the registers need to be saved at a *checkpoint* to maintain forward progress across power failures. Relying on NV memories significantly reduces the cost of a single checkpoint by saving only the (live) registers. However, state-of-the-art static solutions using NV main memory require frequent checkpoints, often at the basic block level. Moreover, selective (instruction-level) checkpoint placement must account for the unique problem present in contemporary (and future) Microcontroller Unit (MCU) architectures that use non-volatile main memory: *Non-Volatile memory corruption in variable manipulations with WAR dependencies caused by re-execution.* This problem, often referred to as a WAR violation [34, 57], was first observed in [48] and is schematically presented in Figure 1. Suppose the power fails after a 'Write' in a WAR operation. During re-execution, the 'Read' of that same WAR operation will read the newly written value instead of the original value. This is because the checkpoint restoration only restores the registers, not the memory. To prevent this memory corruption we must place a checkpoint before executing the 'Write' of a WAR operation. This way, re-execution starts after the 'Read' operation has already been completed. Throughout this paper, we use the term *WAR violation* (or simply WAR) to refer to these possible memory corruption locations which are caused by re-execution following a power failure. When we refer to *resolving a WAR*, we refer to the placement of a checkpoint between its 'Read' and 'Write' to create two distinct idempotent regions.

A state-of-the-art approach is to detect idempotent regions by looking for instruction sequences that perform a WAR to the same memory address, and placing checkpoints at the boundaries of these regions [57], Figure 1 (middle). Nonetheless, strategic checkpoint placement of [57], which performs this task automatically at compile time, does not perform any transformations to *reduce the number of introduced checkpoints* (which are often over-instrumented). Our fundamental insight is that in a code region with many consecutive WAR violations, moving the 'Write' operations belonging to these WARs to a later stage in the code, i.e., *clustering* them, will reduce the number of checkpoints needed, thereby increasing the performance of intermittent computing. The more consecutive (unrelated) WAR operations—the more benefit from checkpoint reordering, which reduces the execution time. This reduction is clearly seen in Figure 1 (right), *halving* the number of checkpoints inserted by [57] (Figure 1 (middle)). To implement the transformations mentioned above, we use NOELLE [36], an LLVM [41] plugin that uses alias analysis [4, 53] to compute a Program Dependence Graph (PDG) (among other information).

Our Contributions. We present WARio, **W**rite **A**fter **R**ead **I**ntermittent-computing **O**ptimizer, a set of compiler transformations for intermittently-executed programs to reduce checkpoint overhead. WARio builds upon the techniques introduced in Ratchet [57] and operates both in the middle end and the back end of the compiler. In the middle end, ① WARio introduces *two novel algorithms that cluster the 'Write' operations* of several WARs to reduce the required number of checkpoints. In the back end, ② WARio reduces the number of checkpoints by introducing a *hitting set algorithm to select the checkpoint locations to resolve back-end WARs* (in addition to the existing hitting set in the middle end [57]) and by ③ *protecting stack pointer modifications* in a novel way that requires fewer checkpoints.

[1]Cache architectures are limited in most embedded architectures compared to their desktop counterparts and not considered [26, 32, 57]

The transformation steps ①–③ of WARio reduce checkpointing overhead for continuous-checkpointing-based intermittent computation. Compared to Ratchet [57], a state of the art system, WARio reduces the checkpoint overhead by up to 88%, and on average by 58%, considering a broad set of software benchmarks.

2 Battery-Free Intermittent Computing

Eliminating batteries from embedded computing platforms brings many benefits, such as reducing the size of the overall system and lowering its environmental footprint. Many systems that are normally powered by batteries have been shown to operate battery-free. These include battery-free handheld gaming platform [13], battery-free phone [54], battery-free eye tracking [27] and various forms of battery-free wireless sensors such as [1, 12, 24, 46, 56].

Battery-free operation using harvested ambient energy might cause power supply intermittency to the device. To protect the code operating on such device different classes of software systems were developed that support the correctness of intermittent operation. We describe them here.

Hardware Systems. In hardware-based checkpointing systems, checkpoints that snapshot certain memory regions of a battery-free device can be triggered either by a capacitor voltage monitor [8, 23, 25, 49] or by an external timer, and not from within the code itself (as shown in Figure 1). Another hardware system is based on monitoring `store` and `load` addresses of a battery-free system through an external hardware module to detect memory inconsistencies [21]. All these hardware systems, although performing their intended task of computation consistency protection, are either not widely available [21], copy large memory regions (as they do not track which addresses were accessed), or are imprecise [8, 23, 49] causing potential incorrect recovery (see also [10, Section 2.2.2]). For this reason software-only systems are still a preferred alternative to guarantee computation consistency protection.

Software Systems with Manual Code Adaptation. Certain software systems for intermittent execution require manual transformation of the original code using a special API. Therein, input code must be transformed to special sections, i.e. tasks—atomic code blocks matched to a specific energy budget—that when interrupted by power failure will restart from the beginning of the task definition. Such frameworks include [35, 50, 60]. Manual code transformation has its price as it requires extra work from a programmer [26, Section 5.4]. Moreover, a programmer must learn a new Domain-Specific Language (DSL). Also, a programmer needs to dimension the idempotent code, i.e., tasks, to a specific energy budget—requiring the programmer to rewrite the tasks when the target energy harvesting environment changes (say, from average available power on time from 2 s to 0.5 s).

Software Systems with Automatic Code Adaptation. These software systems can be further divided into code-oblivious [26] and code-aware [57]. In *code-oblivious systems*, checkpoints copy volatile memory to a NV region with extra information. As a consequence, such checkpoints have a significant checkpoint and restore time. Checkpoint systems that also include logging [26, 32] introduce large overhead to track memory accesses dynamically. In contrast, in *code-aware systems*, checkpoints are inserted statically at compile time at specific regions of the code, i.e. at read locations of every WAR operation (as in Figure 1 (center figure)). This makes checkpoint creation and restoration faster compared to code-oblivious systems. Since the checkpoint saves and restores only the information in use, i.e., the live registers and nothing extra. Our *conjecture regarding software development for intermittently-powered devices* is as follows. Preferred software systems will be the ones that are (i) code-aware, (ii) automatic, (iii) code-transforming, and (iv) compiler-based. Simply, such software systems will require less extra data being copied, and will not require specialized hardware support. Such systems however are sill far from ideal.

3 WARio System Design

Addressing the problem presented in Section 1 we present WARio. During compilation, WARio performs multiple optimizations targeted at reducing the number of WAR violations in the C code. WARio possesses the following features. ❶ **Support for General Purpose C Programs:** WARio takes a regular embedded C code and automatically transforms it to a WAR-protected executable; ❷ **Oblivious to Energy Conditions:** No prior information on the battery-free system's energy use or input harvested energy is needed prior (and during) compilation into a WAR-protected executable; ❸ **Support for Short Device Activity Times:** WARio guarantees forward progress at short device activity times, i.e. in the order of tens of milliseconds; ❹ **No Programmer Involvement:** WARio does not expect to restructure the program manually to help resolve any WAR dependency; and ❺ **Interrupt Support:** During checkpoint placement WARio makes sure that there can be no WAR violations caused by interrupts pushing information to the stack.

3.1 WARio Architecture

WARio targets the following platform: (i) a single processor embedded system (MCU); (ii) direct physical memory access, i.e., no virtual memory; (iii) no data cache, (iv) register access/'bare metal', i.e., no operating system; and (v) non-volatile byte-addressable main memory.

WARio's architecture consists of a set of Intermediate Representation (IR)-based compiler transformations executed in a specific order, as presented in Figure 2. All of the components of WARio are described in detail below.

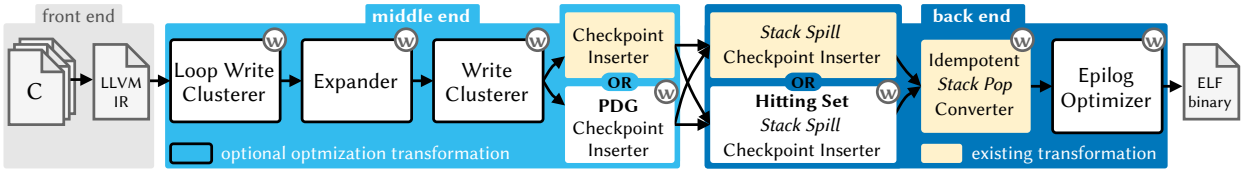


Figure 2. WARio architecture. Input plain C code is transformed, through a set of middle and back end compiler transformations described in Section 3.1, to an output Executable and Linkable Format (ELF) binary file that can be executed (guaranteeing no NVM corruption caused by WARs) on an intermittently-powered system. The complete WARio system consists of all the transformations marked with a ⓦ; other combinations are used to evaluate performance of individual transformations in Section 5.2.1. The transformations marked as *existing* where introduced in prior work [57]. The transformations marked as optional are not needed to avoid WAR violations, but improve the performance by reducing the number of inserted checkpoints.

3.1.1 WARio Front End. WARio takes the C code of a project aimed to be run on a intermittently-powered device and converts it to LLVM IR, per each C source file. Subsequently, WARio merges individual IR files into a single (combined) IR of the whole project. We note that both these steps are standard front end compiler transformations (marked as the gray area in Figure 2).

3.1.2 WARio Middle End. The core tasks performed by WARio are executed in the middle end. Each of the steps (listed within light blue area in Figure 2) is explained below.

Loop Write Clusterer. This transformation aims at reducing the number of checkpoints in a loop that contains one or more WAR violations. Algorithm 1 shows the pseudocode of this transformation, and Figure 3 the resulting IR after each step. Both figures are used to explain the transformation in detail and provide a visual example.

Let us take as an example the unmodified loop code snippet in Figure 3. Directly inserting checkpoints, represented by the orange box, results in one checkpoint per iteration i. After applying the `Loop Write Clusterer` transformation, the loop requires only i/N checkpoints when executing, where N is the unroll factor used during the transformation, provided during compilation[2]. It does so by postponing write operations to NVM until the end of the unrolled loop—essentially combining the checkpoints required for N iterations of the loop into a single checkpoint. First, the transformation analyzes the input code using a PDG analyzer, such as [36], to collect all the memory dependencies in the program. The transformation then collects all loops in the program (denoted as L_{all} in Algorithm 1). For each input loop $L \in L_{\text{all}}$ the `Loop Write Clusterer` checks whether the loop is a candidate to be unrolled.

► ***Candidate Selection:*** Not all loops are candidates to have their writes clustered. Most notably, to be a candidate (Line 3 in Algorithm 1), the loop must contain at least one WAR violation to cluster (Line 11). Otherwise, the loop will not have any checkpoints to remove. Additionally, the write cluster insertion point (the destination of the to-be-moved WAR `store` instructions, i.e., the *loop latch*) must post-dominate all the relocated `store` instructions in order not to change the semantics of the loop (Line 13). The final requirement is that the loop does not contain any function calls, as those implicitly cause checkpoints hindering our ability to cluster the writes (Line 11).

► ***Loop Unrolling:*** If a loop is a candidate, Loop Write Clusterer unrolls it N times (Line 4 in Algorithm 1). The IR resulting from the unroll step is shown in Figure 3—`UnrollLoop` for an unroll factor of $N = 3$.

► ***Loop Analysis:*** Loop analysis is a necessary operation of `Loop Write Clusterer`, as simply moving all the writes to the loop latch is insufficient to retain the loop semantics. Let us therefore proceed with introducing the analysis steps (Line 17 in Algorithm 1) that will be needed to perform correct code transformation (Line 24 in Algorithm 1). The first step is the obtainment of loop dependency graph (Line 18 in Algorithm 1), from which the WAR and Read After Write (RAW) dependencies are obtained (Line 19 and Line 20 in Algorithm 1, respectively).

► ***Clustering WAR Writes:*** Unrolled loops, denoted as L', are passed for analysis using the PDG information (Line 5 in Algorithm 1), which are later on transformed (Line 6 in Algorithm 1) resulting in a set of WARs that are postponed, resulting in moved WAR writes (`store` instructions) shown in Figure 3—`ClusterWarWrites`.

► ***Early-exit Handing:*** When moving all writes to the insertion point, i.e., the loop latch, WARio potentially skips writing those values to NVM due to early exits, e.g., exits introduced due to unrolling. The transformation must guarantee that any early exit (`ModifyExits` in Algorithm 1) that follows a postponed write contains a copy of that postponed write. Otherwise, exiting a loop early (by reaching the desired number of iterations during execution before the end of the unrolled loop) would not execute the postponed write to NVM, invalidating the program execution. Figure 3—`ModifyEarlyExits` shows the addition of these postponed writes (`store` instructions) to the early exits.

[2]The effect of N on the unrolling effectiveness will be a part of WARio evaluation presented in Section 5.2.4

► ***Dependent Read Handling:*** When postponing all the writes to the loop latch, WARio might attempt to move write instructions past reads that depend on them, for example, due to unrolled loop-carried dependencies. First, the transformation collects the read instructions that might depend on a preceding write instruction, using RAW dependency information collected earlier through the PDG. If any of the reads depend on one or more of the postponed writes that are now no longer dominating the read (i.e., they now happen after the read), they would result in reading incorrect information. Therefore, if the read *may* depend on a postponed write, a runtime check is inserted that compares the source address of the read (`load`) instruction and the destination address of the postponed write (`store`) instruction (Line 37 in Algorithm 1). If these are equal, the read is skipped (i.e., the value is not retrieved from memory), and the register containing the content of the postponed write is copied into the read destination (Line 38 in Algorithm 1). On the other hand, if the addresses are not equal, the original read is performed. It might be the case that a read instruction may be dependent on multiple writes. In this case, the transformation adds checks for each of the writes, passing its output as input to the next check as shown in the `InstrumentReads` procedure in Algorithm 1. Figure 3—`InstrumentReads`, shows an example where the `load` of variable *c* *may* depend on the `store` to variables *a* and *b*, which were postponed (worst case). Adding a runtime check introduces overhead, but it is minimal compared to the time it takes to perform a complete checkpoint. However, there is a break-even point as the number of checks added to each read instruction grows depending on the number of aliasing writes before it.

► ***Checkpoint Placement:*** To illustrate the effect of the `Loop Write Clusterer`, the last (dark blue) box in Figure 3, shows the final loop IR with the addition of checkpoints. When the loop is executing, the three iterations from the original loop (now unrolled), containing the **three WAR violations**, are resolved with only **one checkpoint** instead of three.

► ***Correctness:*** When clustering—and therefore moving—the WAR writes, we need to take appropriate steps to maintain correctness. First, when moving a write to a later unrolled loop iteration, we must ensure that the postponed writes are written to NVM when the unrolled loop terminates early. The *Early-exit Handling* step guarantees that all writes are executed by adding writebacks to every loop exit. Second, when moving a write, we have to resolve all reads that *may* depend on it, i.e., attempt to read memory from the same address. The *Dependent Read Handling* step assures that no incorrect read will occur by canceling the write rescheduling or adding runtime checks and handling to aliasing reads. Together, these steps force the `Loop Write Clusterer` transformation to be conservative and semantically correct.

Algorithm 1: `Loop Write Clusterer`

```
1  Algorithm LoopWriteCluster() :
2      for L ∈ L_all do                          // Go through all program's loops
3          if IsCandidate(L) then                                   // See Line 7
4              L' ← UnrollLoop(L, N)                    // See Section 3.1.2
5              W_s, R_s, E ← Analyze(L')                            // See Line 17
6              Transform(W_s, R_s, E)                               // See Line 24
7  Procedure IsCandidate(L) :
8      D ← FindDependencies (L)                                     // Use the PDG
9      W ← FindWARs (D)                                    // Find initial WARs
10     C ← FindFunctionCalls (L)                 // Find any function calls
11     if W ≠ ∅ and C = ∅ then          // If loop has WARs and no calls
12         for w ∈ W do                                  // For each WAR violation
13             if L_latch not post-dominates w_write then
14                 return false                          // Loop is not a candidate
15         return true                                   // Loop is a candidate
16     return false                                      // Loop is not a candidate
17 Procedure Analyze(L) :
18     D ← FindDependencies (L)                                     // Use the PDG
19     W ← FindWARs (D)                                // Extract WAR violations
20     R ← FindRAWs (D)                              // Extract RAW dependencies
21     R_s ← ReadsToResolve (W, R)  // Reads dependent on WAR writes
22     E ← ExitsToModify (W_s)                          // Exit edges in the loop
23     return W_s, R_s, E
24 Procedure Transform(W_s, R_s, E) :
25     PostponeWARs (W_s)                // Move the WAR writes to loop latch
26     ModifyExits (E, W_s)                        // Handle early exits (Line 28)
27     InstrumentReads (R_s)             // Handle dependent reads (Line 32)
28 Procedure ModifyExits(E, W_s) :
29     for w ∈ W_s do                                // For each WAR violation
           // Exit edges that follow the original write location
30         for e ∈ ExitEdges (E, w_write) do
31             copy w_write → e                      // Insert copy of write in exit
32 Procedure InstrumentReads(R_s) :
33     for r ∈ R_s do                    // Go through all the dependent reads
34         r_final ← r                          // Track the last instrumented read
35         for w ∈ AliasingWrites (r) do             // Writes that alias read
36             if r depends on w then
                   // Create new instructions to handle the read
37                 cmp_inst = NewCompareInstruction(r_addr, w_addr)
38                 sel_inst = NewSelectInstruction(sel_inst, w_src, r_final)
39                 r_final = sel_inst                // Track the last read select
40         for u ∈ usages r do
41             replace u with r_final          // Replace with final read select
```

Expander. A large number of checkpoints are caused by function calls. Each function call must perform a checkpoint if it can modify any data on the callee stack. However, more significantly, each function (regardless of the number of arguments) needs at least one checkpoint when returning from a function that uses stack memory. The reason for this is that an interrupt might trigger at any time, and the Interrupt Service Routine (ISR) will automatically push (write) information on the stack causing a WAR violations (Section 3.1.3—Paragraph `Epilog Optmizer`). Strategically inlining functions more aggressively than usual results in fewer checkpoints caused by function calls and returns. In addition, it aids the succeeding transformation by not having a forced checkpoint location due to the function call.

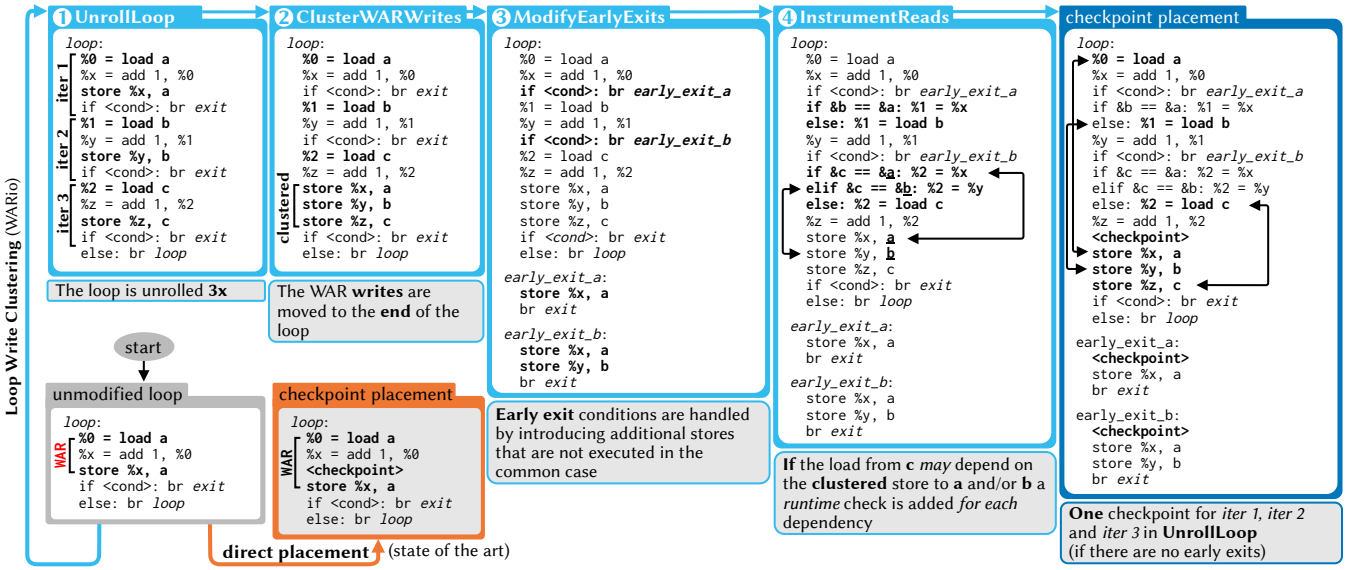


Figure 3. Code blocks of a simplified version of the IR of the loop, where variables starting with a '%' denote registers, '<cond>' is the condition that terminates the loop, 'br' branches to a label. The 'exit' label and IR (not important for the transformation) are omitted. The unmodified loop is directly instrumented with checkpoints (orange box) by Ratchet [57], i.e. the state of the art system. In this example WARio applies the Loop Write Clusterer transformation (light blue) to reduce the required checkpoints from one per iteration to one every three iterations, as shown in the last code block (dark blue).

Write Clusterer. The goal of the Write Clusterer, just as the Loop Write Clusterer, is to reduce the number of checkpoints inserted by clustering write operations belonging to WAR violations together. Doing so will cause the Checkpoint Inserter to resolve more WAR violations using a single checkpoint. Instead of the aggressively clustering used by the Loop Write Clusterer, the write cluster does not insert any runtime checks. The Write Clusterer analyses the individual basic blocks of the IR and looks for instances such as in Figure 1 (left), where multiple WAR violations are not dependent on each other. If this is the case, the Write Clusterer clusters the writes of the WAR violations as in Figure 1 (right). Doing so reduces the number of required checkpoints by handling multiple WAR violations with one checkpoint.

PDG Checkpoint Inserter. After transforming the IR during the previously described transformations, the next and final step is to insert checkpoints to break all the remaining WAR violations. The goal of a checkpoint is to save the current volatile state of the system in a way that it can continue operation after a power failure at that point. A checkpoint saves all the volatile-state of the system in NVM. For WARio, a checkpoint contains only the state of the registers, as the main memory is completely NV. Doing this is a multi-step process similar to that of [57]. For each function in the program, the transformation collects all the WAR dependencies. Next, the transformation collects all the locations of forced checkpoints, e.g., at function calls, and removes WAR violations resolved by these forced checkpoints. The remaining WAR violations are resolved by inserting checkpoints between the read and the write of a WAR violation. Where to place a checkpoint is a crucial decision, as a single checkpoint can resolve multiple WAR violations if placed correctly. The transformation converts each of the remaining WAR violations to a set of locations that resolve that WAR violation. Next, a cost is associated with all the potential checkpoint locations, primarily depending on the loop depth. The resulting sets of potential locations are used in a greedy minimal hitting set algorithm [11, Section 4.2.1] to find a set of checkpoint locations that resolve all the WAR violations. This technique was also used by Ratchet [57]. Both write postponing transformation discussed before are effective because they reschedule the write instructions so that the hitting set algorithm can resolve multiple WAR violations with a single checkpoint. Therefore, the hitting set algorithm would result in fewer overall checkpoint locations and is integral to the system's performance.

3.1.3 WARio Back End.

The final steps of the code transformation are performed by the back end. All steps (listed within the dark blue area in Figure 2) are explained below.

Hitting Set *Stack Spill* Checkpoint Inserter. Up to this point, WARio targeted memory dependencies in the middle end of the compiler. However, to safely support intermittent execution, all WARs to NVM must be handled with a checkpoint, including those that arise in the compiler's back end.

During the register allocation phase, the back end may run out of empty registers and move (spill) some of these registers to a stack slot on the stack. The accesses to the NV stack can introduce new WAR violations. These WAR violations are resolved by first forcing the compiler not to reuse any stack slot during the register allocations phase; after this, only a loop can cause a write after a read to one of these slots. Instead of placing a checkpoint before a write to a stack slot that causes a WAR, as is the case in Ratchet [57]. WARio's `Hitting Set Stack Spill Checkpoint Inserter` handles inserting checkpoints by applying the same algorithm as the middle end. A minimum hitting set algorithm [11, Section 4.2.1] selects the checkpoint locations, not using memory information provided by the PDG, as this information is not available during this compilation stage, but by using the known stack slot locations. Strategically placing the checkpoints to handle more WAR violations per checkpoint dramatically reduces the number of checkpoints introduced in the back end, caused by the register pressure increases following `Write Clusterer` and `Loop Write Clusterer` transformations. It is, therefore, a vital component of WARio that allows the checkpoint reduction achieved in the middle end to propagate through the back end.

Idempotent *Stack Pop* Converter. The remaining WAR violations caused in the back end are due to pop instructions. When executing a pop instruction, the stack variables are first loaded (read) into registers, and then the stack pointer is adjusted. Assuming an interrupt happens, the processor will automatically push some registers on the stack and jump to the interrupt service routine. The act of pushing (writing) data to the stack causes a WAR violation concerning the stack. Resolving these WAR violations is done the same way as in Ratchet [57], breaking all pop instructions into (i) first loading the memory into registers, then (ii) performing a checkpoint, and finally (iii) adjusting the stack pointer.

Epilog Optimizer. Because of the aforementioned checkpoints required to absolve all the pop instructions from WAR violations, the epilog of any functions contains at least one checkpoint whenever it uses stack memory. However, often the stack pointer is not adjusted in one go when a function returns. Factors such as the use of a frame pointer and other back end implementation-specific causes can induce more stack pointer adjustments, leading to an equal number of additional checkpoints. As a final transformation just before the code generation phase, WARio analyzes the epilogs of all the functions and will reduce the required number of checkpoints during the epilog to just one, whenever possible. It does so by temporarily postponing any incoming interrupts until after the stack adjustment, eliminating the chance of an interrupt allocating on the stack and therefore eliminating WAR violations. Doing this will result in a longer delay between the interrupt arrival and handling. However, the delay consists of only a small amount of instructions.[3] This epilog optimization results in only one inserted function epilog checkpoint before the last stack pointer adjustment to avoid interrupt-related WAR violations, **instead of up to three** in [57], reducing the penalty of function calls.

4 WARio Implementation

We now proceed with the implementation details of WARio's architecture presented in Section 3.

4.1 Target Architecture

We implemented WARio for the popular 32-bit ARM Cortex-M processor architecture [6], but with on-chip mixed (volatile and non-volatile) main memory, such as the recent Ambiq Apollo4 Blue [3]. WARio's main memory resides in the NVM, including all global- and stack-allocated variables. Only the processor configuration, e.g., peripheral configurations, and the registers, are volatile. Therefore, only the register's state is being stored during a checkpoint.[4]

4.2 Selected Compiler and PDG Analyzer

We chose LLVM version 9.0.1 [41] as the compiler on top of which WARio is built. For the PDG analysis and loop transformation abstractions WARio uses NOELLE [36] (commit `fc36051`).

4.3 WARio Middle End Transformations

We proceed with the description of all IR transformations performed by WARio.

Loop Write Clusterer.
Using abstractions provided by NOELLE this transformation iterates over all loops in the program. For each loop, it performs the algorithm described in 3.1.2—Paragraph `Loop Write Clusterer`. The unrolling factor N is a compile-time flag provided to WARio. The default unroll factor used to assess WARio performance is $N = 8$, which we found experimentally—refer to Section 5.2.4.

Expander. This transformation goes over all the functions in the input program twice. Firstly, it creates a list of functions containing pointers. These functions are candidates to be inlined, as they might aid in the later transformations. Secondly, the `Expander` goes through all the calls in every function. If a function call is in a loop without any sub-loops—and appears in the list of candidate functions—the `Expander` inlines the function call into the caller.

Write Clusterer. This transformation uses the WAR violation results from the PDG to collect potential WAR clustering candidates. The WAR writes (`store` instructions of

[3] As WARio targets intermittent computing, where the device might power off at any time, this delay in interrupt handling is not a concern.
[4] We emphasize that peripherals are not addressed in this work. We refer to Section 6 for further discussion.

LLVM) of these WAR violations are then clustered as described in Section 3.1.2—Paragraph Write Clusterer.

PDG Checkpoint Insterter. Finding the checkpoint locations happens as described in Section 3.1.2—Paragraph PDG Checkpoint Inserter. The transformation uses PDG information provided by NOELLE to find the WAR violations in the program. Next, the transformation inserts checkpoint intrinsics, i.e., special placeholder instructions that signify the back end to insert a checkpoint at that location, at all the checkpoint locations selected by the hitting set algorithm.

4.4 WARio Back End Transformations

We need to stress that inserting checkpoints to avoid WAR violations to physical NVM is a task 'close' to the actual hardware, which can only be handled by the back end. Not all WAR violations can be discovered and resolved in the middle end of the compiler. Therefore, the final step is to resolve all the WAR violations in the compiler's back end. The reason for not resolving all WARs directly in the back end is that information on, e.g., detailed memory dependency from the PDG, is accessible only in the middle end.

Hitting Set Stack Spill Checkpoint Inserter. The first cause of WAR violations in the back end is handled by the `Hitting Set Stack Spill Checkpoint Inserter`, which occurs after the register allocation. During the `LLVM` register allocation `-no-stack-slot-sharing` option is used to disallow the reuse of stack slots. The remaining stack spills can only cause a WAR violation *if* they occur in a loop, caused by re-executing a basic block that re-uses the stack slot. The transformation goes through all the stack slot accesses in the LLVM Machine IR and checks for non-handled WAR violations, i.e., violations not already handled by checkpoints inserted in the middle end. Instead of inserting checkpoints before the stack slot writes of remaining WARs, as in Ratchet [57, Section 4.1], WARio implements a minimum hitting set algorithm similar to what is used in the middle end (Section 4—Paragraph PDG Checkpoint Inserter) to reduce the required checkpoints needed to eliminate all WARs.

Idempotent *Stack Pop* Converter. The other cause of WAR violations in the back end occurs during the final frame lowering step as discussed in Section 3.1.3—Paragraph Idempotent Stack Pop Converter. WARio implements this step for the Thumb-2 [5] back end in LLVM, instead of the Thumb back end used in Ratchet (as we found out in its source code [22]), in order to support the Cortex-M [6].

Epilog Optimizer. The Thumb-2 back end in LLVM inserts up to three different stack pointer modifications during the epilog of a function to restore (i) callee saved registers, (ii) the frame pointer, and (iii) other allocated stack memory. To handle all these potential WAR violations with a single checkpoint instead of three, we exploit a trait of target Cortex-M architecture. Namely, (i) temporarily disabling the global interrupts before the stack-pointer adjustment, and (ii) and re-enabling them afterwards. During the period where the interrupts are disabled, which usually lasts a few instructions, interrupts are not lost but set as pending. After the interrupts are re-enabled, any pending interrupt will trigger.

4.5 Checkpoints

All the previously discussed transformations do not actually insert checkpoint calls directly. Instead, they insert checkpoint intrinsics happens just before the code generation in the compiler's back end. The checkpoints themselves are assembly routines. As the main memory is NV, the checkpoint only includes the current state of the (live) registers. However, one can not simply copy the content of the registers to a reserved location in NVM, as this would lead to a corrupt checkpoint if the power fails during the creation of a said checkpoint. Instead, in order to be incorruptible, the checkpoint has to be double buffered in NVM, as in other software support systems for intermittently-powered devices, e.g. [26, Section 3], [60, Section 3.4].

4.6 Compilation Process

Creating the intermittently-executable code is as simple as replacing LLVM's `clang` [39] with our dedicated WARio compilation script, denoted as `iclang`. `iclang` orchestrates the different compiler transformations without any user intervention. Within `iclang` the programmer can also specify a compilation path that can be selected from all possible ones shown in Figure 2. `iclang` compiles the C program without any transformations using `gllvm` version 1.3.0 [45]. This compilation stage creates the whole-program IR file from multiple C project files which is then used as an input to the WARio. Additionally, before the `Loop Write Clusterer`, a basic inlining transformation (using LLVM-specific `opt -always-inline -inline` command) is executed. Also, before the `Expander` transformation the user-specified optimization level (e.g., `-O2`, `-O3`) is applied. After all needed transformations the WARio generates the ELF program binary, which can be then executed on a intermittenlty-powered device.

5 WARio Evaluation

We now proceed with the evaluation of WARio vis-á-vis state-of-the-art compiler-based software system for intermittently-powered devices. WARio, together with all supporting code to gather and process the evaluation results is available via an open-source repository [38] and as an artifact [37].

5.1 Evaluation Setup

We begin with the outline. We will justify implementation choices aimed at the correct assessment of WARio.

5.1.1 Target Processor Platform. WARio performance was measured using a custom-built emulator for ARM Cortex-M processors with on-chip byte addressable NVM. During

WARio's development, the only such processor announced commercially was the Ambiq Apollo4 Blue [3], which was not yet available at the time of writing this article due to the ongoing chip shortage started in 2021 [9].

Why Processor Emulation is Needed. The reason for using emulation is threefold. First, an emulator enables us to collect detailed information about the processor status without inserting additional code for data collection (such as variable increments at a traced event). Such code inserts would alter the evaluation results on actual hardware. Simply, these new variables manipulations would introduce additional WAR dependencies to resolve, which were not part of the input benchmark code and should therefore not be counted. Second, emulation enables us to verify the absence of WAR violations during execution by checking all memory accesses in the emulator. Finally, it allows us to evaluate WARio without requiring the delayed Ambiq Apollo4 Blue. We emphasize that processor emulation is a common assessment strategy in many works targeting software systems for intermittently-powered devices. Examples are [57, Section 4.2], [21, Section 6], [34, Section 5.1], and [33, Section 6.1].

Emulator Architecture. The developed emulator is based on the Unicorn [44] CPU emulator version 1.0.3, which itself is based on the QEMU emulator [42]. Unicorn was selected for reasons of (i) native support of the ARM Cortex-M family [6], (ii) support of the Thumb-2 instruction set [5, Section 1.2.1] (which is needed for ARM Cortex-M) and (iii) ability to extend the emulator with new features. Specifically, the features we built on top of Unicorn are as follows.

► ***Performance Statistics Collection:*** The emulator enables to collect information on (i) the number of executed clock cycles, (ii) the number and cause of checkpoints, (iii) the number of clock cycles between two consecutive checkpoints, and (iv) where checkpoints occurred in the code. For the pipeline refill-based instructions of ARM Cortex-M4 [2, Section 3.3.1] we calculate the approximate number of executed clock cycles using our implementation of the three-stage instruction pipeline used by Cortex-M processors.

► ***WAR Violation Absence Verification:*** Our emulator performs the same verification of the absence of WAR violations as in [34, Section 5.2] with one main modification. The work of [34] checked only the middle end code, excluding the processor specific back end. Our WAR violation verification is build into the emulator, which allows us to detect WAR violations also in the back end and in any assembly code.

5.1.2 Software Benchmarks. The first software benchmark used in the evaluation is CoreMark [16], an industry-grade benchmark for measuring CPU performance in embedded systems. Additionally, we have used the following programs from the MiBench [19] suite: CRC, SHA, and Dijkstra. We have also used picojpeg [17] and Tiny AES [43] to represent two real-world libraries for embedded platforms.

As described in Section 4.6, all benchmarks use the same compilation pipeline: from plain C to complete WARio. When a certain transformation is disabled for a specific benchmark compilation (see Section 5.1.3), the IR passes through this specific transformation without any modifications. All benchmarks are compiled using the `-O3` optimization level of LLVM. Furthermore, the loop unroll factor in the `Loop Write Clusterer` transformation is $N = 8$, which we empirically found, as will be presented in Section 5.2.4.

5.1.3 Software Environments. We evaluate all benchmarks, listed in Section 5.1.2, in the following software environments. Justification for our selection of these environments is outlined in Section 7.

WARio and its Components. Benchmarks are evaluated by a WARio and by WARio with `Expander`. We also evaluate individual transformations of WARio, as listed in Figure 2, i.e. `Loop Write Clusterer`, `Expander`, `Write Clusterer` and `Epilog Optimizer`. Note that the `Checkpoint Inserter`, the basic version of the `Stack Spill Checkpoint Inserter`, and the `Idempotent Stack Pop Converter` transformations are always required to create a program that can execute intermittently and are included in all the other WARio transformations. In addition, the `Hitting Set Stack Spill Checkpoint Inserter` includes optimized checkpoint placement algorithm that uses a minimum hitting set to aid the write clustering transformations (Section 3.1.3—Paragraph `Stack Spill Checkpoint Inserter`). This advanced version is enabled during all WARio benchmarks, except for the `Epilog Optimization` (not to impact its results).

Ratchet. Ratchet [57] is the only completely compiler-based software environment for intermittently-powered devices, i.e. operating fully in the middle and back end of the compiler, without runtime memory logging as e.g. [26, 32], or source instrumentation, as e.g. [26, 31, 60]. Ratchet also addresses all features (❶–❺) listed at the beginning of Section 3. During the evaluation we use an unaltered version of the Ratchet middle end available via [22], and re-implemented the back end to support the Thumb-2 instruction set [5, Section 1.2.1] needed for ARM Cortex-M family [6], as we remarked already in Section 5.1.1.

R-PDG. Additionally, we designed and implemented a version of Ratchet [57], denoted as R-PDG, that uses the PDG information provided in NOELLE [36] for checkpoint insertion, instead of the built-in aliasing information available in LLVM. This adaptation to Ratchet is made to evaluate only the effect of WARio transformations while excluding the added benefit of using PDG information.

Non-instrumented Plain C Code. Finally, plain C (non-instrumented version) of all benchmarks are executed. They will be treated as the ultimate reference to all benchmarks run in all software environments listed above.

5.1.4 Energy Traces. We evaluate WARio considering the following power supply cases.

► ***Continuous Power:*** This case is required to measure execution time overhead from checkpoint insertions and code transformations for all software environments.

► ***Intermittent Power with Predefined Pattern:*** For a single scenario a fixed power on period is repeated until a given benchmark completes its execution.

► ***Intermittent Power with Measured Traces:*** We have run our emulator following two example empirical voltage traces measured at the output of an actual energy harvester of a battery-free embedded device. The preexisting traces used in our evaluation, available via [47], were initially used in the evaluation of Mementos [49]: one of the first software frameworks for battery-free intermittently-powered devices.

5.2 Evaluation Results

With the evaluation setup introduced, we are ready to present the evaluation results of WARio.

5.2.1 Execution Time. First, we measured execution time for all benchmarks (listed in Section 5.1.2) executed by all software environments (listed in Section 5.1.3). All results were normalized to the execution time of non-instrumented plain C code versions of each benchmark.[5] The results are presented in Figure 4.

The core message of this evaluation is that the average execution time for all benchmarks with WARio (blue dashed line in Figure 4) is reduced by 45.6% compared to average execution time for Ratchet (gray dotted line in Figure 4) and 27.7% compared to R-PDG (gray dashed line in Figure 4). Average per-benchmark overhead reduction by using WARio was also significant. WARio with `Expander` reduced the overhead of Ratchet and R-PDG by 58.1% and 44.3%, respectively. The above numbers demonstrate that WARio reduces checkpointing overhead on intermittently powered devices.

5.2.2 Checkpoint Cause. Figure 4 shows also how beneficial each compiler transformation is (see Section 5.1.3). We see that each benchmark benefits differently from each transformation. To shed more light into this observation we gathered more statistics. For the same setup as in Figure 4, we recorded the number of inserted checkpoints that were executed and what caused them. The result is presented in Figure 5. Specifically, we gathered how many checkpoints were caused by the (i) back end WAR dependency, (ii) middle end WAR dependency, (iii) function entry, and (iv) function exit. Ratchet is not present in Figure 5 because the number of checkpoints compared to other software environments listed in Section 5.1.3 is disproportionately high. In other words it is far worse than its improved version R-PDG. Therefore we have used R-PDG as a reference point for the evaluation. In Figure 5, R-PDG represents the starting point for each benchmark, i.e., it represents 100% of the checkpoints. Each WARio transformation aims to reduce the number of executed checkpoints relative to R-PDG, represented by the total height of each stacked bar.

Inspecting individual benchmarks, Dijkstra execution time is almost non-visible in Figure 4. This is because of few WAR violations occur in Dijkstra. This is shown by the data gathered for Dijkstra seen in Figure 5, where the numver of reduced checkpoints (except for function exit) at each WARio transformation is not decreasing. For CRC, on the other hand, there are no middle end checkpoints to optimize—this is the reason for smallest improvement from WARio with `Expander` compared to other benchmarks. Benchmarks that benefit most from WARio's write clustering are SHA and Tiny AES, because both benchmarks contain many loop operations. Specifically, for SHA and Tiny AES reduction of middle end WAR checkpoints after the `Loop Write Clusterer` is ≈60% and ≈70%, respectively.

Inspecting individual compiler transformation, the gain from the use of `Expander` is not significant, or is even slightly detrimental, as in the case for Tiny AES. The reason is as follows. `Expander` attempts to guess what functions are good to inline and sometimes this guess is inaccurate (see Section 3.1.2—`Expander`). To really benefit from `Expander`, WARio would need a code profiling information. The `Epilog Optimizer` reduces checkpoints for benchmarks with many exits; CRC benefits from this significantly.

The middle end is the main focus of WARio transformations. These, however, can lead to an increase in register spills due to the increased register pressure. However, as we observe, the reduction in the number of middle-end checkpoints heavily outweighs the increased number of checkpoints in the back end. This is seen in Figure 5 for CoreMark, SHA and Tiny AES, comparing the number of back end checkpoints with and without the transformations.

5.2.3 Code Size. Next, we measured the overhead in terms of extra `.text` size in the ELF of (i) Ratchet, (ii) WARio, and (iii) WARio with the `Expander` transformation compared to the non-instrumented (original C) versions. These measurements, presented in Table 2, show the code-size penalty associated with WARio's speedup demonstrated in Figure 4.

The average code-size increase of Ratchet and WARio are nearly identical. Per-benchmark overhead mainly depends on the number of checkpoints inserted in the code and they are rather consistent between Ratchet and WARio (except for AES—advantageous for WARio and for AES—advantageous for Ratchet). This suggests that not only WARio performs better than Ratchet, and attains this without any extra code footprint penalty. The code size is not significantly affected, even though WARio removes many checkpoints (as demonstrated in Figure 5) because a checkpoint is a simple jump

[5] Note, however, that C-only code is incapable of maintaining forward progress on intermittently-powered device with volatile/non-volatile memory architecture.

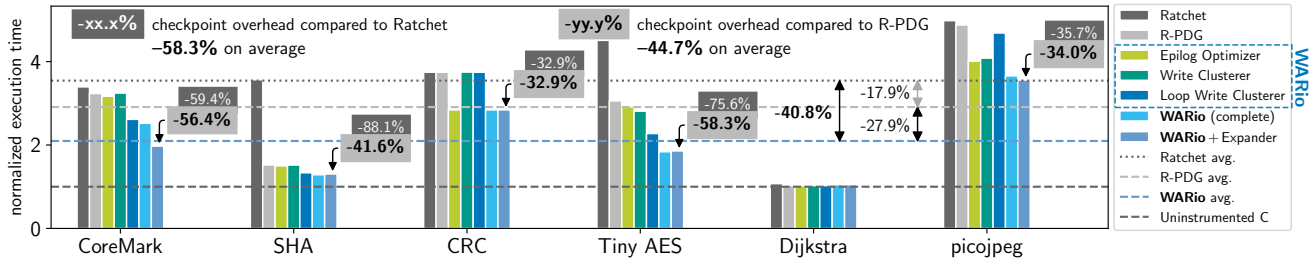


Figure 4. Execution time for all benchmarks for Ratchet [22], R-PDG (i.e. improved PDG-based version of Ratchet, see Section 5.1.3) and various components of WARio (per isolated WARio compiler transformation, complete WARio and WARio with `Expander` [see Section 3.1.2]). All results are normalized to the uninstrumented C version of each benchmark, i.e. the lower bound of execution time of each benchmark.

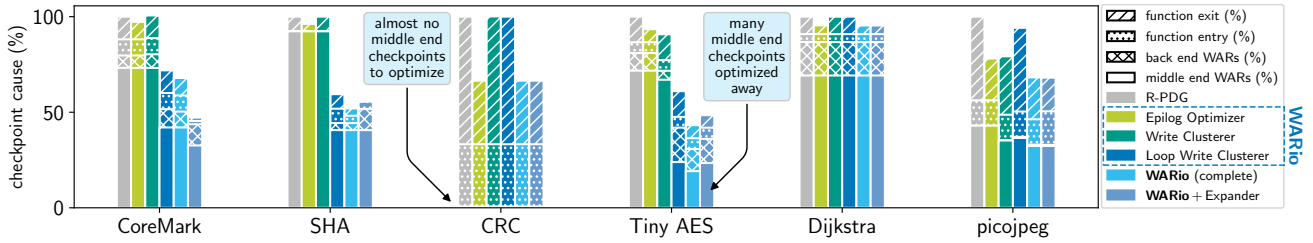


Figure 5. Analysis of the checkpoint cause for the corresponding benchmarks presented in Figure 4. Each stacked bar, per benchmark, represents the reduction of checkpoints compared to R-PDG, representing 100%. As the bars for Ratchet are excluded from the figure due to the scale difference, the reduction in the number of executed checkpoints compared to Ratchet is shown separately in Table 1. Each bar consists of four segments, indicated by the different hashing applied. The different segments depict the checkpoint causes: function exit, function entry, back end, or middle end.

instruction to the checkpoint routine. Hence, removing a checkpoint only removes a single instruction from the executable. Additionally, WARio sometimes adds additional instructions while executing the `write clustering` transformations. On the other hand, adding the `Expander` transformation to WARio does increase in the average code size. Note that `Expander` does not always translate to an increase in performance, as seen in Figure 4. The reason for `Expander` increases the code is because of inlining functions duplicates.

Next, we investigate how large the loop unroll factor N should be. The result are presented in Figure 6. For this experiment we chose a subset of benchmarks that benefited most from `Loop Write Clusterer`, i.e. SHA and Tiny AES, see again Figure 5. We measured the total number of checkpoints (top part of the figure) and execution time overhead reduction (bottom part of the figure) compared to benchmark with $N = 1$, i.e. no unrolling, as a function of N.

5.2.4 Loop Unroll Factor. The first observation is that as N reaches a certain point, the percentage of checkpoint reduction stalls. Simply, there need to be checkpoints for intermittent system to work correctly. However, unrolling a selected loop for loop write clustering. On average, steady state (for both number of checkpoints as well as overhead) is reached when the number of checkpoints in the middle end is reduced from ≈80% to ≈40%. These factors also cause the overhead to fluctuate when the unroll factor N becomes large, as these added checks and checkpoints in the back end will outweigh the reduction of checkpoints in the middle end. The ideal unroll factor for these specific benchmarks appears to be $\approx N = 8$. Therefore, $N = 8$ has been selected for all the other experiments, as we remarked already in Section 5.1.2.

Table 1. The difference in total number of executed checkpoints by WARio compared to Ratchet.

	WARio	WARio + Expander
CoreMark	-36.6%	-56.0%
SHA	-88.6%	-87.8%
CRC	-33.5%	-33.5%
Tiny AES	-74.5%	-71.5%
Dijkstra	-18.7%	-18.7%
picojpeg	-33.6%	-33.7%
average	-47.6%	-50.2%

Table 2. Per-benchmark code-size increase compared to the original C version (without intermittent computing support).

	Ratchet	WARio	WARio + Expander
CoreMark	+39.6%	+38.7%	+67.9%
SHA	+33.2%	+33.4%	+62.3%
CRC	+8.4%	+7.8%	+7.8%
Tiny AES	+16.2%	+12.1%	+37.7%
Dijkstra	+7.9%	+8.2%	+8.2%
picojpeg	+5.2%	+11.9%	+13.4%
average	+18.4%	+18.7%	+32.9%

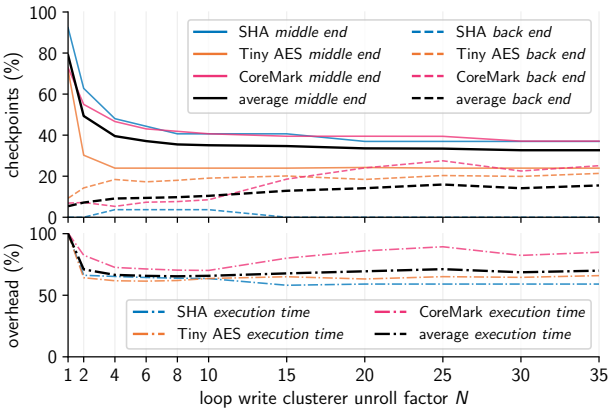


Figure 6. The effect of the loop write clustering transformation factor N on the overhead and number of checkpoints for three example benchmarks. $N = 2$ already gives a substantial improvement, while approximately $N = 8$ provides the most benefit.

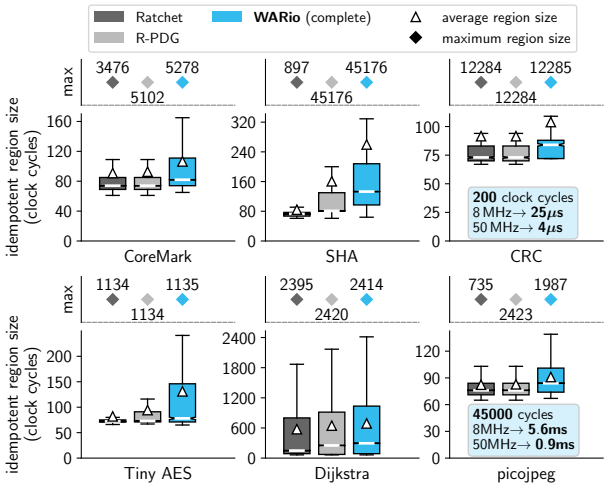


Figure 7. Idempotent region size for all considered benchmarks and software environments. Data is presented as a box plot, where maximum values are given at the top of each benchmark's result.

5.2.5 Impact of Power Intermittency.

We measured the size of the idempotent sections, i.e., the number of CPU clock cycles between two checkpoints during execution. Figure 7 shows the results for Ratchet, R-PDG, and WARio (complete). The median (white line) does not increase significantly. As expected, the 75th percentile (top of the box) and mean (white triangle) increase for most benchmarks. Most importantly, we see that (on average) the maximum idempotent region size is not significantly affected by the removal of over half of the checkpoints. In some cases, e.g., SHA, the maximum idempotent section size did increase dramatically. However, even with this increase, the required power on time is approximately 5.6 ms or 0.9 ms with a processor speed of 8 MHz or 50 MHz, respectively. WARio removes checkpoints at locations where idempotent sections are generally small, e.g., in a loop body or during the epilogue of a function, often leaving the large idempotent sections unmodified. Therefore, WARio does not significantly increase a device's required minimum power-on time to maintain forward progress as compared with Ratchet [57]. We note that additional research is needed to automatically reduce large regions to sustain forward progress for systems requiring even lower minimum power on time. However, the WARs remain protected, preventing inconsistencies due to power failures even in this case.

Furthermore, we executed the same benchmarks using different power on/power off patterns, as specified in Section 5.1.4, until completion. The overhead the intermittent execution introduces is composed of three factors: (i) the processor boot procedure execution, (ii) the last successful checkpoint restoration, and (iii) re-execution of the code between the last checkpoint and the location of the power failure. The first two factors are constant, but the third factor depends on where the power failure happened in the idempotent region. Table 3 shows this overhead as a percentage of the total execution time. For all the benchmarks, this overhead is minimal. Even with very short power on times of 2 ms (at a processor clock speed of 50 MHz), the average overhead is less than 1% compared to continuous power.

6 Discussion

Location-specific Checkpoints. WARio does not place checkpoints that are user- or application-specific, e.g. to guarantee that inter-checkpoint (idempotent region) time is not larger than certain number of cycles. On the other hand, the number of checkpoints placed by WARio is great enough that extra checkpoints might not be necessary, see Figure 7.

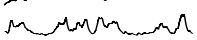
Sensing Applications and Use of Peripherals. WARio does not target sensing-based applications, that require interaction with the peripherals. This is a problem which needs to be solved separately, for example using special libraries [25, Section 3.4], which can be used in combination with WARio.

Code Profiling. WARio would benefit from a code profiler. Specifically, code profiling would improve both checkpoint placement and the effectiveness of the `Expander`. We leave the design of code profiling for the future.

Just In Time Checkpoints. Instead of inserting checkpoints to resolve WAR violations, the Just In Time strategy inserts them based on the developer-specified storage capacitor voltage threshold. This strategy brings some downsides. The incoming energy can be highly unpredictable [49, Figure 1], which means that the configured voltage level does not directly correlate to the amount of execution time left.[6] In

[6] The time between reaching the configured voltage level of the comparator, and when the system experiences a power failure, can highly fluctuate, even for a predictable energy harvesting source [25, Section 6.4].

Table 3. Code re-execution overhead in percentage for WARio with `Expander` compared to the continuously-powered version, $\mathcal{O}$, and number of observed power failures during benchmark execution, $\mathcal{P}$, per different power on cycles.

power *on* duration		CoreMark		SHA		CRC		Tiny AES		Dijkstra		picojpeg	
clock cycles	**time at {8 MHz, 50 MHz}**	$\mathcal{O}$	$\mathcal{P}$	$\mathcal{O}$	$\mathcal{P}$	$\mathcal{O}$	$\mathcal{P}$	$\mathcal{O}$	$\mathcal{P}$	$\mathcal{O}$	$\mathcal{P}$	$\mathcal{O}$	$\mathcal{P}$
50 000	{6.2 ms, 1 ms}	0.24%	127	2.87%	380	7.25%	1	0.23%	7	1.70%	1135	0.18%	2624
100 000	{12.5 ms, 2 ms}	0.14%	63	2.87%	190	0.00%	0	0.09%	3	0.86%	563	0.09%	1310
1 000 000	{125 ms, 20 ms}	0.01%	6	2.78%	19	0.00%	0	0.00%	0	0.07%	55	0.01%	130
5 000 000	{625 ms, 100 ms}	0.00%	1	0.00%	3	0.00%	0	0.00%	0	0.02%	11	0.00%	26
trace α		0.00%	3	0.04%	8	0.00%	0	0.00%	0	0.04%	27	0.00%	66
trace β		0.00%	1	0.00%	2	0.00%	0	0.00%	0	0.01%	5	0.00%	13

✤ Time (for a given processor frequency) is provided as a reference for two example processor clock speeds—8 MHz (i.e. speed at which internal FRAM of TI MSP430 [55] runs on a maximum speed) and 50 MHz.

Table 4. WARio compared against state-of-the-art intermittent execution support systems.

system	non-volatile main memory	register-only checkpoint	no runtime memory log	incorruptible	C language support	compiler-based	code-aware	code-transf.	ARM support
Mementos [49]	✗ no	✗ no	✓ yes	✓ yes	✓ yes	✗ no	✗ no	✗ no	✓ yes
MPatch [13]	✗ no	✗ no	✗ no	✓ yes	✓ yes	✗ no	✗ no	✗ no	✓ yes
Chinchilla [32]	✓ yes	✓ yes	✗ no	✓ yes	~ partially†	✓ yes	✗ no	~ partially	✗ no
TICS [26]	✓ yes	✗ no‡	✗ no	✓ yes	✓ yes	✓ yes♠	✗ no	✗ no	✗ no
InK [60]	~ partially	✓ yes	~ partially	✓ yes	✗ no✿	✗ no	✗ no	✗ no	✗ no
Rachet [57]	✓ yes	✓ yes	✓ yes	✓ yes	✓ yes	✓ yes	✓ yes	✗ no	✓ yes✤
WARio	✓ yes	✓ yes	✓ yes	✓ yes	✓ yes	✓ yes	✓ yes	✓ yes	✓ yes

† Does not support any form of recursion [26]. ‡ The active stack segment is included in the checkpoint. ♠ Source code instrumentation combined with a segmented stack implementation in the TI MSP430 [55] GCC [40] back end [26]. ✿ A C-style domain specific language for energy-task programming [26, Section 5.4]. ✤ Only Thumb instruction subset, no Thumb-2 support [22].

such a system, the configured voltage threshold must set to the worst case, as even one missed checkpoint can cause a WAR violation, corrupting the system's memory.

7 Related Work

The main (and only) system we can compare WARio to was Ratchet [57]. Nonetheless, this is not the only system available, as presented in Section 2. The most concrete comparison is given in Table 4. We also refer to [7, Table 1], [60, Table 1], [13, Table 2], for similar comparisons.

Loop Transformations. Early works considering the macro-level idea of instruction relocation and loop unrolling, however with specifics different from WARio, include [14] (in the context of an automatic coarsening the granularity of locks [by making one lock for multiple objects that can be accessed together] for the data manipulated by a program in a parallel computing system) and [28] (in the context of increasing instruction-level parallelism for processor instruction scheduling). Some volatile memory-based systems, e.g., [59], have introduced counters into loops to check when to create a checkpoint. Sadly, this does not work when the main memory is NV. Some form of loop-result buffering for task-based AI systems programmed using a special DSL was introduced in [18]. However, this approach does not work for general-purpose C-based applications.

Extensions of WARio. Other works can enhance WARio by tackling other optimizations. For instance, WARio can 'cache' some data in volatile memory if that data is both generated and used in one idempotent section, as in [33].

8 Conclusions

We have presented WARio: a set of compiler transformations that generate a binary which can be safely executed on intermittently-powered platforms. WARio injects checkpoints to resolve Write After Read violations but does it only after transforming the input code, moving 'Write' operations from individual WAR operations closer together, effectively reducing the number of required checkpoints—in turn, reducing checkpoint overhead.

Acknowledgments

We thank Dr. Saad Ahmed, our anonymous reviewers, and our shepherd Prof. James Larus 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, and by the National Science Foundation trough grants CCF 1908488, CNS 1763743, CNS 1850496 and CNS 2145584. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] Mikhail Afanasov, Naveed Anwar Bhatti, Dennis Campagna, Giacomo Caslini, Fabio Massimo Centonze, Koustabh Dolui, Andrea Maioli, Erica Barone, Muhammad Hamad Alizai, Junaid Haroon Siddiqui, and Luca Mottola. 2020. Battery-Less Zero-Maintenance Embedded Sensing at the MithræUm of Circus Maximus. In *Proc. SenSys* (Nov. 16–19). ACM, Virtual Event, 368–381. https://doi.org/10.1145/3384419.3430722.

[2] Ambiq Micro Inc. 2010. Cortex-M4 Revision r0p0 Technical Reference Manual. https://documentation-service.arm.com/static/5f19da2a20b7cf4bc524d99a. Last accessed: Nov. 10, 2021.

[3] Ambiq Micro Inc. 2021. Apollo4 Blue Ultra-Low Power Microcontroller. https://ambiq.com/apollo4-blue/. Last accessed: Nov. 10, 2021.

[4] Sotiris Apostolakis, Ziyang Xu, Zujun Tan, Greg Chan, Simone Campanoni, and David I. August. 2020. SCAF: A Speculation-Aware Collaborative Dependence Analysis Framework. In *Proc. PLDI* (June 15–19). ACM, London, UK, 638–654. https://doi.org/10.1145/3385412.3386028.

[5] Arm Limited. 2007. https://documentation-service.arm.com/static/5e8e116188295d1e18d34a29. Last accessed: Nov. 5, 2021.

[6] Arm Limited. 2021. Arm Cortex-M Series Processors. https://developer.arm.com/ip-products/processors/cortex-m. Last accessed: Nov. 4, 2021.

[7] 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.

[8] 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 Trans. Comput.-Aided Design Integr. Circuits Syst.* 35, 12 (2016), 1968–1980. https://doi.org/10.1109/TCAD.2016.2547919.

[9] Chris Baraniuk. 2021. Why is There a Chip Shortage? https://www.bbc.com/news/business-58230388. Last accessed: Mar. 18, 2022.

[10] Jongouk Choi, Qingrui Liu, and Changhee Jung. 2019. CoSpec: Compiler Directed Speculative Intermittent Computation. In *Proc. MICRO* (Oct. 12–16). ACM/IEEE, Columbus, OH, USA, 399–412. https://doi.org/10.1145/3352460.3358279.

[11] Marc de Kruijf, Karthikeyan Sankaralingam, and Somesh Jha. 2012. Static Snalysis and Compiler Design for Idempotent Processing. In *in Proc. PLDI* (June 11–16). ACM, Beijing, China, 475–486. https://doi.org/10.1145/2254064.2254120.

[12] 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.

[13] 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.

[14] Pedro C. Diniz and Martin C. Rinard. 1998. Lock Coarsening: Eliminating Lock Overhead in Automatically Parallelized Object-Based Programs. *J. Parallel Distrib. Comput.* 49, 2 (March 1998), 218–244. https://doi.org/10.1006/jpdc.1998.1441.

[15] Kristina Edström (Executive Publisher). 2020. Horizon 2020 EU Program Battery 2030+: Inventing the Sustainable Batteries of the Future: Research Needs and Future Actions. https://battery2030.eu/digitalAssets/861/c_861350-l_1-k_roadmap-27-march.pdf. Last accessed: Jun. 29, 2021.

[16] Embedded Microprocessor Benchmark Consortium. 2018. CoreMark Benchmark. https://github.com/eembc/coremark/releases/tag/v1.01. Last accessed: Jun. 29, 2021.

[17] Rich Geldreich. 2020. picojpeg: Plain C JPEG Decompressor. https://github.com/richgel999/picojpeg. Last accessed: Nov. 5, 2021.

[18] Graham Gobieski, Brandon Lucia, and Nathan Beckmann. 2019. Intelligence Beyond the Edge: Inference on Intermittent Embedded Systems. In *Proc. ASPLOS*. ACM, Providence, RI, USA, 199–213. https://doi.org/10.1145/3297858.3304011.

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

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

[21] 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.

[22] Matthew Hicks. 2017. Ratchet (Source Code from OSDI 2016). https://github.com/impedimentToProgress/Ratchet. Last accessed: Nov. 5, 2021.

[23] Hrishikesh Jayakumar, Arnab Raha, Woo Suk Lee, and Vijay Raghunathan. 2015. Quickrecall: A HW/SW Approach for Computing Across Power Cycles in Transiently Powered Computers. *J. Emerg. Technol. Comput. Syst.* 12, 1 (July 2015), 8:1–8:19. https://doi.org/10.1145/2700249.

[24] Mohamad Katanbaf, Anthony Weinand, and Vamsi Talla. 2021. Simplifying Backscatter Deployment: Full-Duplex LoRa Backscatter. In *Proc. NSDI* (April 12–14). USENIX, Virtual Event, 955–972. https://www.usenix.org/system/files/nsdi21spring-katanbaf.pdf.

[25] 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.

[26] 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.

[27] Tianxing Li and Xia Zhou. 2018. Battery-Free Eye Tracker on Glasses. In *Proc. MobiCom* (October 29 — November 2). ACM, New Delhi, India, 67–82. https://doi.org/10.1145/3241539.3241578.

[28] Jack L. Lo and Susan J. Eggers. 1995. Improving Balanced Scheduling with Compiler Optimizations that Increase Instruction-level Parallelism. In *Proc. PLDI* (June 18–21). ACM, La Jolla, CA, USA, 151–162. https://doi.org/10.1145/207110.207132.

[29] Brandon Lucia, Vignesh Balaji, Alexei Colin, Kiwan Maeng, and Emily Ruppel. 2017. Intermittent Computing: Challenges and Opportunities. In *Proc. SNAPL* (May 7–10). Alisomar, CA, USA, 8:1–8:14. https://drops.dagstuhl.de/opus/volltexte/2017/7131/pdf/LIPIcs-SNAPL-2017-8.pdf.

[30] 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. *Micro* 36, 3 (May–Jun. 2016), 72–83. https://doi.org/10.1109/MM.2016.35.

[31] 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.

[32] 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.

[33] Andrea Maioli and Luca Mottola. 2021. ALFRED: Virtual Memory for Intermittent Computing. In *Proc. SenSys* (Nov. 15–17). ACM, Coimbra, Portugal. https://arxiv.org/pdf/2110.07542.pdf.

[34] Andrea Maioli, Luca Mottola, Muhammad Hamad Alizai, and Junaid Haroon Siddiqui. 2021. Discovering the Hidden Anomalies of Intermittent Computing. https://www.ewsn.org/file-repository/ewsn2021/Article1.pdf. In *Proc. EWSN* (Feb. 17–19). Delft, The Netherlands.

[35] Amjad Yousef Majid, Carlo Delle Donne, Kiwan Maeng, Alexei Colin, Kasım Sinan Yıldırım, Brandon Lucia, and Przemysław Pawełczak. 2020. Dynamic Task-based Intermittent Execution for Energy-harvesting Devices. *ACM Trans. Sens. Netw.* 16, 1 (Feb. 2020), 5:1–5:24. https://doi.org/10.1145/3360285.

[36] Angelo Matni, Enrico Armenio Deiana, Yian Su, Lukas Gross, Souradip Ghosh, Sotiris Apostolakis, Ziyang Xu, Zujun Tan, Ishita Chaturvedi, Brian Homerding, Tommy McMichen, David I. August, and Simone Campanoni. 2022. NOELLE Offers Empowering LLVM Extensions. In *Proc. CGO* (April 2–6). ACM, Seoul, South Korea.

[37] Delft University of Technology Sustainable Systems Lab. 2022. WARio Artifact. https://doi.org/10.5281/zenodo.6413018. Last accessed: Apr. 07, 2022.

[38] Delft University of Technology Sustainable Systems Lab. 2022. WARio Source Code Repository. https://www.github.com/tudssl/wario. Last accessed: Mar. 18, 2022.

[39] Open Source Community Contributors. 2021. C Language Family Front-end. https://github.com/llvm/llvm-project/tree/main/clang. Last accessed: Nov. 10, 2021.

[40] Open Source Community Contributors. 2021. GNU Compiler Collection. https://gcc.gnu.org/git.html. Last accessed: Jul. 21, 2021.

[41] Open Source Community Contributors. 2021. The LLVM Compiler Infrastructure. https://github.com/llvm/llvm-project. Last accessed: Jun. 28, 2021.

[42] 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: Nov. 5, 2021.

[43] Open Source Community Contributors. 2021. Tiny AES in C. https://github.com/kokke/tiny-aes-c. Last accessed: Nov. 5, 2021.

[44] Open Source Community Contributors. 2021. Unicorn: a Lightweight, Multi-platform, Multi-architecture Central Processing Unit (CPU) Emulator Framework based on QEMU. https://github.com/unicorn-engine/unicorn. Last accessed: Nov. 5, 2021.

[45] Open Source Community Contributors. 2021. Whole Program LLVM in Go. https://github.com/SRI-CSL/gllvm. Last accessed: Nov. 10, 2021.

[46] Aaron N. Parks, Angli Liu, Shyamnath Gollakota, and Joshua R. Smith. 2014. Turbocharging Ambient Backscatter Communication. In *Proc. SIGCOMM*. ACM, Chicago, IL, USA, 619–630. https://doi.org/10.1145/2740070.2626312.

[47] 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: Nov. 1, 2021.

[48] Benjamin Ransford and Brandon Lucia. 2014. Nonvolatile Memory is a Broken Time Machine. In *Proc. Memory Systems Performance and Correctness Workshop* (June 14). ACM, Edinburgh, United Kingdom, 1–3. https://doi.org/10.1145/2618128.2618136.

[49] 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.

[50] Emily Ruppel and Brandon Lucia. 2019. Transactional Concurrency Control for Intermittent, Energy-Harvesting Computing Systems. In *Proc. PLDI*. ACM, Phoenix, AZ, USA, 1085–1100. https://doi.org/10.1145/3314221.3314583.

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

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

[53] Yulei Sui and Jingling Xue. 2016. SVF: Interprocedural Static Value-Flow Analysis in LLVM. In *Proc. CC* (March 17–18). ACM, Barcelona, Spain, 265–266. https://doi.org/10.1145/2892208.2892235.

[54] 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.

[55] Texas Instruments Inc. 2017. MSP430FR59xx Mixed-Signal Microcontrollers (Rev. F). http://www.ti.com/lit/ds/symlink/msp430fr5969.pdf. Last accessed: Nov. 4, 2021.

[56] University of Washington, Seattle, WA, USA. 2010. Wireless Identification and Sensing Platform GitHub Page. https://github.com/wisp. Last accessed: Nov. 1, 2021.

[57] 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.

[58] Wiliot. 2021. Wiliot Battery Free IoT Pixel BLE Tags Website. https://www.wiliot.com/product/iot-pixel. Last accessed: Jul. 22, 2021.

[59] Bahram Yarahmadi and Erven Rohou. 2021. So Far So Good: Self-Adaptive Dynamic Checkpointing for Intermittent Computation based on Self-Modifying Code. In *Proc. International Workshop on Software and Compilers for Embedded Systems* (Nov. 24). Eindhoven, The Netherlands, 1–7. https://hal.inria.fr/hal-03410647/document.

[60] 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.

[61] G. Pascal Zachary. 2016. The Search for a Better Battery. https://spectrum.ieee.org/at-work/innovation/the-search-for-a-better-battery. Last accessed: Jun. 29, 2021.