ALFRED: Virtual Memory for Intermittent Computing

Andrea Maioli
Politecnico di Milano, Italy

Luca Mottola
Politecnico di Milano, Italy and RI.SE Computer Science and Uppsala University, Sweden

ABSTRACT

We present ALFRED: a virtual memory abstraction that resolves the dichotomy between volatile and non-volatile memory in intermittent computing. Mixed-volatile microcontrollers allow programmers to allocate part of the application state onto non-volatile memory. Programmers are therefore to manually explore the trade-off between simpler management of persistent state against energy overhead and possibility of intermittence anomalies due to non-volatile memory operations. This approach is laborious and yields sub-optimal performance. We take a different stand with ALFRED: we provide programmers with a virtual memory abstraction detached from the specific volatile nature of memory and automatically determine an efficient mapping from virtual to volatile or non-volatile memory. Unlike existing works, ALFRED does not require programmers to learn a new language syntax and the mapping is entirely resolved at compile-time, reducing the run-time energy overhead. We implement ALFRED through a series of machine-level code transformations. Compared to existing systems, we demonstrate that ALFRED reduces energy consumption by up to *two orders of magnitude* given a fixed workload. This enables workloads to finish sooner, as the use of available energy shifts from ensuring forward progress to useful application processing.

CCS CONCEPTS

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

KEYWORDS

Intermittent computing, virtual memory abstraction.

ACM Reference Format:
Andrea Maioli and Luca Mottola. 2021. ALFRED: Virtual Memory for Intermittent Computing. In *The 19th ACM Conference on Embedded Networked Sensor Systems (SenSys'21), November 15–17, 2021, Coimbra, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3485730.3485949

1 INTRODUCTION

Ambient energy harvesting [10] enables deployments of batteryless sensing devices [1, 20, 23, 27, 48, 50], reducing environment impact and maintenance costs. However, harvested energy is erratic and may not suffice to power devices continuously. Frequent power failures occur that cause executions to become *intermittent*, with periods of active operation interleaved by periods where a device is off recharging energy buffers. Power failures cause devices to lose the program state, restarting all over again when energy is newly available. Forward progress of programs is therefore compromised.

Problem. Several systems exist to ensure forward progress, as we discuss in Sec. 2. Common to these solutions is the insertion of state-saving operations within the execution flow. These operations offer the opportunity to create a replica of the program state, including main memory, register files, and program counter, onto non-volatile memory. The program state is eventually restored from non-volatile memory when energy returns, ensuring forward progress across power failures. The placement of state-saving operations in the program may be either decided in a (semi-)automatic fashion [7, 8, 11, 29, 36, 46, 49] or driven by programmers with custom programming abstractions [16, 34, 35, 44, 47, 52].

Mixed-volatile microcontrollers also exist, which offer the ability to store slices of the program state directly onto non-volatile memory. This is achieved using specific pragma statements [28], as

```
#pragma PERSISTENT(x)
unsigned int x = 5;
```

Program state allocated on non-volatile memory is automatically retained across power failures and may be excluded from state-saving operations, simplifying the management of persistent state.

Using mixed-volatile microcontrollers comes at the cost of increased energy consumption: non-volatile memory operations may require up to 247% the energy of their volatile counterpart [28, 40]. Storing only parts of the program state on non-volatile memory may also yield intermittence anomalies [43, 45], due to re-executions of non-idempotent code, which require further energy to be corrected. Using mixed-volatile platforms, quantifying the advantages in simpler management of persistent state against the corresponding energy overhead is complex, as these depend on multiple factors including energy patterns and execution flow.

ALFRED. We take a different stand. Rather than requiring programmers to manually determine when to use non-volatile memory for what slice of the program state, we promote a higher-level of abstraction through a concept of *virtual memory*. Programmers write intermittent code without explicitly mapping variables to volatile or non-volatile memory. Given a placement of state-saving operations in the code, we *automatically* decide *what* slice of the program state must be allocated onto non-volatile memory, and *at what point* in the execution. Programmers are therefore relieved from deciding the mapping between program state and memory. Moreover, the mapping is not fixed at variable level, but is *automatically adjusted* at different places in the code for the same data item, based on read/write patterns and program structure.

SenSys'21, November 15–17, 2021, Coimbra, Portugal

ACM ISBN 978-1-4503-9097-2/21/11…$15.00
https://doi.org/10.1145/3485730.3485949

ALFRED[1] is our implementation of virtual memory for intermittent computing, based on two key features:

(1) it is *transparent to programmers*: no dedicated syntax is to be learned, and programmers write code in the familiar sequential manner without the need to tag variables.
(2) the mapping from virtual to volatile or non-volatile memory is entirely *resolved at compile-time*, reducing the energy overhead that represents the cost of using the abstraction.

The virtual memory abstraction we conceive does not provide virtualization in the same sense as mainstream OSes. Instead, it offers an abstraction that shields programmers from the need to statically determine a specific memory allocation schema. ALFRED is therefore sharply different compared to mainstream virtual memory systems [5, 19]. These usually provide an idealized abstraction of storage resources, so that software processes operate as if they had access to a contiguous memory area, possibly even larger than the one physically available. Address translation hardware maps virtual addresses to physical addresses at run-time. In ALFRED, we target resource-constrained energy-harvesting devices that compute intermittently [23]. The abstraction we offer provides programmers with a higher-level view on the persistency properties of different memory areas, and automatically determines the mapping from the virtual memory to the volatile or non-volatile one. Because of resource constraints, we determine this mapping at compile-time.

ALFRED determines this mapping using three key program transformation techniques, illustrated in Sec. 3. Their ultimate goal is simple, yet challenging to achieve, especially at compile-time:

Use the energy-efficient volatile memory as much as possible, while enabling forward progress using non-volatile memory with reduced energy consumption compared to existing solutions.

This entails that we need to promote the use of volatile memory whenever convenient, for example, to compute intermediate results or to store temporary data that need not survive a power failure, while allocating the data that does require to be persistent onto non-volatile memory in anticipation of a possible power failure. By doing so, we decrease energy consumption by taking the best of both worlds: we benefit from the lower energy consumption of volatile memory *whenever possible*, and rely on the persistency features of non-volatile memory *whenever required*.

Applying program transformations at compile-time is, however, challenging because of the lack of run-time information. Sec. 4 illustrates how we address the uncertainty that arises, using a set of dedicated program normalization passes. The result of the transformations require a specific memory layout to operate correctly and a solution to the possible intermittence anomalies. We describe in Sec. 5 how we deal with these issues, using an approach that is co-designed with our program transformation techniques.

We build an implementation of ALFRED based on ScEpTIC [38, 43], an extensible open-source emulation environment for intermittent programs. Given fixed workloads and staple benchmarks in the field [7, 8, 26, 29, 43, 46, 49], we measure ALFRED performance in energy consumption, number of clock cycles, memory accesses, and restore operations. We compare ALFRED with multiple baselines obtained by abstracting the key design dimensions of existing systems in a framework that allows us to instantiate baselines matching existing systems, while also exploring alternative configurations. Depending on the benchmark, ALFRED can provide *several-fold improvements* in energy consumption, which allow the system to shift the energy budget to useful application processing. This correspondingly allows the system to achieve comparable improvements in the time to complete the workloads.

[1] **A**utomatic a**L**location o**F** non-volatile memo**R**y for transi**E**ntly-powered **D**evices.

2 RELATED WORK

Ensuring forward progress is arguably the focus of most existing works in intermittent computing [23]. Common to these is the use of some form of persistent state on non-volatile memory.

A significant fraction of existing solutions employ a form checkpointing to cross power failures [3, 7, 11, 36, 46]. This consists in replicating the content of main memory, special registers, and program counter onto non-volatile memory at specific points in the code. Whenever the device resumes with new energy, state is retrieved back from non-volatile memory and computations restart. Systems such as Hibernus [7, 8] operate in a reactive manner: an interrupt is fired from a hardware device that prompts the application to take a checkpoint, for example, whenever the energy buffer falls below a threshold. Differently, systems exist that place explicit function calls to perform checkpoints [11, 36, 46, 49]. The specific placement is a function of program structure and energy patterns.

Other approaches offer abstractions that programmers use to define and manage persistent state [16, 34, 35, 52] and time profiles [24]. For example, DINO [34] allows programmers to split the sequential execution in individual tasks and ensures transactional semantics between consecutive task boundaries. Alpaca [35] goes a step further and provides dedicated abstractions to defines tasks as individual execution units that run with transactional semantics against power failures and subsequent reboots.

Using mixed-volatile platforms, intermittence anomalies potentially occur due to repeated executions of non-idempotent code [43, 45]. These are unexpected program behaviors that make executions differ from their continuous counterparts. Systems are available that address these issues with dedicated checkpoint placement strategies [49] or custom programming abstractions [16, 34, 35, 52], and to test their occurrence [38, 43]. Approaches are available that conversely take advantage of them to realize intermittence-aware control flows, promoting the occurrence of power failures to an additional program input [40]. Additional issues in intermittent computing include performing general testing of intermittent programs [15, 17, 21, 22], profiling their energy consumption [2, 15, 21], and handling peripheral states across power failures [6, 9, 12, 37].

Our work offers a different standpoint. Unlike the works above, we take the decision about what part of the application state to allocate on non-volatile memory away from programmers, and offer a uniform abstraction that does not entail any specific memory configuration. A set of program transformation techniques automatically determines an energy-efficient allocation at compile time, as a function of program structure and read/write patterns. Most importantly, such an allocation is not fixed once and for all at variable-level as in current practice, but is possibly adjusted at different places in the code for the same data item.

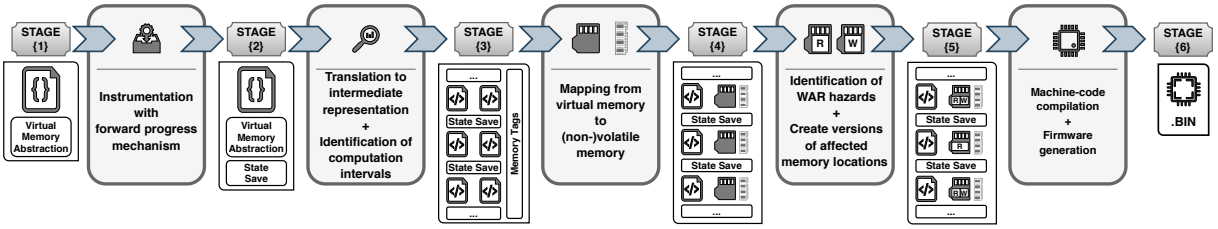


Figure 1: ALFRED compile-time pipeline.

Closest to our work are TICS [31] and the system of Jayakumar et al. [30]. TICS [31] limits the size of persistent state by solely saving the active stack frame and modified memory locations outside of it, which is conceptually similar to our approach. However, TICS primarily helps programmers deal with time across power failures, whereas we specifically target energy efficiency. TICS also exclusively uses non-volatile memory for global data and undo-logging [36] to avoid intermittence anomalies [43, 45]. In contrast, we opportunistically allocate slices of program state onto the energy-efficient volatile memory and employ program transformation techniques that ensure memory idempotency [49].

The system of Jayakumar et al. [30] adjusts the mapping of global variables, program code, and stack frames between volatile and non-volatile memory, doing so at the granularity of individual functions. They rely on hardware interrupts to trigger state-saving operations at runtime and tentatively allocate everything to non-volatile memory first, then incrementally move data or code to volatile memory until forward progress is compromised. At that point, they backtrack to the latest functioning configuration. Besides working at the granularity of single data items and at compile-time, rather than at run-time, our design is fundamentally different, as memory allocations are thought to *systematically* improve energy consumption. Therefore, if forward progress is possible before applying ALFRED, it remains so afterwards. ALFRED is thus never detrimental to the application's ability to do useful work.

3 VIRTUAL MEMORY MAPPING

The program transformation techniques of ALFRED determine the mapping from virtual to volatile or non-volatile memory. They are independent of the target architecture, as they are applied on an architecture-independent intermediate representation of the input program commonly used in compilers [33]. We illustrate the compile-time pipeline in Sec. 3.1, followed by an explanation of the single techniques in Sec. 3.2 to Sec. 3.4.

3.1 Overview

Fig. 1 shows the compile-time pipeline of ALFRED. The input at stage ⟨1⟩ is a program written using the virtual memory abstraction; therefore, variables in the program are not explicitly mapped to either volatile or non-volatile memory.

The program is first processed through the compile-time support an existing checkpoint system [7, 8, 11, 29, 36, 46, 49] or task-based programming abstraction [16, 34, 35, 44, 47, 52]. Either way, at stage ⟨2⟩ the program includes state-save operations inlined in the execution flow as calls to a checkpointing subsystem or placed at task boundaries. These operations are meant to dump program state onto non-volatile memory prior to a power failure and to restore the program state from non-volatile memory when energy is newly available. The techniques we explain next are orthogonal to how state-save operations are placed in the code.

Unlike existing programming systems for intermittent computing, our techniques work at the level of machine-code. At this level, memory operations are visible as they are actually executed on the target platform. At stage ⟨3⟩ in Fig. 1 we translate the program into an intermediate representation of the source code and initially map every memory operation *to volatile memory*. If we were to execute the code this way, state-save operations would need to dump the entire main memory to the non-volatile one when executing.

At the same stage we also partition the code into logical units we call *computation intervals*. A computation interval consists in a sequence of machine-code instructions executed between two state-save operations. For programs using checkpoint mechanisms [7, 8, 11, 29, 36, 46, 49], computation intervals correspond to sequences of instructions between two checkpoint calls. For programs using task-based programming abstractions [16, 34, 35, 44, 47, 52], computation intervals essentially correspond to tasks.

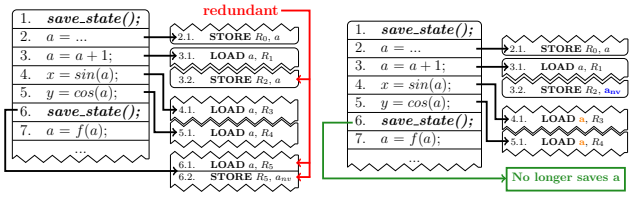
From now on, the three program transformations we illustrate next are applied in the order we present them. We focus on the intuition and general application of each transformation and postpone the discussion about dealing with compile-time uncertainty to Sec. 4. Our techniques operate on every memory target in the program, including not just memory targets that the compiler uses to map variables in source code, but also memory locations used by operations that are normally transparent to programmers, such as `PUSH` or `POP`. We detail how we identify the memory addresses of data items possibly involved in a transformation in Sec. 4 and how to compute their addresses after the transformations in Sec. 5.

As we hinted earlier, the mapping we want to achieve is one where volatile memory is used as much as possible for data that requires no persistency, for example, intermediate results or temporary data, as it is more energy efficient than its non-volatile counterpart. However, we want to make sure to use the latter, paying an energy overhead, whenever persisting data to survive power failures is necessary. Intuitively, the transformations generate a mapping from virtual to volatile or non-volatile memory where the former acts as a volatile cache of sorts.

The snippets we show next include both source and machine code for clarity. Line numbers refer to source code.

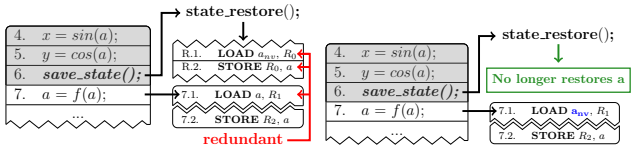
3.2 Mapping Write Operations

The first transformation we apply is based on a key intuition: *a memory write operation should target non-volatile memory as soon*



(a) Before the transformation. (b) After the transformation.

Figure 2: Example of mapping write operations.



(a) Before the transformation. (b) After the transformation.

Figure 3: Example of mapping read operations.

as the written data is final compared to the next state-save operation, so it relieves the latter from the corresponding processing.

The notion of *final* describes situations where the program no longer alters the data before the next state-save operation. Our intuition essentially corresponds to *anticipating* the actions that the state-save operation would perform anyways. This allows these operations to spare the overhead for saving data that can be considered final earlier: after the transformation the data is already on non-volatile memory when the state-save operation executes.

Example. Consider the program of Fig. 2(a) and let us focus on the computation interval extending up to line 6. We find two **STORE** instructions that target the volatile memory location that variable a is initially mapped to. Note that the second **STORE** instruction writes the same value that the state-save operation of line 6 stores for variable a, because the latter is initially allocated onto volatile memory and must be preserved across power failures. This is the case because the data for variable a is *final* already at line 3.

To save the overhead of redundant memory operations, we make the **STORE** instruction of line 3 immediately target non-volatile memory. This transformation allows us to remove the instructions that are necessary to save variable a at the state-save operation of line 6, along with the corresponding energy overhead, as line 3 already saves the content variable a onto non-volatile memory.

Fig. 2(b) shows the resulting program, which has reduced energy overhead because the state-save operation is no longer concerned with variable a that is made persistent already at line 3. Conceptually, this corresponds to moving the **STORE** instruction that would normally be part of the state-save operation to the last point in the program where variable a is actually written.

This transformation does not alter the target of the **STORE** instruction of line 2, where the data is not final yet. Doing so would incur an unnecessary energy overhead due to a write operation on non-volatile memory for non-final data, which is going to be overwritten soon after. In fact, the **STORE** instruction of line 2 produces an intermediate result for variable a, which we need not persist.

Generalization. We apply this technique to an arbitrary computation interval as follows. For each memory location x, we consider the possibly empty set of memory write instructions $I_w = (I_{w1}, ..., I_{wn})$ that manipulate x and are included in the computation interval; I_{wn} is the last such instruction and there is no other memory write instruction before the next state-save operation.

We relocate the target of I_{wn} to non-volatile memory, as whatever data I_{wn} stores is final. The targets of all other write instructions $I_{w1}, ..., I_{w(n-1)}$ remains on volatile memory, as they produce intermediate results that I_{wn} eventually overwrites. Note that this transformation is sufficient to preserve the value of the memory location x across power failures, while reducing the number of instructions targeting non-volatile memory.

By applying this transformation to all computation intervals and all memory locations, state-saving operations at stage ⟨4⟩ in Fig. 1 are left with *only* register file and special registers to handle, and accordingly modified. If a memory location is altered in a computation interval, our technique identifies when such a change is final and persists the data there. Otherwise, if $I = \emptyset$ there is no need to persist the data, as some previous state-save operation already did that the last time the data changed.

This processing not only reduces the operations on non-volatile memory, but also reduces the overhead of state-saving operations. A regular checkpoint mechanism would save the entire content of volatile memory onto the non-volatile one [7, 8, 11, 46], including unmodified memory locations. In our case, memory locations not modified in a computation interval are excluded from processing. We thus achieve differential checkpointing [3] with *zero* run-time overhead in both energy and memory consumption.

Next, consider the read instructions possibly included in the computation interval between I_{wn} and the state-save operation. As the data is now on non-volatile memory, in principle, they should also be redirected to non-volatile memory. Whether this is the most efficient choice, however, is not as simple. The third transformation, described in Sec. 3.4, addresses the related trade-offs.

3.3 Mapping Read Operations

The second transformation is based on the dual intuition: *when resuming, restore routines may be limited to register file and special registers, while memory read operations from non-volatile memory should be postponed to whenever the data is needed, if at all.*

This transformation effectively corresponds to *postponing* the restore operation to when the data is actually used and a read operation would execute anyways. By doing so, we spare the instructions in the restore routines that would load the data back to volatile memory from the non-volatile one. This is the case after applying the first transformation, which makes state-save operations be limited to restoring the register file and special registers. The content of main memory is persisted earlier, when it becomes final.

Example. Consider the program of Fig. 3(a). Following a power failure, the execution resumes from line 6 as the restore routines loads the value of the program counter from non-volatile memory, along with register file, other special registers, and the slice of main memory that was persisted prior to the power failure. However, note that the **LOAD** instruction of line 7 reads the same value for variable a that is loaded earlier as part of the restore routine.

A more efficient strategy is to limit the restore routine to register file and special registers, and make the **LOAD** instruction of line 7 target the non-volatile memory where the data resides. Compared

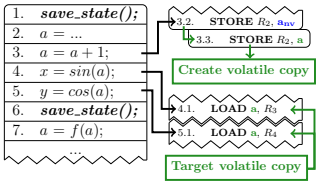


Figure 4: Consolidating read operations.

to a regular checkpoint mechanism, this transformation allows us to remove the instructions that restore variable a from checkpoint data, as the first read instruction that is actually part of the program is relocated to the right address on non-volatile memory.

Fig. 3(b) shows the program after this transformation, which bears reduced energy overhead because the restore routine is no longer concerned with variable a, as it is loaded straight from non-volatile memory if and when necessary. This corresponds to moving the **LOAD** instruction normally be part of the restore to routine for variable a to where in the program variable a is actually read.

Generalization. Similar to the previous transformation, we apply this technique to an arbitrary computation interval as follows. First, we limit restore routines to register file and special registers. Next, for each memory location x, we consider the possibly empty set of memory read instructions $I_r = (I_{r1}, ..., I_{rn})$ that manipulate x and are included in the computation interval. Dually to the first transformation, I_{r1} is the first such instruction and there is no other memory read instruction after the state-save operation at the start of the computation interval. We relocate the target of I_{r1} to non-volatile memory, as that is where the data is to be loaded from.

Whether the remaining $n - 1$ read operations $I_{r2}, ..., I_{rn}$ in a computation interval are to target volatile or non-volatile memory is determined by applying the program transformation that follows.

3.4 Consolidating Read Operations

Starting with a program that exclusively uses volatile memory at stage ⟨3⟩ in Fig. 1, the first two transformations relocate the target of selected read or write operations to non-volatile memory. As data now resides on non-volatile memory near state-save operations, further relocations to non-volatile memory may be required for other read operations. This is the case, for example, for read operations following the last non-volatile write operation that makes data final, as mentioned in Sec. 3.2. Whether this is the most efficient choice, however, is not straightforward to determine.

The third transformation is based on the intuition that *whenever memory operations are relocated to non-volatile memory, it may be convenient to create a volatile copy of data to benefit from lower energy consumption for read operations.*

Example. The program in Fig. 2(b) includes further read operations after line 3 and memory location a is on non-volatile memory as a result of the first transformation. In principle, we should relocate the read instructions on line 4 and 5 to non-volatile memory, as that is where the sought data resides. Because of the higher energy consumption of non-volatile memory, doing so may possibly backfire, outweighing the gains of the first transformation.

We must thus determine whether it is worth paying the penalty for creating a volatile copy of variable a to benefit from the more energy efficient operations there. Such a penalty is represented by an additional **STORE** instruction to create a copy of the data on volatile memory, as shown in Fig. 4. The new **STORE** uses the same source register, hence it represents the only added overhead. The benefit is the improved energy consumption obtained by making the instructions of line 4 and 5 target volatile memory, instead of the non-volatile one. Note that the exact same situation occurs for read instructions following the first **LOAD** instruction in Fig. 3(b).

Consider the frequently used MSP430-FR5969 [16, 28, 34, 35, 40] with internal FRAM as non-volatile memory, and say it runs at $16MHz$, where FRAM accesses require an extra clock cycle. Based on the datasheet [28], we calculate that if read operations in line 4 and 5 target non-volatile memory, the program consumes $1.522nJ$ for these operations. I we pay the penalty of the additional **STORE** instruction, but use volatile memory for all other read operations, the program consumes $1.376nJ$ for the same processing. This is a 10.6% improvement. We accordingly insert an additional **STORE** instruction after line 3 to copy a to volatile memory and we keep the read operations of line 4 and 5 target volatile memory.

Generalization. For each memory location x, we consider the n read instructions $I_{r1}, ..., I_{rn}$ in a computation interval that we need to consolidate, thus excluding those altered by the second transformation. We compute the energy consumption of a single non-volatile memory read instruction as

$$E_{read\_nv} = E_{nv\_read\_cc} * (1 + CC_{read}), \tag{1}$$

where $E_{nv\_read\_cc}$ is the energy consumption per clock cycle of the non-volatile memory read instruction and CC_{read} are the extra clock cycles possibly required, as mixed-volatile microcontrollers may incur in extra clock cycles when operating on the slower non-volatile memory. These clock cycles consume the same energy as a regular non-volatile read operation.

The break-even point between paying the penalty of an additional **STORE** instruction to benefit from more energy-efficient volatile read operations, versus the cost of allocating all read operations to non-volatile memory is determined according to inequality

$$E_{read\_nv} * n < E_{write} + E_{read} * n, \tag{2}$$

where E_{read_nv} is the one of Eq. 1, n is the number of considered memory read instructions, and E_{read} and E_{write} represent the energy consumption of a volatile memory read and write instruction, respectively. This can be rewritten as

$$0 < E_{write} - n * (E_{nv\_read\_cc} * (1 + CC_{read}) - E_{read}). \tag{3}$$

As the energy figures are fixed for a given microcontroller, Eq. 3 is exclusively a function of n, that is, the number of memory read instructions to consolidate in the computation interval. We can accordingly state that creating a volatile copy of the considered memory location is beneficial as long as

$$n > n_{min}, \text{ with } n_{min} = \left\lfloor \frac{E_{write}}{E_{nv\_read\_cc} * (1 + CC_{read}) - E_{read}} \right\rfloor, \tag{4}$$

where n_{min} is the minimum number of memory read instructions to ensure that creating a volatile copy of a memory location incurs in lower overall energy consumption. If the condition of Eq. 4 is not met, we make the n read operations target non-volatile memory.

Interestingly, n_{min} is independent of the specific read/write memory patterns and of program structure. It only depends on hardware

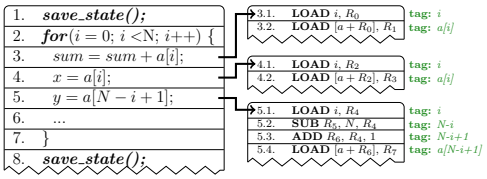


Figure 5: Example of the same group of instructions accessing multiple memory locations.

features. As an example, n_{min} is 0 (2) for the MSP430-FR5969 at a clock frequency of 16MHz (8Mhz). This means that if the microcontroller runs at 16MHz, it is *always* beneficial to create a volatile copy of the relevant memory locations.

4 COMPILE-TIME UNCERTAINTY

The transformation techniques of Sec. 3 rely on program information, such as the order of instruction execution and accessed memory addresses, that may not not be completely available at compile time. Constructs altering the control flow, such as conditional statements or loops, and memory accesses through pointers make these information a function of the run-time state. We describe next how we resolve this uncertainty, making it possible to apply the techniques of Sec. 3 to arbitrary programs.

We distinguish between two types of compile-time uncertainty. *Memory uncertainty* occurs when the exact memory address that a read/write operation targets cannot be determined. We resolve this uncertainty using virtual memory tags, as described in Sec. 4.1. *Instruction uncertainty* occurs when the order of instruction execution is not certain. Addressing this issue requires different techniques depending on program structure. In the interest of brevity, we give an intuition of how we can achieve this in the case of loops in Sec. 4.2. The corresponding generalization is available nonetheless [41].

Here again, the code snippets include both source and machine code for easier illustration, with line numbers pointing to the former, yet ALFRED operates entirely on machine code.

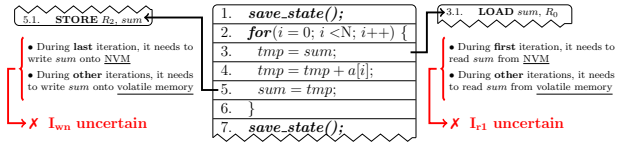
4.1 Memory Uncertainty

Our key observation here is that the techniques of Sec. 3 do not necessarily require exact memory addresses to operate; rather, they must identify the groups of instructions accessing the same memory location, whatever that may be.

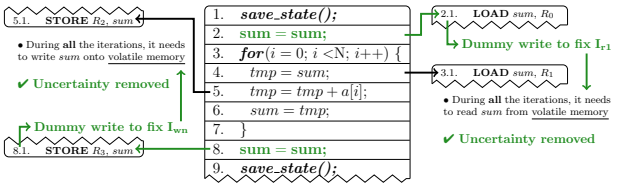
Example. Fig. 5 shows an example. Lines 3, 4, and 5 target multiple memory locations across different iterations of the loop. The corresponding physical addresses in memory change at every iteration.

To apply the techniques of Sec. 3, however, exact knowledge of the physical addresses in memory is not required. We rather need to determine that, at any given iteration of the loop, lines 3 and 4 target the same memory location, whereas line 5 targets a different one. Note that the information available in machine code is insufficient to this end: from that, we can only conclude that lines 3, 4, and 5 access all the addresses in the range $(a[0], a[N-1])$.

We automatically associate a *virtual memory tag* to every memory locations an instruction targets, as shown in Fig. 5. A virtual memory tag is an abstraction of physical memory that aids the application of the techniques of Sec. 3 by succinctly capturing what memory locations are *the same* in a computation interval.



(a) Example of a compile-time uncertainty in a loop.



(b) Normalized form of the loop that removes the compile-time uncertainty.

Figure 6: Example of compile-time uncertainty with loops.

In the program of Fig. 5, we attach the tag $a[i]$ to the memory locations read in lines 3 and 4. Instead, we attach the tag $a[N-i+1]$ to the memory location read in line 5. This information is sufficient for the technique mapping read operations, described in Sec. 3.3, to understand that line 3 and 4 are to be considered as one sequence I'_r, whereas line 5 is to be considered as a different sequence I''_r.

Virtual memory tags are, in a way, similar to debug symbols attached to machine code. They are obtained by inspecting the source code ahead of the corresponding translation, through multiple passes of a dedicated pre-processor. The transformations of Sec. 3 look at these information, instead of the memory locations in machine code. To handle pointers, we combine virtual memory tags with memory alias analysis [14, 32] to identify cases of indirect access to the same memory location. Unlike debug symbols, however, these information is removed from the program at stage ⟨5⟩.

4.2 Instruction Uncertainty → Loops

Key to the application of the program transformations in Sec. 3.2 and Sec. 3.3 is the identification of the last (first) memory write (read) instruction in a computation interval. This may be affected by loops, conditional statements, and function calls that alter the order of instruction execution. Further, whenever the execution of state-save operations depends on run-time information, for example, when a checkpoint call lies in a loop, the span of computation intervals is also undefined at compile time.

We describe next how we address these issues in the case of loops; how we deal with all other cases is available elsewhere [41].

Example. Fig. 6(a) exemplifies the situation. Say we are to apply the mapping of write operations, described in Sec. 3.2. Doing so requires to identify the last memory write instruction I_{wn} before the state-save operation. Depending on the value of i compared to N, the write operation in line 5 may or may not be the one that makes the data final for variable *sum*. The same reasoning is valid when we are to apply the mapping of read operations, described in Sec. 3.3. Depending on the value of i compared to N, the read operation in line 3 may or may not be the first for variable *sum* after the state restore. As a matter of fact, i and N are in control of what write (read) instruction is the I_{wn} (I_{r1}).

One may operate pessimistically and make both the **LOAD** on line 3 and the **STORE** on line 5 target non-volatile memory. This choice may be inefficient, because for all values of i that are neither 0 nor $N-1$, the loop computes intermediate results that are going to be overwritten anyways, so the cost of non-volatile memory operations is unnecessary. To complicate matters, the value of N itself may vary across different executions of the same fragment of code, as it may depend on runtime state.

Normalization. We apply techniques of *program normalization* [4, 51] to resolve this uncertainty, as well as all others that possibly arise when the order of instruction execution depends on run-time information. Program normalization refers to a set of established program transformations designed to facilitate program analysis and automatic parallelization. Many compilers [18] for multi-core processors, for example, include multiple normalization passes.

To resolve the uncertainty in Fig. 6(a), we need to be in the position to persist the value of *sum* once we are sure the loop is over and *before* the state-save operation. Fig. 6(b) shows one way to achieve this. We add a *dummy write* consisting in a pair of **LOAD** and **STORE** instructions for variable *sum after* the loop. These instructions are inserted after code elimination steps and bear no impact on program semantics, but fix where in the code the data for *sum* is final, regardless of the value of i and N. We add a similar instruction *prior* to the loop to fix where the first read for *sum* occurs. We can now make both **STORE** on line 8 and the **LOAD** on line 2 target non-volatile memory without unnecessary overhead. All other operations now concern intermediate results that may be stored on volatile memory. As a result, i and N are no longer in control of what is the I_{wn} (I_{r1}) write (read) instruction that the transformation in Sec. 3.2 and Sec. 3.3 would consider.

The normalization step introduces an overhead. To reduce that, whenever possible we leverage information cached in registers. For example, in Fig. 6(b), the value for *sum* stored in a register in line 6 may be picked up later in line 8, instead of re-loading the value from main memory. Applying this kind of optimization is, however, not always possible, as the content of registers may be overwritten by other instructions that execute in between. In Sec. 6 we prove that, despite the overhead of normalization, ALFRED programs are more energy-efficient than their regular counterparts.

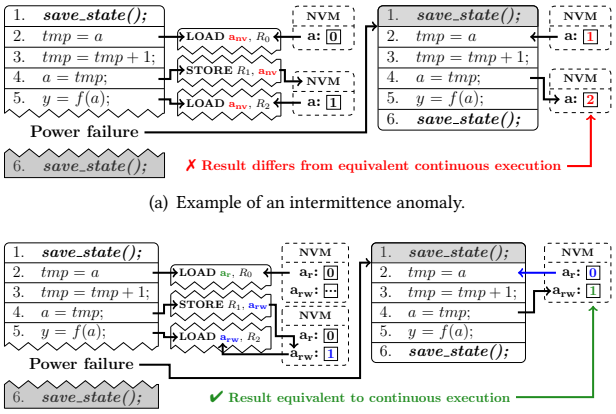
We apply similar normalization passes to resolve the uncertainty possibly arising with conditional statements, function calls, and when the span of computation intervals is undefined at compile time. Further optimizations to abate the overhead we generate are also possible depending on the programming construct [41].

5 MEMORY HANDLING

To make the techniques of Sec. 3 and Sec. 4 work correctly, we devise a custom memory layout that can be determined at compile-time and a schema to address the possible intermittence anomalies.

5.1 Memory Layout

Despite virtual memory tags ensure we can correctly group instructions, we still need to identify the addresses of the volatile or non-volatile versions of the same memory location. We address this problem by placing the volatile and non-volatile versions of a



(a) Example of an intermittence anomaly.

(b) Example of how to avoid the intermittence anomaly with memory versioning.

Figure 7: Example of an intermittence anomaly.

memory location at the same offset with respect to the corresponding base address. Note that the compiler treats the two segments as separate memory sections and makes them start at a fixed offset. This ensures that the volatile and non-volatile versions of the same memory location are at a fixed offset, too.

We can then express the address of the non-volatile version of a memory location as a function of the address of its volatile version, and vice versa. This allows us to allocate memory operations to either memory segment with ease, even in the presence of indirect accesses through pointers. To make an instruction that originally operates on volatile memory now target the non-volatile one, we add the offset between volatile and non-volatile segments to its target address. We operate the other way around when we make an instruction target volatile memory from the non-volatile one. When the instruction executes, it retrieves the address information that are unknown at compile-time and calculates the actual target.

5.2 Dealing with Intermittence Anomalies

Using mixed-volatile platforms, the re-executions of non-idempotent portions of code may cause intermittence anomalies [34, 42, 43, 45, 49], consisting in behaviors unattainable in a continuous execution. The problem possibly arises regardless of whether the code is written directly by programmers [34, 42, 43, 49] or is the result of the program transformations of Sec. 3.

Example. Consider the program of Fig. 7(a). Variable a is non-volatile. Following the state-save operation on line 1, the current value of variable a, that is 0, is initially retrieved from non-volatile memory. The execution continues and line 4 updates the value of variable a on non-volatile memory to 1. This is how a continuous execution would normally unfold.

Imagine a power failure happens right after the execution of line 4. When the device resumes as energy is back, the program restores the program state from non-volatile memory, which includes the program counter. The program then resumes from line 2, which is re-executed. As variable a on non-volatile memory retains the effects of the operations executed before the power failure, the value read by line 2 is now 1, that is, the value written in line 4 before the power failure in the previous power cycle. This causes

line 4 to produce a result that is unattainable in any continuous execution, as it updates the value of variable a to 2, instead of 1.

Many such situations exist that possibly cause erratic behaviors, including memory operations on the stack and heap [42, 43, 49].

Memory versioning. Intermittence anomalies happen whenever a power failure introduces a Write-After-Read (WAR) hazard [34, 42, 49] on a non-volatile memory location. In Fig. 7(a), the memory read of line 2 and the memory write of line 4 represent a WAR hazard for variable a. Several techniques exist to avoid the occurrence of intermittence anomalies [26, 34–36, 42, 43, 49]. In general, it is sufficient to break the sequences of instructions involved in WAR hazards [34, 42, 43, 49] so the involved instructions necessarily execute in different power cycles. Existing solutions place additional checkpoints [49] or enforce transactional semantics to specific portions of code [26, 34–36].

We use a different approach that tightly integrates with the compile-time operation of ALFRED. First, to reduce the number of instructions possibly re-executed, every call to a state-save operation in ALFRED systematically dumps the state on non-volatile memory, regardless of the current energy level. This is different than in many checkpoint systems, where the decision to take a checkpoint is subject to current energy levels [7, 8, 11, 46]. The overhead we impose by doing this is very limited, as state-save operations are limited to register file and program counter after applying the transformations of Sec. 3.

For each computation interval, we then create two versions of each non-volatile memory location possibly involved in a WAR hazard. One version is a *read-only copy* and contains the result produced by previous computation intervals; the other version is a *read-and-write copy* and contains the result of the considered computation interval. We direct the memory read (write) instructions to the read-only (read-and-write) copy. This ensures that in case of a re-execution, the read operations access the values produced by the previous computation interval, as the (partial) results of the current computation interval remain invisible in the read-and-write copies. When transitioning to the next computation interval, the read-only and read-and-write copies are swapped to allow the next computation interval to access the (now, read-only) data of the computation interval just concluded.

Fig. 7(b) shows how this solves the intermittence anomaly of Fig. 7(a). Line 2 reads variable a's read-only copy, whereas line 4 writes variable a's read-and-write copy. Line 4 accordingly reads variable a's read-and-write copy, as it needs the data that line 4 produces. If a power failure happens after line 4 and line 2 is eventually re-executed, that read operation still targets a read-only copy, which correctly reports 0. Instead, after swapping the two copies, the next computation interval correctly accesses the copy of variable a that reports value 1, equivalently to a continuous execution.

We apply this technique as a further code processing step, as shown in stage ⟨5⟩ of Fig. 1. First, we identify the WAR hazards. For each memory write instruction I_w on a non-volatile memory location with tag x, we check if there exists a memory read instruction I_r such that *i)* I_r targets a non-volatile memory location with the same memory tag x, and *ii)* I_r may execute before I_w, that is, I_r happens before I_w in the control-flow graph. If such I_r exists, the pair (I_w, I_r) represents a WAR hazard.

Next, we create the read-only and read-and-write copies by doubling the space that the compiler normally reserves to the data structure x refers to. As we allocate the two copies in contiguous memory cells, their relative offset is fixed and may be used at compile time to direct the memory operation to either copy. We then make I_r target the read-only copy, together with every memory read instruction that operate on x and executes before I_w. In contrast, we make I_w target the read-and-write copy of x, together with all corresponding memory read instructions that happen after I_w. As this processing occurs after program normalization, the compile-time uncertainty in the order of instruction execution or in the span of computation intervals is already resolved at this stage.

6 EVALUATION

Our evaluation of ALFRED considers multiple dimensions. We describe next the experimental setup and the corresponding results.

6.1 Setting

We opt for system emulation over hardware-based experimentation, as it enables better control on experiment parameters and allows us to carefully reproduce program execution and energy patterns across ALFRED and the baselines we consider. Because of the highly non-deterministic behavior of energy sources, achieving perfect reproducibility is extremely challenging using real devices [22].

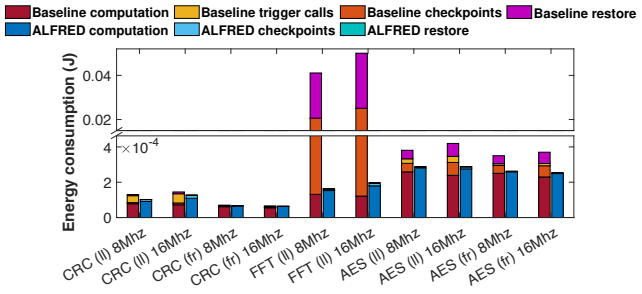
Tool and implementation. We use ScEpTIC [38, 43], an open-source extensible emulation tool for intermittent programs. ScEpTIC emulates the execution of the LLVM Intermediate Representation (IR) [33] of a source code and provides bindings for implementing custom extensions to *i)* apply program transformations and *ii)* map specific performance metrics of the IR to those of machine-specific code, for example, to measure energy consumption.

ScEpTIC organizes the LLVM IR into a set of Abstract Syntax Trees (ASTs), one for each function in source code. Each of these ASTs is generated by augmenting the original LLVM AST with dedicated ScEpTIC elements, which represent information on the emulated instructions and architectural elements, such as I/O operations and registers. We implement the pipeline of Fig. 1 from stage ⟨3⟩ onwards as a set of further transformations of these ASTs. A detailed description of this implementation is available [41], along with an open-source prototype release of our ScEpTIC extension implementing ALFRED transformations [39].

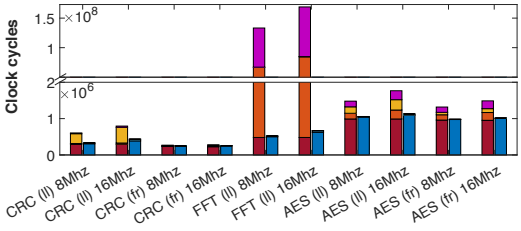
We also implement a machine-specific ScEpTIC extension to map the execution of the IR to the energy consumption of the MSP430-FR5969 [28], a low-power MCU that features an internal and directly-addressable FRAM as non-volatile memory. The MSP430-FR5969 is often employed for intermittent computing [8, 34, 35, 46, 49]. Our extension takes as configuration parameters the energy consumption per clock cycle of various operating modes of the MSP430-FR5969 [28], such as regular computation, (non-)volatile memory read/write operations, and peripheral accesses.

Dimension	Possible instances
Memory configuration	Volatile, NonVolatile
Checkpoint call placement	Loop-Latch, Function-Return, IdempotentBoundaries
Checkpoint execution	Probe, Execute

Figure 8: Design dimensions for baselines.



(a) Energy consumption



(b) Clock cycles

Figure 9: Energy consumption and number of clock cycles comparing ALFRED with a baseline using Volatile, Probe, and either Loop-Latch or Function-Return. *For a baseline, 'll' or 'fr' indicate Loop-Latch or Function-Return.*

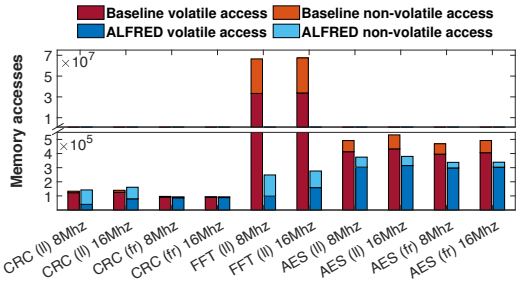


Figure 10: Memory accesses comparing ALFRED with a baseline using Volatile, Probe, and either Loop-Latch or Function-Return.

Benchmark	Baseline VM (bytes)	Baseline NVM (bytes)	ALFRED VM (bytes)	ALFRED NVM (bytes)
CRC (ll) 8Mhz	812	1688	6	850
CRC (ll) 16Mhz	812	1688	26	850
CRC (fr) 8Mhz	812	1636	26	810
CRC (fr) 16Mhz	812	1636	30	810
FFT (ll) 8Mhz	16708	33514	64	29082
FFT (ll) 16Mhz	16708	33514	2188	29082
AES (ll) 8Mhz	1276	2614	40	1334
AES (ll) 16Mhz	1276	2614	42	1334
AES (fr) 8Mhz	1276	2614	58	1338
AES (fr) 16Mhz	1276	2614	62	1338

Figure 11: Volatile memory (VM) and non-volatile memory (NVM) in ALFRED against a baseline using Volatile, Probe, and either Loop-Latch or Function-Return.

Baselines and benchmarks. We compare ALFRED with checkpoint mechanisms that instrument programs automatically [11, 36, 46, 49] by placing calls to a checkpoint library at specific places in the code. We do not consider, instead, checkpoints mechanisms that use interrupts to trigger the execution of checkpoints [7, 8, 29–31], including TICS [31] and the work of Jayakumar et. al [30], as checkpoints do not execute at pre-defined places in the code and thus boundaries of computation intervals cannot be identified. The latter is required for ALFRED to apply the transformations of Sec. 3.

Due to the variety of existing compile-time checkpoint systems, we abstract the key design dimensions in a framework that allows us to instantiate baselines that correspond to existing works, while retaining the ability to explore configurations not strictly corresponding to available systems. Fig. 8 summarizes these dimensions.

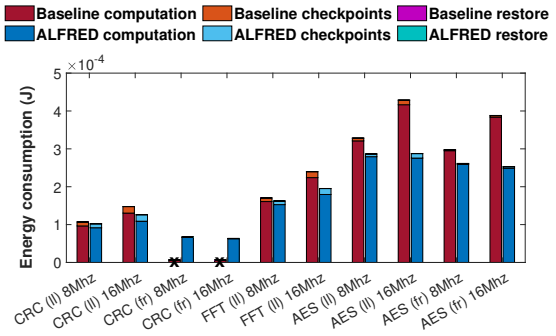
On such design dimension is the *memory configuration*. We consider two possible instances, Volatile and NonVolatile. Volatile allocates the entire main memory onto volatile memory. To ensure forward progress, each checkpoint must therefore save the content of main memory, register file, and special registers onto non-volatile memory. This is the case, for example, in Mementos [46] and HarvOS [11]. Instead, the NonVolatile instance allocates the entire main memory onto non-volatile memory. Here checkpoints may be limited to saving the content of the register file and program counter onto non-volatile memory, as main memory is already non-volatile. This is the case of Ratchet [49].

A given memory configuration is typically coupled to a dedicated *strategy for placing checkpoint calls* in the code. Systems that only use volatile main memory, as in Volatile, may place checkpoints using the Loop-Latch or Function-Return placement strategies of Mementos [46]. Systems that only use non-volatile main memory, as in NonVolatile, place checkpoints using the strategy of Ratchet [49]. This entails identifying idempotent sections of the code and placing checkpoint calls at their boundaries. We accordingly call this strategy IdempotentBoundaries. This ensures that intermittence anomalies are solved by construction, as re-execution of code only occurs across idempotent sections of code.

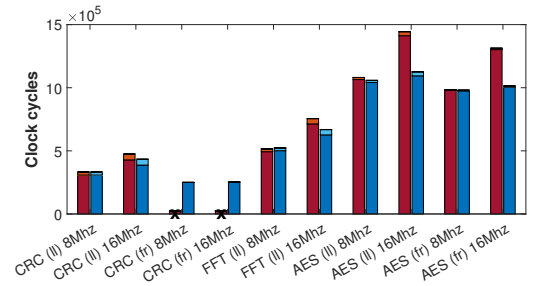
Once checkpoint calls are placed in the code, the *checkpoint execution policy* dictates the conditions that possibly determine the actual execution of a checkpoint. Indeed, a checkpoint call may systematically cause a checkpoint to be written on non-volatile memory, or rather probe the current energy levels first, for example, through an ADC query, and postpone the execution of a checkpoint if energy is deemed sufficient to continue without it. The former kind of behavior, which we call Execute, is the case of Ratchet [49], Chinchilla [36], and TICS [31] when it relies on checkpoints manually placed by developers; the latter kind of behavior we call Probe and reflects HarvOS [11] and Mementos [46].

A combination of memory configuration, strategy for placing checkpoint calls, and checkpoint execution policy represents the single baseline. Note that not all combinations of these dimensions are necessarily meaningful. For instance a NonVolatile memory configuration necessarily requires checkpoints to behave in an Execute manner, or the risk of intermittence anomalies would be too high and the overhead to address them correspondingly prohibitive [45]. As ALFRED requires as input a placement of state-saving operations, when comparing with a certain baseline we use the same such placement. Moreover, being the FRAM performance and energy consumption affected by the MCU operating frequency [28], we consider both $8Mhz$ and $16Mhz$ clock configurations.

Applications deployed onto battery-less devices typically consist in a sense-process-transmit loop [1, 13, 27]. Checkpoint techniques



(a) Energy consumption



(b) Clock cycles

Figure 12: Energy consumption and number of clock cycles comparing ALFRED with a baseline using NonVolatile, Execute, and either Loop-Latch or Function-Return.

and memory configurations mainly affect processing, whereas sensing and transmissions impose the same overhead regardless of the former. For this reason, similar to related literature, we focus on processing functionality and consider a diverse set of benchmarks commonly used in intermittent computing [7, 8, 26, 29, 43, 46, 49]: Cyclic Redundancy Check (CRC) for data integrity, Advanced Encryption Standard (AES) for data encryption, and Fast Fourier Transform (FFT) for signal analysis. We use Clang version 7.1.0 to compile their open-source implementations, as available in the MiBench2 [25] suite, using the default compiler settings. The binaries output by the compiler never exceed 30*kB*.

Metrics and energy patterns. We focus on *energy consumption* and *number of clock cycles* necessary to complete a fixed workload. Being harvested energy scarce, the former captures how battery-less devices perform when deployed and represents an indication of the perceived end-user performance [1, 13, 27]. The latter allows us to identify how the overhead of ALFRED affects performance, as it mainly consists in the additional instructions required to address the compile-time uncertainties, as described in Sec. 4. Note that the two metrics are not necessarily proportional, because non-volatile memory accesses may require extra clock cycles and consume more energy than accesses to volatile memory [28]. ALFRED may also introduce an overhead in the form of additional memory occupation, as the same data may need space in both volatile and non-volatile memory. To measure this, we keep track of the use of *volatile/non-volatile memory spaces* during the execution. To gain a deeper insight into the performance trends we also record *volatile/non-volatile memory accesses*, and the execution of *checkpoint and restore operations*.

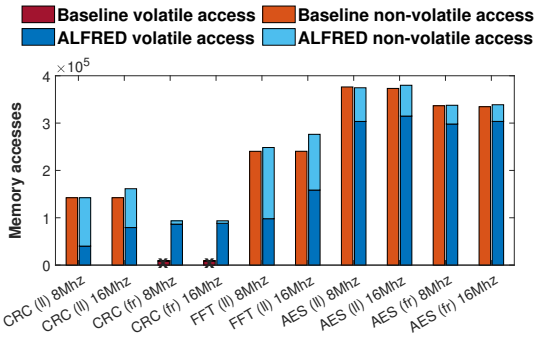


Figure 13: Memory accesses comparing ALFRED with a baseline using NonVolatile, Execute, and either Loop-Latch or Function-Return.

Patterns of ambient energy harvesting may be simulated using IV surfaces [21, 22] or by repetitively simulating power failures after a pre-determined number of executed clock cycles [40, 49]. The former makes simulated power failures happen at arbitrary points in times and provides little control on experiment executions, making it difficult to sweep the parameter space. The latter may be tuned according to statistical models, and offers better control on experiment executions by properly tuning model parameters. The behavior of ALFRED is largely independent of the specific number of executed clock cycles between consecutive power failures; we therefore opt for the second option.

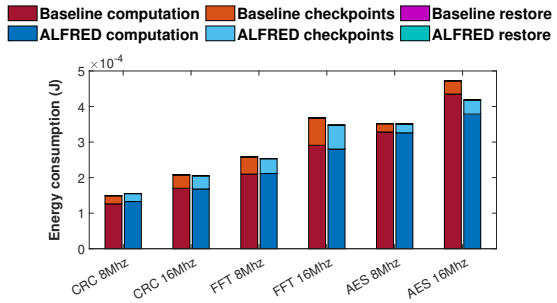
We model an RF energy source. To determine the number of executed clock cycles between two power failures, we rely on the existing measurements from ten real RF energy sources used for the evaluation of Mementos [46], which features a MCU configuration compatible with our setup. To evaluate multiple scenarios, including the worst and best possible ones, we execute each benchmark considering the minimum, average, and maximum number of executed clock cycles between power failures, modeled after the aforementioned real measurements. We report on the results obtained in the average scenario, as there is no sensible difference among the three scenarios. Note that, when using the Probe strategy, we make sure that the last checkpoint call before a power failure is the one that does save a checkpoint, as this represents the same behavior of real scenarios.

6.2 Results

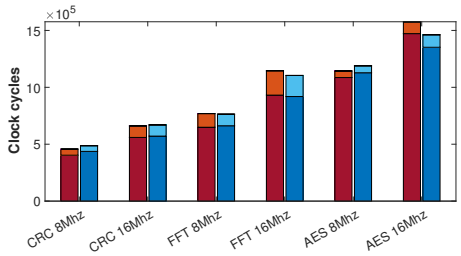
We consider three combinations of the design dimensions of Fig. 8.

Checkpointing from volatile memory. We begin comparing with a baseline configuration using Volatile, Probe, and either Loop-Latch or Function-Return. This configuration represents Mementos [46] and solutions inspired by its design [11, 36].

Fig. 9 shows the results we obtain. Fig. 9(a) shows how, depending on the benchmark, ALFRED provides up to several-fold improvements in energy consumption to complete the fixed workload. CRC computation is the simplest benchmark and has little state to make persistent. The improvements are marginal here, especially when using Function-Return as the checkpoint placement strategy, which is unsuited to the structure of the code in the first place. The improvements grow as the complexity of the code increases. Computing FFTs is the most complex benchmark we consider, and



(a) Energy consumption



(b) Clock cycles

Figure 14: Energy consumption and number of clock cycles comparing ALFRED with a baseline using NonVolatile, Execute, and IdempotentBoundaries.

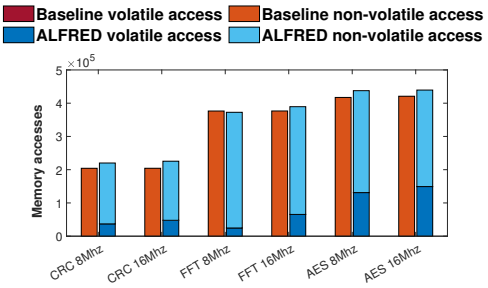


Figure 15: Memory accesses comparing ALFRED with a baseline using NonVolatile, Execute, and IdempotentBoundaries.

Benchmark	Baseline VM Size (Bytes)	Baseline NVM Size (Bytes)	ALFRED VM Size (Bytes)	ALFRED NVM Size (Bytes)
CRC 8Mhz	0	826	6	854
CRC 16Mhz	0	826	16	854
FFT 8Mhz	0	16730	40	29074
FFT 16Mhz	0	16730	1116	29074
AES 8Mhz	0	1294	24	1342
AES 16Mhz	0	1294	40	1342

Figure 16: Volatile memory (VM) and non-volatile memory (NVM) in ALFRED against a baseline using NonVolatile, Execute, and IdempotentBoundaries

the improvements are largest in this case. These observations are confirmed by the measurements of clock cycles, shown in Fig. 9(b).

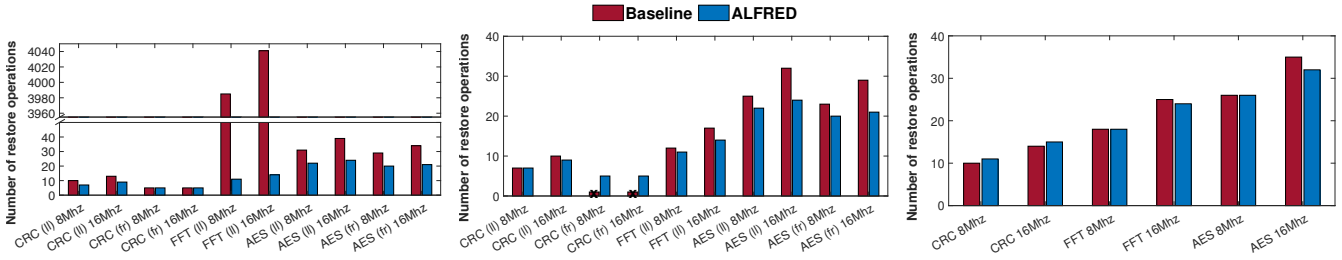
Fig. 10 provides a finer-grained view on the results in this specific setting. The small state in CRC corresponds to the fewest number of memory accesses, especially in volatile memory, as little data is to be made persistent to cross power failures. In both AES and FFT, ALFRED greatly reduces the number of memory accesses. Checkpoint operations in these benchmarks must load a significant amount of data from volatile memory and copy it to non-volatile memory for creating the necessary persistent state. These accesses are not necessary in ALFRED, as data is made persistent as soon as it becomes final; therefore, checkpoint operations do not process main memory, but only register file and program counter. As for the nature of memory accesses, ALFRED can promote, on average, 65% of the accesses the baseline executes on non-volatile memory to volatile memory instead, with a minimum of 20% in *CRC* at 8*Mhz* with a Loop-Latch configuration and a maximum of 95% in *CRC* with a Function-Return configuration. This is a key factor that grants ALFRED better energy performance.

Fig. 11 reports on the use of volatile/non-volatile memory. In the baseline, the state to be preserved across power failures includes the entire volatile memory, the register file, and special registers. Requiring to double-buffer the state saved to non-volatile memory, its use in the baseline amounts to more than double the use of volatile memory. In both CRC and AES, ALFRED requires to double buffer less than *4%* of the program state to avoid intermittence anomalies, resulting in a drastically lower use of non-volatile memory. Interestingly, despite a significant improvement in energy consumption, ALFRED promotes very few memory locations to volatile memory. These correspond to the memory locations that are most frequently accessed, as shown in Fig. 10.

Moving to non-volatile memory. Fig. 12 shows the results we obtain comparing with configuration using NonVolatile, Execute, and either Loop-Latch or Function-Return. This combination represents a hybrid solution combining features of several existing systems [11, 36, 46]. As Loop-Latch and Function-Return do not necessarily guarantee that intermittence anomalies cannot occur, we lend our versioning technique, described in Sec. 5, to the baseline. The major difference between ALFRED and the baseline, therefore, is in the use of volatile or non-volatile memory.

Fig. 12(a) shows that the program transformations we devise are effective at improving the energy performance of intermittent programs. Significant improvements are visible across all benchmarks. Configurations exist where the baseline cannot complete the workload using the energy patterns we consider, as in the case of the CRC benchmark when using Function-Return to place checkpoints. In contrast, ALFRED reduces energy consumption to an extent that allows the workload to successfully complete.

The corresponding results in the number of executed clock cycles, shown in Fig. 12(b), enables a further observation. When running at 16*Mhz*, the baseline shows a significant increase of clock cycles, at least 20% with respect to the same benchmark running at 8*Mhz*. The cause of this increase is in the extra clock cycles required to access the FRAM when the MCU is clocked at 16*Mhz*. In the same scenarios, ALFRED shows a lower increase of clock cycles when comparing the 8*Mhz* and 16*Mhz* configurations, especially in the AES benchmark. Rather than massively employing non-volatile memory, ALFRED switches to volatile memory whenever possible within a computation interval. This not only reduces the clock cycles spent waiting for non-volatile memory access, but also enables energy savings in the operations that involve temporary data or intermediate results that do not need persistency.



(a) Baseline using Volatile, Probe, Loop-Latch or Function-Return (b) Baseline using NonVolatile, Execute, Loop-Latch or Function-Return (c) Baseline using NonVolatile, Execute, Idempotent-Boundaries

Figure 17: Restore operations to complete the fixed workload in ALFRED compared to the three baselines.

Fig. 13 confirms this reasoning, showing that ALFRED promotes an average of 65% of the non-volatile memory accesses in the baseline to the more energy-efficient volatile memory. This functionality grants ALFRED the completion of the CRC benchmark when using the Function-Return configuration. As the baseline directs all memory accesses to non-volatile memory, the resulting energy consumption causes CRC to be stuck in a livelock, as energy is insufficient to reach a checkpoint that would enable forward progress. This situation is called "non-termination" bug [17]. Instead, in the case of CRC, ALFRED promotes more than 95% of the non-volatile memory accesses in the baseline to volatile memory. This significantly reduces the energy consumption of memory accesses to an extent that allows ALFRED to complete the workload.

Note that the use non-volatile memory in the baseline is the same as ALFRED, shown in Fig. 11, as they employ the same technique to avoid intermittence anomalies. The difference in memory occupation consists in the data that ALFRED allocates onto volatile memory, which ultimately yields lower energy consumption.

Ruling out intermittence anomalies. We compare the performance of ALFRED with a configuration using NonVolatile, Execute, and IdempotentBoundaries, as in Ratchet [49]. Because of the specific placement of checkpoint calls and the Execute policy, intermittence anomalies cannot occur by construction. ALFRED and the baseline here only differ in memory management.

Fig. 14 shows the results. Fig. 14(a) illustrates the performance in energy consumption; this time, the improvements of ALFRED are generally less marked than those seen when using Loop-Latch or Function-Return to place checkpoints. The results in the number of executed clock cycles are coherent with these trends, as illustrated in Fig. 14(b). This is because IdempotentBoundaries tends to create much shorter computation intervals, sometimes solely worth a few instructions; therefore, ALFRED has fewer opportunities to operate on the energy-efficient volatile memory. ALFRED still improves the energy efficiency overall, especially for the AES benchmark and the configurations running at 16*Mhz*. At this clock frequency, non-volatile memory operations induce higher overhead due to the necessary wait cycles. Sparing operations on non-volatile memory allows the system not to pay this overhead.

Fig. 15 and Fig. 16 provide an assessment on ALFRED's ability to employ volatile memory whenever convenient. ALFRED promotes the use of volatile memory from the non-volatile use in the baseline in up to 30% of the cases. The impact of this, however, is more limited here because of the shorter computation intervals, as discussed above. In this plot, it also becomes apparent that sometimes, the total number of memory accesses in ALFRED is higher than in the baseline. This is a combined effect of the program transformation techniques of Sec. 3 and of the normalization passes in Sec. 4. The increase in the total number of memory accesses, however, does not yield a penalty in energy consumption, as a significant fraction of these added accesses operate on volatile memory.

These results are confirmed in Fig. 16. Despite being the program partitioned in non-idempotent code sections, our techniques to address compile-time uncertainties introduce intermittence anomalies that require ALFRED to double-buffer a portion of the program state. This situation is particularly evident with FFT. Fig. 16 provides additional evidence of how ALFRED employs volatile memory for frequently-accessed data, which ultimately yields lower energy consumption across all benchmarks executed at 16*Mhz*.

Restore operations. We complete the discussion by showing in Fig. 17 the number of restore operations executed in ALFRED compared to those in the three baseline configurations we consider.

The plots demonstrate that the better energy efficiency provided by ALFRED allows the system to restore the state less times. This trend is especially visible in Fig. 17(a) and Fig. 17(b). As a result, ALFRED shifts the available energy budget to useful application processing, leading to workloads that finish sooner compared to the performance offered by the baselines.

7 CONCLUSION

ALFRED is a virtual memory abstraction for intermittent computing that spares programmers the need to manage application state across memory facilities, and efficiently employ volatile and non-volatile memory to improve energy consumption, while ensuring forward progress. The mapping from virtual to volatile or non-volatile memory is decided at compile time to use volatile memory whenever possible because of the lower energy consumption, resorting to non-volatile memory to ensure forward progress. In contrast to existing works, the memory mapping is not fixed at variable level, but is adjusted at different places in the code, based on read-/write patterns and program structure. Our evaluation indicates that, depending on the workload, ALFRED provides several-fold improvements in energy consumption compared to the multiple baselines we consider, leading to a similar improvement in the number of restore operations required to complete a fixed workload.

Acknowledgments. We thank the shepherd and reviewers for the feedback received on the initial submission. This work was supported by the Google Faculty Award programme and by the Swedish Foundation for Strategic Research (SSF).

REFERENCES

[1] M. Afanasov, N. A. Bhatti, D. Campagna, G. Caslini, F. M. Centonze, K. Dolui, A. Maioli, E. Barone, M. H. Alizai, J. H. Siddiqui, and L. Mottola. 2020. Battery-Less Zero-Maintenance Embedded Sensing at the Mithræum of Circus Maximus. In *Proceedings of the 18th Conference on Embedded Networked Sensor Systems (SenSys '20)*.

[2] S. Ahmed, A. Bakar, N. A. Bhatti, M. H. Alizai, J. H. Siddiqui, and L. Mottola. 2019. The Betrayal of Constant Power × Time: Finding the Missing Joules of Transiently-powered Computers. In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*.

[3] S. Ahmed, M. H. Bhatti, N. A. Alizai, J. H. Siddiqui, and L. Mottola. 2019. Efficient Intermittent Computing with Differential Checkpointing. In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2019)*.

[4] Z. Ammarguellat. 1992. A Control-Flow Normalization Algorithm and Its Complexity. *IEEE Transactions on Software Engineering* (1992).

[5] J. A. Anderson and G. J. Lipovski. 1974. A Virtual Memory for Microprocessors. In *Proceedings of the 2nd Annual Symposium on Computer Architecture (ISCA '75)*.

[6] A. R. Arreola, D. Balsamo, G. V. Merrett, and A. S. Weddell. 2018. RESTOP: Retaining External Peripheral State in Intermittently-Powered Sensor Systems. *Sensors* (2018).

[7] D. Balsamo, A. S. Weddell, A. Das, A. R. Arreola, D. Brunelli, B. M. Al-Hashimi, G. V. Merrett, and L. 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).

[8] D. Balsamo, A. S. Weddell, G. V. Merrett, B. M. Al-Hashimi, D. Brunelli, and L. Benini. 2015. Hibernus: Sustaining Computation During Intermittent Supply for Energy-Harvesting Systems. *IEEE Embedded Systems Letters* (2015).

[9] G. Berthou, T. Delizy, K. Marquet, T. Risset, and G. Salagnac. 2018. Sytare: a Lightweight Kernel for NVRAM-Based Transiently-Powered Systems. *IEEE Trans. Comput.* (2018).

[10] N. A. Bhatti, M. H. Alizai, A. A. Syed, and L. Mottola. 2016. Energy Harvesting and Wireless Transfer in Sensor Network Applications: Concepts and Experiences. *ACM Transactions on Sensor Networks* (2016).

[11] N. A. Bhatti and L. Mottola. 2017. HarvOS: Efficient Code Instrumentation for Transiently-powered Embedded Sensing. In *Proceedings of the 16th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*.

[12] A. Branco, L. Mottola, M. H. Alizai, and J. H. Siddiqui. 2019. Intermittent Asynchronous Peripheral Operations. In *Proceedings of the 17th Conference on Embedded Networked Sensor Systems (SENSYS)*.

[13] Q. Chen, Y. Liu, G. Liu, Q. Yang, X. Shi, H. Gao, L. Su, and Q. Li. 2017. Harvest Energy from the Water: A Self-Sustained Wireless Water Quality Sensing System. *ACM Transactions on Embedded Computing Systems* (2017).

[14] J. Choi, M. Burke, and P. Carini. 1993. Efficient Flow-Sensitive Interprocedural Computation of Pointer-Induced Aliases and Side Effects. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '93)*.

[15] A. Colin, G. Harvey, B. Lucia, and A. P. Sample. 2016. An Energy-interference-free Hardware-Software Debugger for Intermittent Energy-harvesting Systems. *SIGOPS Operating Systems Review* (2016).

[16] A. Colin and B. 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 (OOPSLA)*.

[17] A. Colin and B. Lucia. 2018. Termination Checking and Task Decomposition for Task-based Intermittent Programs. In *Proceedings of the 27th International Conference on Compiler Construction (CC 2018)*.

[18] C. Dave, H. Bae, S. Min, S. Lee, R. Eigenmann, and S. Midkiff. 2009. Cetus: A Source-to-Source Compiler Infrastructure for Multicores. *Computer* (2009).

[19] P. J. Denning. 1970. Virtual Memory. *Comput. Surveys* (1970).

[20] F. Fraternali, B. Balaji, Y. Agarwal, L. Benini, and R. Gupta. 2018. Pible: Battery-Free Mote for Perpetual Indoor BLE Applications. In *Proceedings of the 5th Conference on Systems for Built Environments (BUILDSYS)*.

[21] M. Furlong, J. Hester, K. Storer, and J. Sorber. 2016. Realistic Simulation for Tiny Batteryless Sensors. In *Proceedings of the 4th International Workshop on Energy Harvesting and Energy-Neutral Sensing Systems (ENSsys'16)*.

[22] J. Hester, T. Scott, and J. Sorber. 2014. Ekho: Realistic and Repeatable Experimentation for Tiny Energy-harvesting Sensors. In *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems (SenSys '14)*.

[23] J. Hester and J. Sorber. 2017. The Future of Sensing is Batteryless, Intermittent, and Awesome. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems (SENSYS)*.

[24] 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*. 1–13.

[25] M. Hicks. 2016 (last access: Oct 15th, 2021). MiBench2 porting to IoT devices. https://github.com/impedimentToProgress/MiBench2.

[26] M. Hicks. 2017. Clank: Architectural Support for Intermittent Computation. In *Proceedings of the 44th annual International Symposium on Computer Architecture (ISCA)*.

[27] N. Ikeda, R. Shigeta, J. Shiomi, and Y. Kawahara. 2020. Soil-Monitoring Sensor Powered by Temperature Difference between Air and Shallow Underground Soil. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies (IMWUT)* (2020).

[28] Texas Instruments. 2017 (last access: Oct 15th, 2021). MSP430-FR5969 datasheet. https://www.ti.com/lit/ds/symlink/msp430fr5969.pdf.

[29] H. Jayakumar, A. Raha, W. S. Lee, and V. Raghunathan. 2015. QuickRecall: A HW/SW Approach for Computing Across Power Cycles in Transiently Powered Computers. *ACM Journal on Emerging Technologies in Computing Systems* (2015).

[30] H. Jayakumar, A. Raha, J. R. Stevens, and V. Raghunathan. 2017. Energy-Aware Memory Mapping for Hybrid FRAM-SRAM MCUs in Intermittently-Powered IoT Devices. *ACM Transactions on Embedded Computing Systems* (2017).

[31] V. Kortbeek, K. S. Yildirim, A. Bakar, J. Sorber, J. Hester, and P. Pawełczak. 2020. Time-Sensitive Intermittent Computing Meets Legacy Software. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*.

[32] W. Landi and B. G. Ryder. 1992. A Safe Approximate Algorithm for Interprocedural Aliasing. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation (PLDI '92)*.

[33] llvm 2003 (last access: Oct 15th, 2021). The LLVM Compiler Infrastructure. https://llvm.org/.

[34] B. Lucia and B. 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 (PLDI)*.

[35] K. Maeng, A. Colin, and B. Lucia. 2017. Alpaca: Intermittent Execution Without Checkpoints. *Proceedings of the ACM Programming Languages* (2017).

[36] K. Maeng and B. Lucia. 2018. Adaptive dynamic checkpointing for safe efficient intermittent computing. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[37] K. Maeng and B. Lucia. 2019. Supporting Peripherals in Intermittent Systems with Just-in-Time Checkpoints. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[38] A. Maioli. 2021 (last access: Oct 15th, 2021). ScEpTIC documentation and source code. http://sceptic.neslab.it/.

[39] A. Maioli. 2021 (last access: Oct 15th, 2021). ScEpTIC extension implementing a prototype of ALFRED pipeline. http://alfred.neslab.it/.

[40] A. Maioli and L. Mottola. 2020. Intermittence Anomalies Not Considered Harmful. In *Proceedings of the 8th International Workshop on Energy Harvesting and Energy-Neutral Sensing Systems (ENSsys '20)*.

[41] A. Maioli and L. Mottola. 2021 (last access: Oct 15th, 2021). ALFRED: Virtual Memory for Intermittent Computing. https://arxiv.org/abs/2110.07542 . arXiv:2110.07542

[42] A. Maioli, L. Mottola, M. H. Alizai, and J. H. Siddiqui. 2019. On Intermittence Bugs in the Battery-Less Internet of Things (WIP Paper). In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*.

[43] A. Maioli, L. Mottola, M. H. Alizai, and J. H. Siddiqui. 2021. Discovering the Hidden Anomalies of Intermittent Computing. In *Proceedings of the 2021 International Conference on Embedded Wireless Systems and Networks (EWSN 2021)*.

[44] A. Y. Majid, C. Delle Donne, K. Maeng, A. Colin, K. S. Yildirim, B. Lucia, and P. Pawelczak. 2020. Dynamic Task-Based Intermittent Execution for Energy-Harvesting Devices. *ACM Transactions on Sensor Networks* (2020).

[45] B. Ransford and B. Lucia. 2014. Nonvolatile Memory is a Broken Time Machine. In *Proceedings of the Workshop on Memory Systems Performance and Correctness (MSPC '14)*.

[46] B. Ransford, J. Sorber, and K. Fu. 2011. Mementos: System Support for Long-running Computation on RFID-scale Devices. *ACM SIGARCH Computer Architecture News* (2011).

[47] E. Ruppel and B. Lucia. 2019. Transactional Concurrency Control for Intermittent, Energy-harvesting Computing Systems. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[48] E. Sazonov, H. Li, D. Curry, and P. Pillay. 2009. Self-Powered Sensors for Monitoring of Highway Bridges. *IEEE Sensors Journal* (2009).

[49] J. Van Der Woude and M. Hicks. 2016. Intermittent Computation Without Hardware Support or Programmer Intervention. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI)*.

[50] K. Vijayaraghavan and R. Rajamani. 2010. Novel Batteryless Wireless Sensor for Traffic-Flow Measurement. *IEEE Transactions on Vehicular Technology* (2010).

[51] T. Wang, X. Su, and P. Ma. 2008. Program Normalization for Removing Code Variations. In *Proceedings International Conference on Software Engineering*.

[52] K. S. Yildirim, A. Y. Majid, D. Patoukas, K. Schaper, P. Pawelczak, and J. Hester. 2018. InK: Reactive Kernel for Tiny Batteryless Sensors. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems (SENSYS)*.