Intermittence Anomalies not Considered Harmful

Andrea Maioli
Politecnico di Milano, Italy
andrea1.maioli@polimi.it

Luca Mottola
Politecnico di Milano, Italy and RISE, Sweden
luca.mottola@polimi.it

ABSTRACT

We consider a new perspective on intermittence anomalies arising in intermittently-computing mixed-volatile systems. Existing forward progress techniques avoid such anomalies by enforcing a computation that corresponds to a continuous one, introducing a significant overhead. We take a different stand: by allowing the presence of specific anomalies, we make the program aware of intermittence, unlocking new design patterns. We argue about the various possibilities emerging from this and we make the concept concrete by applying it to loops. We show how intermittence anomalies allow to preserve the results of loop iterations across power failures, without requiring to save the device's volatile state after each iteration. Compared to existing checkpoint mechanisms, our technique shows on average a 35.2x lower energy consumption and a 48.4x lower execution time across several staple benchmarks.

CCS CONCEPTS

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

KEYWORDS

Intermittence anomalies, intermittent computing.

ACM Reference Format:
Andrea Maioli and Luca Mottola. 2020. Intermittence Anomalies not Considered Harmful. In *The 8th International Workshop on Energy Harvesting and Energy-Neutral Sensing Systems (ENSsys '20), November 16–19, 2020, Virtual Event, Japan.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3417308.3430266

1 INTRODUCTION

Ambient energy harvesting for embedded sensing devices removes the maintenance costs and environment impact associated with battery replacement and disposal. Being harvested energy erratic and usually not sufficient to power a device continuously, these devices experience frequent power failures. Executions thus become *intermittent* [5], as periods of active computation are interrupted by periods where the device is powered off and recharges its energy buffer. Frequent power failures harm program forward progress, as power outages cause a device to shut down and loose the computational state, making it restart from scratch when power returns.

Managing persistent state. As we point out in Sec. 2, ensuring program forward progress across power failures requires saving a

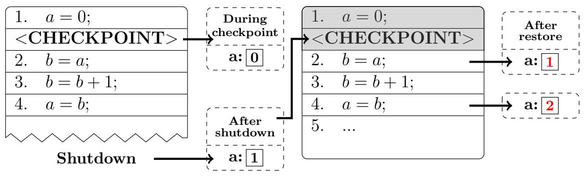


Figure 1: Example of intermittence anomaly. *A checkpoint saves the volatile state and then line* 4 *updates* a *to* 1. *Next, a power failure occurrs. When energy returns, computation resumes from line* 2. *Being* a *non-volatile, it is not included in the checkpoint and retains the effects that line* 4 *produced during the previous power cycle. The execution produces a different result than a continuous execution.*

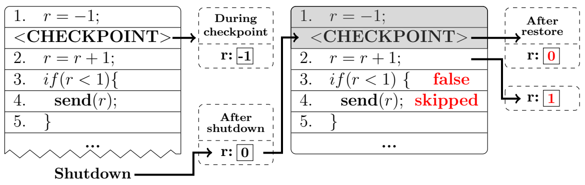


Figure 2: Example of intermittence-aware program. *Line* 2 *experiences the same intermittence anomaly as in Fig. 1. Variable* r *tracks the number of power failures.*

snapshot of the volatile state, namely a *checkpoint*, onto a non-volatile memory (NVM) location, which can be internal or external to the Micro Controller Unit (MCU). When power returns, restoring a checkpoint allows the MCU to resume the computation from where it stopped, as checkpoints contain a copy of main memory, program counter, and register file. Mixed-volatile systems [10, 11, 18] feature an internal NVM that they use as a portion of main memory. NVM is not included into checkpoints, as it already ensures persistency. This reduces checkpoint overhead, as the system saves only the volatile slice of main memory.

The use of mixed-volatile platforms may cause *intermittence anomalies* [11, 14], due to repeated executions of non-idempotent code. Fig. 1 shows an example. Being variable a non-volatile, it is not included in the checkpoint. The execution reaches line 4, which alters a, then a power failure happens. When the device resumes, it restores the volatile state from the checkpoint, and the execution resumes from line 2. Being non-volatile, a retained the effect that line 4 produced during the previous power cycle. This leads to a result that is unattainable in a continuous execution, as the re-execution of line 4 updates a to 2 instead of 1.

Avoiding intermittence anomalies requires to save additional checkpoints in specific program locations to break harmful sequences [14, 18]. For example, in Fig. 1 a checkpoint between line 2 and line 4 solves the issue. Generally, the more portions of the main memory are non-volatile, the more frequently checkpoints must be placed to avoid intermittence anomalies. This may nullify the performance gains due to reduced volatile state.

Intermittence awareness. Existing checkpoint mechanisms [8, 11, 13, 18] generally aim at enforcing an execution that corresponds to the continuous one. We take a different stand. We show how deliberately allowing the presence of specific intermittence anomalies in mixed-volatile MCUs may unlock new program design patterns. We call this concept *intermittence awareness*.

Intentionally allowing specific intermittence anomalies allows developers to consider intermittence as a program input. The resulting *intermittence-aware* program can change its behavior according to when and where a power failure happens. Fig. 2 shows an example. Variable r is non-volatile. Similar to Fig. 1, when the execution resumes after a power failure, r retains the effects that line 2 produced during the previous power cycle. By deliberately allowing the anomaly to occur, we can use r to track the number of power failures since the last checkpoint, as line 2 increments r every time the computation resumes. This ensures that line 4 is not re-executed when the program resumes, as the if statement of line 3 evaluates to *false*. Such behavior is not possible with existing approaches [3, 11, 16, 18], as they enforce results equivalent to a continuous computation, that is, r must equal 0 after line 2.

Intermittence awareness gives developers new degrees of freedom, as it unlocks new design patterns that would otherwise not be possible, applicable to either program *control flow* or *data flow*. Fig. 2 shows an example where intermittence awareness allows developers to affect the program *control flow* when resuming after a power failure. In constrast, by allowing the intermittence anomaly Fig. 1, we make the computation dependent from the number of power failures by altering its data flow.

Intermittence-aware loops. To demonstrate the use of intermittence awareness, we use it to reduce checkpoint overhead inside loops, as described in Sec. 3.

Power failures cause a device to loose the work done inside loops, unless a checkpoint is saved at the end of each iteration. This introduces a significant overhead, yet it is necessary in the absence of a priori knowledge on energy provisioning patterns. We identify a set of variables, called *loop state set*, that represent the minimum data to preserve. We instrument a loop by allocating its loop state set onto NVM, thus making it intermittence aware. This makes a checkpoint before the loop sufficient for resuming the computation from the latest loop iteration, ensuring forward progress with much lower overhead. Checkpoint frequency and size decrease, as checkpoints inside loops are no longer required and they do not include the loop state set.

Nonetheless, every time a device resumes from a power failure, it restores the latest checkpoint, introducing a startup overhead. Our technique also allows us to reduce this. We exploit intermittence awareness to skip the latest unfinished loop iteration, instead of re-executing it. The latter mitigates the startup overhead at the cost of a decreased precision, resulting in a behavior similar to the loop-perforation technique [17] used in approximate computing [15].

To enable the application of our loop instrumentation technique, we design and implement the LAPSUS[1] programming abstraction. LAPSUS exposes a small set of macros that allow developers to apply our loop instrumentation techniques without manually managing checkpoints, allocating variables, or designing dedicated data structures. Moreover, LAPSUS allows developers to decide and fine-tune where to apply our technique for mitigating the startup overhead when the program resumes after power failures.

In Sec. 4 we evaluate how LAPSUS affects the overhead of existing checkpoint mechanisms, based on staple intermittent computing benchmarks. Experimental results show that LAPSUS significantly lowers the overhead of existing approaches, reducing on average the number of executed instructions by $48.4x$, and obtaining a $35.2x$ lower energy consumption and a $48.4x$ lower execution time. In the worst case, that is, the CRC benchmark where checkpoint size is small, LAPSUS lowers the energy consumption overehead by $2.08x$. Instead, with higher checkpoint sizes, such as the implementation of Dijkstra algorithm, LAPSUS lowers the energy consumption overhead up to $227.79x$.

2 BACKGROUND AND RELATED WORK

We provide the necessary background and a brief discussion of related work here.

Ensuring forward progress. Various techniques [1–3, 10, 16, 18] adopt the concept of checkpoint to ensure program forward progress across power failures. Depending on how and where checkpoints execute, we classify these techniques as *dynamic* or *static*.

Dynamic checkpoint mechanisms, such as Hibernus [1, 2] and QuickRecall [10], rely on external interrupts that signal a low energy buffer for saving checkpoints. The program may thus be preempted at arbitrary places to take a checkpoint. Differently, *static* checkpoint mechanisms [3, 16, 18] place checkpoint calls in the program at compile time, fixing where checkpoints execute in the code. Among these systems, Ratchet [18] always saves a checkpoint when the execution encounters a checkpoint call. Mementos [16] and HarvOS [3] execute "trigger" calls to first verify the energy buffer for deciding whether to save a checkpoint.

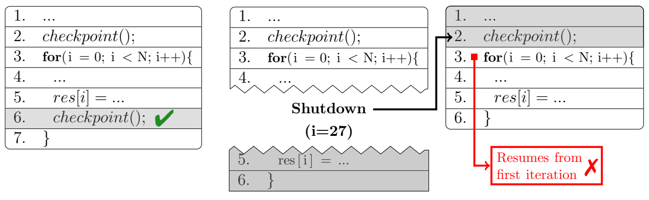
In contrast to checkpoint mechanisms that are applicable to unmodified source code, task-based programming abstractions [4, 6, 12, 19] require programmers to split the application logic in separate tasks executing with transactional semantics.

Intermittence anomalies. Checkpoint operations save the MCU volatile state into a NVM location. When power returns, restoring a checkpoint allows the MCU to resume the computation from where it stopped. However, the resulting runtime state may differ from the one of a continuous execution. In such a scenario, we define the runtime state as *anomalous*.

Resuming the computation with an anomalous runtime state may lead to *intermittence anomalies* [11, 14, 18], consisting in unexpected behaviors unattainable in a continuous execution. The effects of intermittence anomalies depend on how the program interacts with the anomalous part of the runtime state [14]. For example, in Fig. 1, the re-execution of lines 2-4 introduces a write-after-read (WAR) hazard [11, 14, 18]. Being a non-volatile, the re-execution of line 2 sees the effects that line 4 produced on a during the previous power cycle, as if line 2 re-executes just after line 4.

Avoiding intermittence anomalies. Two classes of techniques exist to verify the presence of intermittence anomalies [14] and to avoid their occurence [4, 8, 11, 13, 14, 18, 19]. One class of approaches breaks the sequence of operations involved in WAR hazards using a checkpoint [13, 14, 18] to avoid the operations accessing

[1]**L**ow-overhead intermittence-**A**ware **P**rogram in**S**trumentation techniq**U**e for loop**S**



(a) Checkpoint **(b) No checkpoint**

Figure 3: Preserving forward progress inside loops. *Fig.* ***(a)*** *shows a checkpoint placement that preserves progress across power failures. Fig.* ***(b)*** *shows the effects of removing the checkpoint. After a power failure occurs, the loop restarts all over again.*

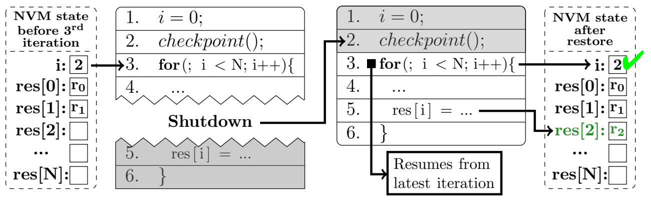


Figure 4: Example of an intermittence-aware loop. *A power failure happens during the third iteration. When the checkpoint is restored, the computation resumes from the beginning of the loop, but being* `i` *and* `res` *non-volatile, they retain the value right before the previous power cycles. Hence, the loop resumes from the third iteration, that is, the one interrupted by the power failure.*

```
1. int res[N];
2. int a = 0;
3. for(int i = 0; i < N; i++){
   ...
4.   a = f(a);
5.   res[i] = g(a);
6. }
```

(a) Original

```
1. LOOP-STATE(res[N], int);
2. C-LOOP-STATE(a, int, 0);
3. PERSISTENT-LOOP(i, i < N) {
   ...
4.   C-WRITE(a, i) = f(C-READ(a, i));
5.   res[i] = g(C-USE(a, i));
6. }
```

(b) Instrumented

Figure 5: Example of LAPSUS instrumentation macros. *Fig.* ***(a)*** *shows the program to instrument; Fig.* ***(b)*** *shows the instrumented program using LAPSUS macros.*

the anomalous runtime state. For example, in Fig. 1, a checkpoint between lines 2 and 4 removes the WAR hazard and solves the anomaly. The second class of approaches creates multiple versions of the involved variables [8, 11], ensuring that read and write operations involved in the WAR hazard access different versions.

To our knowledge, no previous work considers the possibility of taking advantage from selected intermittence anomalies.

3 INTERMITTENCE-AWARE LOOPS

We show one possible application of the concept of intermittence awareness for mixed-volatile MCUs. We rely on specific intermittence anomalies to preserve the computation across loops, reducing checkpoint overhead. In doing so, we primarily target static checkpoint mechanisms. Dynamic checkpoint mechanisms may trigger checkpoints at any place in the code and only when it is strictly required to do so, essentially yielding no re-executions as the device likely immediately dies. This spares the overhead of trigger calls, at the cost of dedicated hardware support [1, 2, 10].

3.1 Example

Existing static checkpoint mechanisms [3, 16, 18] require to possibly save a checkpoint at the end of each loop iteration to preserve the work done inside loops. Such a conservative choice is necessary as, in general, erratic energy patterns may not provide guarantees on the complete executions of multiple loop iterations.

Consider the example of Fig. 3. A power failure inside the loop causes the device to resume from the latest checkpoint. The latter is saved at line 2, thus the computation resumes prior to the loop, re-executing it from scratch and wasting 28 iterations. Placing checkpoint calls at the end of each loop iteration introduces an overhead even if checkpoints do not actually take place, as certain operations occur anyways when executing the call, such as probing the energy buffers for their current energy content [3, 16].

Unlike existing techniques [3, 11, 16, 18], intermittence awareness allows specific intermittence anomalies to preserve the loop computational state across power failures without requiring a checkpoint at each iteration. Fig. 4 shows how to apply this concept to the example of Fig. 3. A checkpoint executes at line 2 and the loop starts the first iteration. Variables i and res are non-volatile. The execution reaches line 5, which stores the result of the first iteration into $res[0]$. Next, i increments to 1, and the second iteration completes. A power failure occurs during the third iteration. The computation eventually resumes from the checkpoint of line 2. Being i and res non-volatile, they retain state at the previous power cycle: i has a value of 2, and res stores the results of the previous loop iterations. Thus, the loop starts from the third iteration, as if a checkpoint is saved at the end of the second iteration.

These accesses represent an anomaly, as the value of i is produced during the previous power cycle. By allowing such an anomaly, we obtain the same results of a checkpoint placed at the end of each loop iteration, but without its overhead. Existing techniques for mixed-volatile systems [11, 18] do not allow this behavior. Despite they allow to directly allocate variables into NVM, as we do, they enforce executions equivalent to continuous ones. Thus, they would apply variable versioning [11] or place checkpoints [18] to ensure that line 3 and 5 do not access the anomalous value of variable i.

3.2 Instrumenting Loops

Loop state set. We first need to identify the variables representing the minimum set of data we must preserve across power failures, which we call *loop state set*. This includes, for example, the loop iterator and the variables carrying loop intermediate or final results. In the example of Fig. 3a, the loop state set includes variables i and res, which are the loop iterator and the results vector.

We allocate the loop state set into NVM. Variables not included in the loop state set remain at their original memory location. They are not necessary for resuming the computation without restarting the loop from the beginning, and thus we do not require to preserve them across power failures. A simple example is (volatile) variables local to the loop body, which are recomputed at every loop iteration.

Altering the loop. We remove any checkpoint inside the loop body and place a checkpoint before the loop statement. This ensures that the computation resumes at the beginning of the loop when power fails during the loop execution. Finally, we remove the initialization of the loop iterator from the loop statement, and we place it before the only checkpoint remaining. This modification ensures that the loop resumes from the latest iteration and not from the first one, as the iterator is no longer re-initialized when resuming.

Fig. 4 shows an example. By allocating the loop state set onto NVM and by removing any checkpoint inside the loop, we are deliberately allowing intermittence anomalies. In a sense, we make the loop iterator i function as a checkpoint, as it saves onto NVM the index i of the last completed iteration. Once i increments, no power failure can lead to the re-execution of a loop iteration previous to i.

LAPSUS. To allow the application of intermittence awareness to loops, we design a programming abstraction called LAPSUS. We can use LAPSUS with a broad range of static checkpoint mechanisms, as our techniques do not rely on specific ones. LAPSUS provides a set of macros that allow developers to instrument a program by specifying the loop to be instrumented and its loop state set.

Fig. 5 shows an example. The original code is in Fig. 5a, whereas Fig. 5b shows the instrumented one. First, we substitute the loop construct with the macro **`PERSISTENT-LOOP`**, which takes two arguments: the loop iterator, and the loop condition. **`PERSISTENT-LOOP`** allocates the loop iterator into NVM and initializes it. Then, **`PERSISTENT-LOOP`** generates the *for* loop statement and places a checkpoint before it. Next, we specify the loop state set, which includes the variables *res*, *a*, and *i*. To that end, we substitute each variable declaration with the macro **`LOOP-STATE`**, except for the loop iterator i, which LAPSUS already identifies with **`PERSISTENT-LOOP`**. **`LOOP-STATE`** takes three arguments: the variable name, the variable type, and the initialization value, which is optional. **`LOOP-STATE`** allocates the variables into NVM and initializes them.

3.3 Avoiding Unwanted Anomalies

By allocating the loop state set onto NVM, we may introduce additional unwanted anomalies. Fig. 6a shows an example. Variable a is non-volatile, as it is included in the loop state set. Line 5 represents a WAR hazard [11, 14, 18] that leads to an intermittence anomaly. It first reads the value of a from NVM, executes function f, then writes the result back to NVM. As no checkpoint happens between read and write in line 5, a power failure during or after the function call causes an unwanted intermittence anomaly.

The technique we describe next remedies this issue for scalar variables, with additional overhead. It is not applicable for more complex data structures, such as arrays or linked lists. Addressing this limitation opens up interesting avenues for future work.

Versioning. To avoid placing a checkpoint inside the loop body, we apply a versioning technique. Fig. 6b shows how to avoid the intermittence anomaly of Fig. 6a. Variable a becomes a vector of two elements, each representing a version of a. At each iteration, a write operations target a copy and a read operations target the other. To carry the results across loop iterations, a read and write versions switch after every loop iteration.

This access pattern breaks the sequence of operations involved in the WAR hazard, as now line 5 read and write operations target different copies of a. As such, a power failure can no longer cause line 5 read operation to access an anomalous value, and we can avoid the intermittence anomaly without inserting a checkpoint.

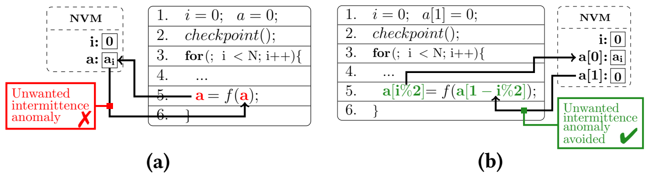

```
1.  i = 0;  a = 0;
2.  checkpoint();
3.  for(; i < N; i++){
4.    ...
5.  a = f(a);
6.  }
```
```
1.  i = 0;  a[1] = 0;
2.  checkpoint();
3.  for(; i < N; i++){
4.    ...
5.    a[i%2]= f(a[1 − i%2]);
6.  }
```
NVM
i: 0
a: a_i
Unwanted intermittence anomaly ✗
(a)
NVM
i: 0
a[0]: a_i
a[1]: 0
Unwanted intermittence anomaly avoided ✔
(b)

Figure 6: Avoiding unwanted intermittence anomalies in an intermittence-aware loop. *In Fig.* ***(a)****, line* 5 *read and write operations represents a WAR hazard [11, 14, 18] on variable* `a`*. Fig.* ***(b)*** *shows how to avoid the unwanted anomaly of variable* `a`*.*

LAPSUS support. LAPSUS includes macros to protect variables against unwanted intermittence anomalies. We use the example of Fig. 5a, where variable a exposes the same anomaly as Fig. 6a.

Being a part of the loop state set, we substitute its declaration at line 2 with the macro **`P-LOOP-STATE`**, where P stands for protected. **`P-LOOP-STATE`** takes the same arguments of **`LOOP-STATE`**, but it also creates the two copies of variable a that our technique requires. Next, we make read and write operations target the correct copy of a. To this end, LAPSUS provides three macros: **`P-WRITE`**, **`P-READ`**, and **`P-USE`**. They take two arguments: a variable and the loop iterator. **`P-READ`** and **`P-WRITE`** target the copy reserved for read and write operations, respectively. For example, at line 4 of Fig. 5a, we substitute the definition of variable a with **`P-WRITE(a, i)`**. Similarly, we substitute **`P-READ(a, i)`** in line 4. Instead, we use **`P-USE`** to access a value written by a previous operation in the same iteration, as in line 5 in Fig. 5a. Fig. 5b shows the final result. Note that we must address all protected variable accesses using the corresponding macro, even for accesses outside the loop.

3.4 Restore Approximation

Intermittence awareness not only reduces checkpoint overhead, but also makes resume operations more efficient, as the device restores a lower amount of data from the checkpoint.

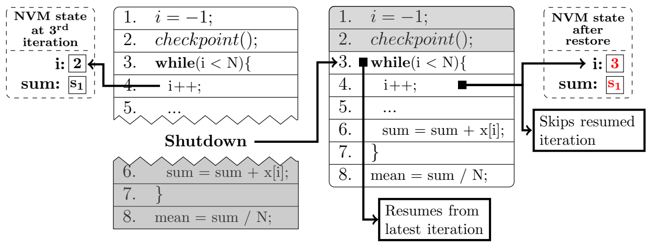
Nonetheless, we may further tune our technique to mitigate the overhead when resuming. Fig. 7 shows an example. Here we intentionally place the increment of the loop iterator i at the beginning of the loop body. Let us suppose a power failure happens during the third loop iteration. When the computation resumes, i increments as first operation and the loop resumes from the fourth iteration, jumping the iteration interrupted by the power failure.

As a result, we obtain a behavior similar to loop perforation techniques [17] used in approximate computing [15], where iterations are skipped to trade accuracy for reduced energy consumption or execution time. Instead of considering a certain perforation rate to decide which iteration to skip, we skip iterations every time the device resumes after a power failure.

LAPSUS supports both the regular and the approximate approach when resuming. Programmers use macro **`APPROX-PERSISTENT-LOOP`** for selecting the approximation strategy, in place of **`PERSISTENT-LOOP`**. The two macros act similarly and take the same arguments. They declare the loop iterator as non-volatile, initialize it, and select the appropriate loop construct.

4 EVALUATION

We discuss our experimental setup and early results we gather to assess feasibility and potential impact of intermittence awareness.



```
1. i = −1;
2. checkpoint();
3. while(i < N){
4.   i++;
5.   ...
6.   sum = sum + x[i];
7. }
8. mean = sum / N;
```

```
1. i = −1;
2. checkpoint();
3. while(i < N){
4.   i++;
5.   ...
6.   sum = sum + x[i];
7. }
8. mean = sum / N;
```

Figure 7: An intermittence-aware loop with restore approximation. *At the second iteration, line* 2 *increments the non-volatile loop iterator* `i` *to* 2*. Execution then resumes from the beginning of the loop after a power failure, but* `i` *retains the effects produced during the previous power cycle. Hence, the loop resumes from the third iteration, decrease result precision for mitigating the startup overhead.*

4.1 Setup

We consider the MSP430-FR5969 [9] MCU, an ultra-low power MCU often adopted in intermittent computing [2, 11, 12, 16, 18].

Baseline and benchmarks. We evaluate the performance of our technique by comparing it against a generic trigger-based static checkpoint mechanism that, akin to existing systems [3, 16], uses NVM only for storing checkpoints. At runtime, when a checkpoint call executes, it queries the ADC to decide whether to execute a checkpoint. We use the default compiler configuration when producing machine code [3, 16]. We call this approach TRIGGER.

We consider benchmarks commonly used in intermittent computing [1, 2, 8, 10, 16, 18], including Cyclic Redundancy Check (CRC) for data integrity, Fast Fourier Transform (FFT) for signal analysis, and the Dijkstra algorithm for finding the shortest path among nodes in a graph. We take these benchmarks from the open-source implementation of the MiBench2 [7] benchmark suite.

We do not quantitatively evaluate the approximate restore technique of Sec. 3.4. As with any approximation technique [15], the degree of acceptable approximation is inherently application-specific and thus an unbiased comparative evaluation is difficult.

Metrics. We focus on the main loops of each benchmark. We compare the increase in *i)* number of executed machine-code instructions, *ii)* energy consumption, and *iii)* execution time that LAPSUS and TRIGGER show compared to the non-instrumented program.

We calculate the increase in the number of machine-code instructions by identifying the loop body operations that differ from the non-instrumented program, that are, FRAM accesses, operations to protect against unwanted intermittence anomalies, trigger calls, and actual checkpoints. We then calculate the increase in energy consumption and execution time by considering the executed clock cycles, the energy consumption of each clock cycle, and the energy consumption and access latency for the ADC and FRAM usages [9].

We calculate the energy consumption per clock cycle of various operating modes as $e_x = \frac{V_{cc} * I_x}{f_{mcu}}$, where V_{cc} is the operating voltage ($3V$) and I_x is the current draw of the MCU under the operating mode x. We also consider an operating clock frequency f_{mcu} of either $8Mhz$ and $16Mhz$, as FRAM accesses require one wait state at $16Mhz$. Being not specified in the datasheet, we calculate the current draw I_{fram} of the MCU when it stores only the data segment into FRAM as $\frac{I_{fram\_uni} - I_{sram}}{2} + I_{sram}$, where I_{sram} and I_{fram_uni} are the current draws when the MCU operates respectively from SRAM and FRAM. Note that I_{fram_uni} refers to two FRAM accesses per clock cycle: one for instruction fetch and one for data access.

We consider trigger calls to happen at every loop iteration and the possible checkpoint to save the minimum amount of data. This places the baseline in the best possible conditions.

4.2 Results

Fig. 8 reports in logarithmic scale the overhead of single operations for LAPSUS and TRIGGER compared to the non-instrumented program across the benchmarks we consider. For TRIGGER, we report both the costs of trigger calls and of actual checkpoints, as trigger calls do not necessarily yield a checkpoint.

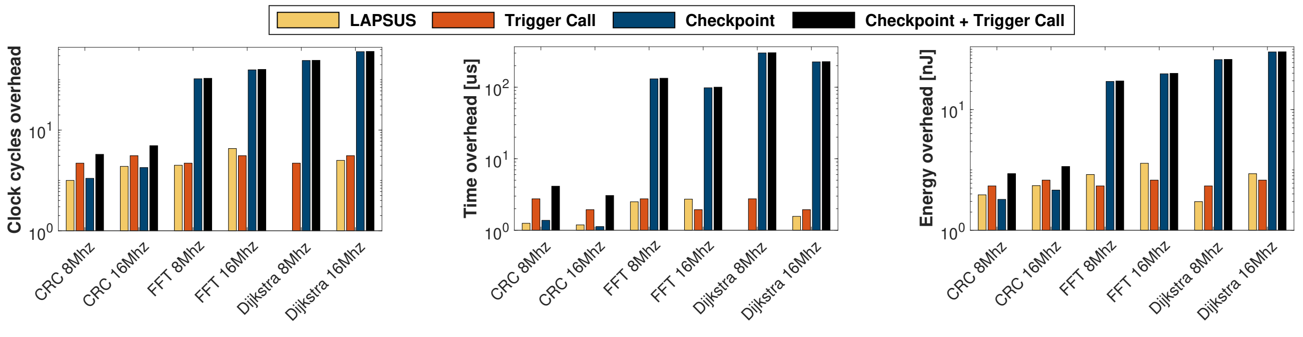
Overall, TRIGGER's complete checkpoint operations require on average a $48.4x$ more clock cycles and execution time than LAPSUS, as illustrated in Fig. 8a and Fig. 8b. LAPSUS consumes on average $35.2x$ less energy than TRIGGER complete checkpoint operations, as Fig. 8c shows. Compared to trigger calls alone, on average LAPSUS executes 37% fewer clock cycles and has a 37% lower execution time, but it has a 24% higher energy consumption. The latter are due to the cost of accessing FRAM.

The results specifically vary depending on the program structure. In the CRC benchmark, LAPSUS has a lower energy consumption than trigger calls alone, as Fig. 8c reports. LAPSUS instrumentation in CRC bears very low overhead, and querying the ADC results in higher energy consumption. Here we also notice that trigger calls represent most of the overhead of a complete checkpoint.

This is not the case for the FFT and Dijkstra. At $8Mhz$, LAPSUS introduces a lower number of clock cycles than trigger calls alone, as Fig. 8a shows. Compared to the same baseline, however, Fig. 8c reports a higher energy overhead for LAPSUS in the FFT benchmark, as now FRAM accesses are more costly than querying the ADC. This is is due to the additional operations to protect the execution against unwanted intermittence anomalies, as explained in Sec. 3.3. Increasing the clock to $16Mhz$ is further detrimental to LAPSUS performance, as now FRAM accesses require one wait state. Notwithstanding the higher energy consumption than trigger calls alone for FTT and Dijkstra, LAPSUS requires $99x$ less energy than complete checkpoint operations, as Fig. 8c shows.

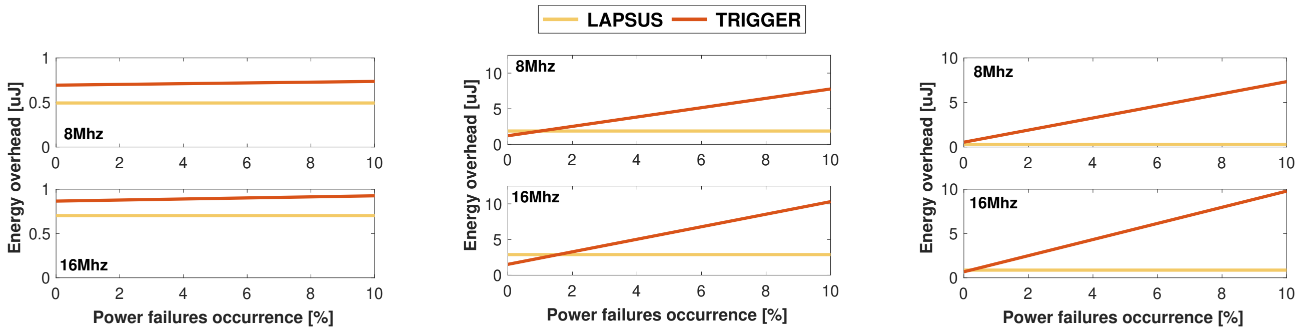
We also investigate how LAPSUS and TRIGGER energy overhead increase with the frequency of power failures. As TRIGGER executes a trigger call at the end of each loop iteration, the frequency of power failures also represents the number of trigger calls converting to a checkpoint. Note that Ransford et al. [16] reports 16 power failures during the execution of the CRC benchmark with $2Kb$ of data to checkpoint when using RF energy sources, corresponding to a 1.56% of trigger calls converting to a checkpoint. Being the FFT and Dijkstra benchmarks way more complex than CRC, we expect a higher frequency of checkpoint occurrences there.

Fig. 9 shows the results. LAPSUS performance across the board is constant, as LAPSUS does not save any checkpoint inside loops. In constrast, Fig. 9a shows that the energy overhead of TRIGGER starts above the one of LAPSUS already with infrequent power failures and slowly grows when power failures occur more often. Consistently with the earlier discussion, Fig. 9b demonstrates that LAPSUS energy overhead is larger than TRIGGER only with very



(a) Clock cycles overhead (b) Execution time overhead (c) Energy overhead

Figure 8: Overhead per loop iteration. *LAPSUS on average requires* 48.4x *less clock cycles than TRIGGER complete checkpoint operations, lowering the execution time by* 48.4x *and the energy consumption by* 35.2x. *Trigger calls alone in TRIGGER require on average a* 37% *higher number of clock cycles and execution time. However, LAPSUS FRAM accesses cause a* 24% *higher energy consumption than trigger calls.*



(a) CRC benchmark (b) FFT benchmark (c) Dijkstra benchmark

Figure 9: Energy overhead during complete executions, against a certain rate of power failures. *LAPSUS overhead is constant, whereas TRIGGER overhead increases with more frequent power failures, especially where checkpoints save a significant amount of data. LAPSUS overhead is lower than TRIGGER, except for cases with a scarce frequency of power failures.*

Benchmark	LAPSUS per loop iteration	TRIGGER per checkpoint
CRC	14	5
FFT	33,5	264
Dijkstra	25	605

Figure 10: Average NVM accesses. *Despite TRIGGER requires fewer NVM accesses than LAPSUS for the CRC benchmark at each checkpoint, the latter ultimately yields lower energy overhead, as shown in Fig. 10, because of the energy cost of trigger calls.*

rare power failures. As the latter happen more frequently, LAPSUS becomes most efficient. A similar observation applies to Fig. 9c.

While the energy overhead of LAPSUS is only due to NVM accesses to handle the loop state set, that of TRIGGER comes from a combination of NVM accesses for checkpointing and trigger calls. Fig. 10 reports statistics on NVM accesses for either solution. For LAPSUS, the NVM accesses are an average per loop iteration, whereas for TRIGGER they are required for each checkpoint. Interestingly, NVM accesses for TRIGGER with the CRC benchmark are lower than those for LAPSUS at every loop iteration. We conclude that the better performance of LAPSUS compared to TRIGGER in Fig. 9a is due to the overhead of trigger calls that do not yield a checkpoint. The opposite situation holds for the other benchmarks, even though the two figures are not directly comparable as LAPSUS incurs in the given number of NVM accesses at every iteration, whereas TRIGGER pays the overhead only when checkpointing.

5 CONCLUSION

Intermittence awareness allows the occurrence of specific anomalies to gain new information regarding intermittence, unlocking new design patterns. Developers exploit intermittence awareness to make their program react to intermittence, altering the program control flow and/or data flow accordingly. We make this concept concrete with an instrumentation technique that uses intermittence awareness to reduce checkpoints overhead inside loops. Our technique preserves the loop computational state across power failures, without requiring to save a checkpoint after each iteration. The LAPSUS programming abstraction facilitates developers in applying our technique to loops. We compare LAPSUS against existing trigger-based checkpoint mechanisms. Across the benchmarks we test, on average LAPSUS lowers the energy overhead of existing checkpoint mechanism by 35.2x and reduces the execution time by 48.4x, demonstrating the impact of intermittence awareness.

Our technique has limitations, such as handling non-idempotent accesses to complex data structures. It also introduces some non-determinism that may complicate testing, as program execution becomes dependent on energy patterns. As we plant the seed for intermittence awareness, we also seek to address these issues.

REFERENCES

[1] 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).

[2] 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).

[3] 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).*

[4] 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).*

[5] 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).*

[6] J. Hester, K. Storer, and J. Sorber. 2017. Timely Execution on Intermittently Powered Batteryless Sensors. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems (SENSYS).*

[7] M. Hicks. 2016 (last access: September 18th, 2020). MiBench2 porting to IoT devices. https://github.com/impedimentToProgress/MiBench2.

[8] M. Hicks. 2017. Clank: Architectural Support for Intermittent Computation. (2017).

[9] Texas Instruments. 2017 (last access: September 18th, 2020). MSP430-FR5969 datasheet. https://www.ti.com/lit/ds/symlink/msp430fr5969.pdf.

[10] 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).

[11] 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).*

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

[13] 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).*

[14] 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).*

[15] S. Mittal. 2016. A Survey of Techniques for Approximate Computing. *Comput. Surveys* (2016).

[16] 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).

[17] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard. 2011. Managing Performance vs. Accuracy Trade-Offs with Loop Perforation. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE).*

[18] 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).*

[19] 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).*