On Intermittence Bugs in the Battery-Less Internet of Things (WIP Paper)

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

Luca Mottola
Politecnico di Milano, Italy and RI.SE SICS, Sweden
luca.mottola@polimi.it

Muhammad Hamad Alizai
LUMS, Pakistan
hamad.alizai@lums.edu.pk

Junaid Haroon Siddiqui
LUMS, Pakistan
junaid.siddiqui@lums.edu.pk

Abstract

The resource-constrained devices of the battery-less Internet of Things are powered off energy harvesting and compute *intermittently*, as energy is available. Forward progress of programs is ensured by creating persistent state. Mixed-volatile platforms are thus an asset, as they map slices of the address space onto non-volatile memory. However, these platforms also possibly introduce *intermittence bugs*, where intermittent and continuous executions differ. Our ongoing work on intermittence bugs includes (i) an analysis that demonstrates their presence *in settings that current literature overlooks*; (ii) the design of efficient testing techniques to *check their presence in arbitrary code*, which would be otherwise prohibitive given the sheer number of different executions to check; (iii) the implementation of *an offline tool called* `ScEpTIC` that implements these techniques. `ScEpTIC` finds the same bugs as a brute-force approach, but is *six orders of magnitude faster*.

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

Keywords Intermittence bugs, Intermittent computing, Transiently-powered computing, Mixed-volatile systems.

ACM Reference Format:
Andrea Maioli, Luca Mottola, Muhammad Hamad Alizai, and Junaid Haroon Siddiqui. 2019. On Intermittence Bugs in the Battery-Less Internet of Things (WIP Paper). In *Proceedings of the 20th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '19), June 23, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3316482.3326346

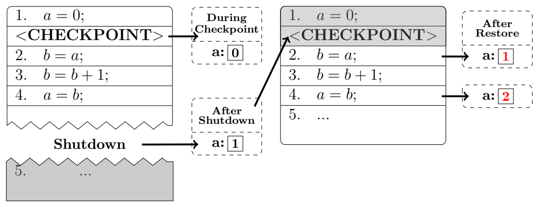

Figure 1. Example of a data access bug.

1 Introduction

Energy harvesting is enabling a battery-less Internet of Things (IoT) [10, 13, 19, 22, 23, 26]. Energy provisioning from the environment, however, is generally erratic. Resource-constrained IoT devices consequently experience frequent shutdowns. Executions become *intermittent*, namely, they consist of intervals of active computation interleaved by periods of recharging energy buffers, such as capacitors, and no computation. Shutdowns normally make devices lose their state, to later restart from scratch when energy is back.

Existing systems rely on persistent state to ensure forward progress of programs [1–3, 12, 14, 21, 25]. Recent solutions target mixed-volatile platforms, such as the MSP430-FRxxxx [11] series, which facilitate handling persistent state as they map slices of the address space to non-volatile memory (NVM), such as FRAM. While data structures allocated on NVM require no additional handling to survive power failures, explicit *checkpoints* create persistent duplicates of volatile data, including registers and program counter.

Existing solutions differ in when to take a checkpoint [1–3, 12, 14, 21, 25]. Systems based on voltage monitoring may, in principle, preempt the execution to take a checkpoint *anywhere* in the code [1, 2]. The execution then continues until either another checkpoint takes place or power fails.

Intermittence bugs. Intermittent execution introduces the possibility of *intermittence bugs* [4, 14, 20, 25], where programs reach a state unattainable in a continous execution.

Fig. 1 shows an example. Variable **a** is allocated on NVM. A checkpoint occurs after line 1. Lines 2 to 4 eventually modify the value of **a**. The execution continues until power fails. When energy is back, the execution resumes with the state of volatile data from the checkpoint, that is, it restarts from line 2. However, **a** being on NVM, its value is the one written at line 4 *before* the power failure, that is, the value produced by a *later* instruction compared to where execution resumes *after* the power failure [20]. Lines 2 to 4 increment **a** again, producing a different result than a continous execution.

In this work, we aim to (i) *comprehensively understand* the conditions possibly leading to an intermittence bug, and (ii) given certain program inputs, *exhaustively test* arbitrary code to identify their presence. Identifying intermittence bugs allows programmers to take informed decisions on target platforms, software configurations, and system support. For example, if given data structures or checkpoint placements appear to introduce intermittence bugs, programmers may opt for a different design or system support.

Related work. We call the kind of bug in Fig. 1 *data access bug*. We demonstrate in Sec. 2, however, that intermittence bugs may appear in other settings as well, overlooked by prior work. Existing literature addresses intermittence bugs at compile time by placing checkpoints to avoid their occurrence [16, 25] or with custom programming abstractions that encourage programmers to write bug-free code [5, 14, 15].

For example, Ratchet [25] is a compile-time solution that instruments source code to prevent data access bugs, but remains largely oblivious to other types of intermittence bugs. Ratchet energy overhead is significant; system performance is arguably practical only where all data but registers and program counter are on NVM. Custom programming abstractions [5, 14, 15] force programmers to learn new concepts and possibly to refactor existing codebases, hampering adoption. They also incur in significant run-time overhead.

A few tools exist to perform general testing of intermittent programs. For example, Ekho [9] allows developers to recreate a given environments by recording and replyaing energy harvesting traces. EDB [4] offers an interactive debugging environment that reduces interference with the energy state of the target device. Siren [7] introduces NVM and energy simulation capabilities in the MSPSim [6] emulator. These tools may somehow be adapted to manually check for data access bugs, if one knows what to look for.

Testing intermittence bugs. Given certain program inputs, exhaustively checking for intermittence bugs is, in principle, a challenge. One should check any possible combination of checkpoint placement and number of instructions until power failure, which may happen at any point in the code. A static analysis of the program would not provide runtime information required for analyzing the NVM, such as accessed addresses and memory content. Performing this check on target hardware is plainly impractical.

We call *execution depth* (ED) the maximum number of instructions possibly re-executed when resuming after a power failure. One of the simplest benchmark in intermittent computing is *CRC* computation, which accounts for $5 \cdot 10^4$ machine-code instructions. Checking all possible combinations for reasonable values of *ED* as explained above results

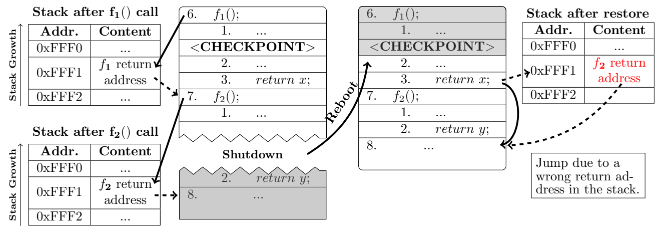

Figure 2. Example of activation record bug.

in analyzing $2.34 \cdot 10^{13}$ machine-code instructions. Our prototype emulator reaches a speed of $1.8 \cdot 10^4$ instructions per second, which means 41 years for testing *CRC* computation.

We present in Sec. 3 techniques that reduce the processing times to identify intermittence bugs. Sec. 4 reports early results using ScEpTIC: a prototype tool we design that implements these techniques. ScEpTIC yields a speedup of six orders of magnitude compared to a brute-force approach, which makes testing for intermittence bugs feasible in a matter of hours in the worst case.

Our ongoing work, discussed in Sec. 5, includes analysing interactions with the external environment through sensors and actuators, as well studying as intermittence bugs intentionally left occurring to implement intermittence-aware programs. Both features are not discussed in the literature.

2 Understanding Intermittence Bugs

We recognize the occurrence of intermittence bugs in settings other than Fig. 1. Key to these insights is reasoning at the level of machine code and raw memory accesses, rather than source code [4, 14, 20]. We identify three kinds of intermittence bug. They all share the same underling write/read pattern, but the consequences are different for each of them.

2.1 Data Access Bug

The example of Fig. 1 is a case of data access bug. In essence, it is caused by a write-after-read hazard on a NVM address. We say a *data access bug* exists whenever x is a memory address in NVM and an ordered sequence of machine-code instructions $I_1, \ldots, I_n$ exists such that:

- I_1 loads a value from an address x,
- I_n modifies the value stored at address x,
- no checkpoint exists in the sequence $I_1, \ldots, I_n$.

These conditions entail that if a power failure occurs after I_n, the system resumes before I_1 which is then re-executed; I_1 then reads the value produced by I_n before the power failure, that is, from a later instruction.

This type of intermittence bug is the only one recognized in the literature [4, 14, 20, 25]. A possible fix for this bug is placing a checkpoint between I_1 and I_n to avoid re-executing the *load* operation when resuming [25].

2.2 Activation Record Bug

An *activation record bug* is a program state where a function reads non-volatile information from the activation record of

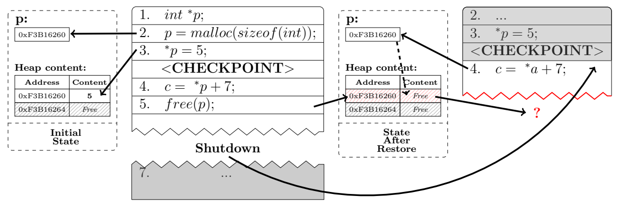

Figure 3. Example of memory map bug.

a function to be executed in the future. This bug may lead to wrong results or unwanted jumps.

Fig. 2 shows an example. A call to function f_1 executes first and its activation record is placed on the stack. A checkpoint takes place after line 2 inside f_1. When f_1 returns, its activation record pops from the stack and execution continues from line 7. The stack content, which is stored on NVM, is not deleted when returning from f_1; merely the stack pointer register changes. A call to f_2 executes next. When placing its activation record on the stack, the one of f_1 is overwritten. If a shutdown happens during the execution of f_2, the execution resumes inside f_1 according to the checkpoint data, but the activation record is that of f_2.

The consequences depend on a number of factors. Fig. 2 shows the case where the return address from f_2 is read as the one of f_1 when execution resumes. In the general case, the sequence of **`pop`** instruction belonging to the epilogue of f_1 may read the values produced by **`push`** instructions belonging to the prologue of f_2. This applies to saved registers, parameters, and local variables. Also, note that f_2 may equally be a programmer-defined interrupt handler that asynchronously fires at unpredictable times, making the issue even more difficult to track down.

We say an *activation record bug* exists whenever the stack is allocated on NVM and an ordered sequence of machine-code instructions $I_1, \ldots, I_n$ exists such that:

- I_1 is a **`call`** instruction for function f_x,
- the execution of f_x includes at least one checkpoint,
- I_n is a **`call`** instruction,
- no checkpoint exists in the sequence $I_1, \ldots, I_n$.

Note that the sequence $I_1, \ldots, I_n$ does not include the code of f_x. The bug exists because a checkpoint is saved inside the context of a function f_1, f_1 returns, and a subsequent call to f_2 may overwrite part or the whole activation record of f_1. For example, placing a checkpoint between the return of f_1 and the call to f_2 addresses the issue, as it prevents the execution from resuming inside f_1.

Ratchet [25] identifies a specific instance of the problem arising with interrupts. The general case above is overlooked in existing literature, and may be recognized only by reasoning at the level of machine code rather than source code.

2.3 Memory Map Bug

A *memory map bug* occurs when a dynamic memory operation observes a future state of memory due to heap allocations and deallocations on NVM.

Fig. 3 shows an example. Line 2 allocates a heap block and saves its address in pointer p. A checkpoint occurs before line 5 de-allocates the same memory block. If a shutdown happens after line 5, the execution resumes from line 4, whose memory access fails because the memory cell was de-allocated by the previous execution of line 5.

One may construct arbitrary combinations of heap operations before and after a checkpoint, leading to this kind of bug. If pointer information are not updated, the re-execution targets the memory address before the shutdown, whereas the re-allocated block is now somewhere else.

We say a *memory map bug* exists whenever the heap is allocated on NVM and an ordered sequence of machine-code instructions $I_1, \ldots, I_n$ exists such that:

1. I_1 is a **`load`** or **`store`** instruction targeting the heap block pointed by x,
2. I_n is a **`free`** or **`realloc`** instruction that modifies the heap block pointed by x,
3. no checkpoint exists in the sequence $I_1, \ldots, I_n$.

The bug exists because pointer information are not consistent with the state of the heap. Properly placing checkpoints to avoid re-executing instructions based on possibly inconsistent pointer information solves the issue.

Note how allocating the heap on NVM without a transactional memory controller [24] does not ensure atomicity for heap modifications. Power failures happening during the execution of any such instructions leave the heap state partially changed. Similarly, the re-execution of instructions that perform destructive changes to the heap, such as **`free`** or **`realloc`**, is a source of bugs, whereas re-executing memory allocation operations, such as **`malloc`**, does not affect correctness but may yield memory leaks.

Existing literature overlooks the existence of this kind of bugs too, which may only be recognized by reasoning at the level of machine code and raw memory accesses.

3 Hunting For Intermittence Bugs

Given certain program inputs, *exhaustively* testing intermittence bugs requires, in principle, to pretend a checkpoint occurs after every instruction and to check the aforementioned conditions for the execution of following *execution depth* instructions, that is, $n = ED$ in Sec. 2. This bears huge processing times as discussed before.

We seek to recognize the minimal amount of information necessary for the identification of intermittence bugs. Our techniques differ based on whether programmers are solely interested in *locating* bugs without appreciating their effects on program behavior, or rather wish to investigate how the occurrence of bugs *alters* the behavior. In the following, we use the case of data access bug as a running example and refer to an extended report for the other bugs [17].

Locating intermittence bugs. To determine where intermittence bugs are possibly placed, we execute the code while

Table 1. Relevant checkpoint and power failure locations.

	Data Access	Activation Record	Memory Map
Checkpoint before	`load`	`ret/pop`	`load`, `store`, `realloc`, `free`
Bug occurs after	`store`	`call/push`	`malloc`, `realloc`, `free`

tracking every operation in NVM and the execution of checkpoints. We find that the obtained execution trace suffices to *locate* intermittence bugs. Despite checkpoints might occur at any point in the execution, complete information for the identification of intermittence bugs only requires to test a subset of possible checkpoint locations.

In the case of data access bugs, for example, it is sufficient to verify situations where a checkpoint takes pace *right before* a **`load`** instruction. Since there, we check the execution trace for the following *ED* instructions looking for **`store`** instructions. If a power failure happens after executing a **`store`** on the same memory address, a data access bug may occur if the **`store`** writes a different value than the one **`load`**ed earlier. Cases where non-**`load`** instructions are executed multiple instructions after a checkpoint reduce the coverage since the **`load`**; thus, they cannot provide more information on intermittence bugs. Cases where a checkpoint occurs after a **`load`** do not meet the conditions in Sec. 2.1.

Tab. 1 summarizes how we determine the checkpoint locations and the instructions to look for within the following *ED* instructions to locate the bugs in Sec. 2.

Evaluating the effects. The processing above does not suffice to examine how a bug alters the program behavior, for example, by causing a crash. To that end, we need to emulate the re-execution. We use a per-instruction *logical clock* that we save/restore with checkpoints and a *lookup table* to keep track of events on the NVM. The latter serves to recognize how continuous and intermittent executions differ.

For data access bugs, for example, we emulate the execution since a checkpoint and until we encounter the first **`store`** within the next *ED* instructions. At this point, we restore the checkpoint state and re-execute the instructions until the **`store`**. The lookup table now contains the state of the (resumed) execution since the checkpoint, which is potentially different than in a continuous execution.

This processing may repeat for every **`store`** instruction within *ED* instructions since the checkpoint. This ensures that all bugs since the **`load`** are eventually recognized and their effect on program behavior emulated.

4 Early Results

We prototype a tool called `ScEpTIC` that implements the techniques in Sec. 3. `ScEpTIC` emulates the execution of LLVM intermediate-representation (IR) instructions. `SCEpTIC` takes as inputs what slices of the address space are allocated to NVM, the LLVM IR of the program, and the value for *ED*.

We conduct an early assessment of the techniques that also evaluate the effects of intermittence bugs. We select a set of common benchmarks in intermittent computing [1–3, 21]: CRC computation, FFT for signal analysis, and AES for encryption. The benchmarks are taken from MiBench2 [18], that is, MiBench [8] ported to IoT devices [25]. We determine that the MSP430 in Hibernus [2] executes from 1457 up to 4600 instructions after a checkpoint, depending on capacitor size. We thus use an average of $ED = 3000$ instructions.

Table 2. Comparison of processing times.

Benchmark	Instructions	Brute-force	ScEpTIC	Speedup
CRC	$5.2 \cdot 10^4$	$1.3 \cdot 10^9 s$	$451 s$	$2.88 \cdot 10^6$
FFT	$3.82 \cdot 10^5$	$5.16 \cdot 10^{11} s$	$2.31 \cdot 10^4 s$	$2.23 \cdot 10^7$
AES	$6.7 \cdot 10^5$	$2.74 \cdot 10^{12} s$	$7.05 \cdot 10^4 s$	$3.89 \cdot 10^7$

We compare `ScEpTIC` against a brute-force approach that examines all possible combinations of checkpoints and power failures. We consider a worst-case scenario for `ScEpTIC` by placing the entire address space on NVM, which yields the maximum number of checkpoints and power failures to test. We use input data provided with the MiBench2 suite [18]. As the time required by the brute-force approach is not practical, we analytically calculate it based on the maximum number of instructions `ScEpTIC` can emulate per unit of time.

Tab. 2 shows our analytical calculations against the measures obtained by running `ScEpTIC` while employing the techniques of Sec. 3. The minimum speedup is $2.88 \cdot 10^6$ and it grows with the number of instructions. These results demonstrate the effectiveness of our techniques, which grant a significant speedup over a brute-force approach.

5 Ongoing Work and Outlook

In addition to a full-fledged evaluation, our ongoing work includes an analysis of two additional un-explored facets.

One facet deals with *environment interactions*. Re-executing operations that read from the environment through sensors or affect it through actuators may lead to unexpected behaviors or unintended external effects. We understand some of the latter may be unavoidable, as output actions may not be undone. Our aim is to act cautiously, that is, understand the conditions in the execution flow that lead to these situations, develop techniques to identify these conditions efficiently, and give programmers hints to defend.

There also exists the possibility that intermittence bugs are *intentionally left occurring* to make programs aware of intermittence. The data access bug in Fig. 1, for example, may implement a counter of the number of power failures. The problem here is flipped: programmers must know whether their intermittence-aware programs do behave as expected. We seek to understand where and how programmers rely on this feature and to conceive techniques to verify their correspondence to the programmers' intentions.

As the techniques we develop are progressively integrated in `ScEpTIC`, we increase the reliability of intermittent programs and therefore the dependability of resource-constrained embedded systems powered off energy harvesting, ultimately providing a solid basis for the upcoming battery-less IoT.

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 35, 12 (2016), 1968–1980. https://doi.org/10.1109/TCAD.2016.2547919

[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 7, 1 (March 2015), 15–18. https://doi.org/10.1109/LES.2014.2371494

[3] Naveed Anwar Bhatti and Luca 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 '17). ACM, New York, NY, USA, 209–219. https://doi.org/10.1145/3055031.3055082

[4] Alexei Colin, Graham Harvey, Brandon Lucia, and Alanson P. Sample. 2016. An Energy-interference-free Hardware-Software Debugger for Intermittent Energy-harvesting Systems. SIGOPS Oper. Syst. Rev. 50, 2 (March 2016), 577–589. https://doi.org/10.1145/2954680.2872409

[5] Alexei Colin and Brandon Lucia. 2016. Chain: Tasks and Channels for Reliable Intermittent Programs. SIGPLAN Not. 51, 10 (Oct. 2016), 514–530. https://doi.org/10.1145/3022671.2983995

[6] Joakim Eriksson, Adam Dunkels, Niclas Finne, Fredrik Österlind, and Thiemo Voigt. 2007. MSPsim - an Extensible Simulator for MSP430-equipped Sensor Boards. In Proceedings of the European Conference on Wireless Sensor Networks (EWSN), Poster/Demo session.

[7] Matthew Furlong, Josiah Hester, Kevin Storer, and Jacob 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). ACM, New York, NY, USA, 23–26. https://doi.org/10.1145/2996884.2996889

[8] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. 2001. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop (WWC '01). IEEE Computer Society, Washington, DC, USA, 3–14. https://doi.org/10.1109/WWC.2001.15

[9] Josiah Hester, Timothy Scott, and Jacob 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). ACM, New York, NY, USA, 1–15. https://doi.org/10.1145/2668332.2668336

[10] Josiah Hester and Jacob Sorber. 2017. The Future of Sensing is Batteryless, Intermittent, and Awesome. In Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems (SenSys '17). ACM, New York, NY, USA, Article 21, 6 pages. https://doi.org/10.1145/3131672.3131699

[11] Texas Instruments. [n. d.]. MSP430FRxxxx datasheet. http://www.ti.com/lit/ds/symlink/msp430fr5737.pdf.

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

[13] Y. Lee, G. Kim, S. Bang, Y. Kim, I. Lee, P. Dutta, D. Sylvester, and D. Blaauw. 2012. A modular 1mm3die-stacked sensing platform with optical communication and multi-modal energy harvesting. In 2012 IEEE International Solid-State Circuits Conference. 402–404. https://doi.org/10.1109/ISSCC.2012.6177065

[14] Brandon Lucia and Benjamin Ransford. 2015. A Simpler, Safer Programming and Execution Model for Intermittent Systems. SIGPLAN Not. 50, 6 (June 2015), 575–585. https://doi.org/10.1145/2813885.2737978

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

[16] Kiwan Maeng and Brandon Lucia. 2018. Adaptive dynamic checkpointing for safe efficient intermittent computing. In 13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18).

[17] Andrea Maioli. 2019. Understanding and Testing Intermittence Bugs in Transiently-powered Computers. Technical Report n 37/2019. Politecnico di Milano (Italy).

[18] mibench2 [n. d.]. MiBench2 porting to IoT devices. https://github.com/impedimentToProgress/MiBench2.

[19] Proteus 2015. Proteus Digital Health. https://www.proteus.com/.

[20] Benjamin Ransford and Brandon Lucia. 2014. Nonvolatile Memory is a Broken Time Machine. In Proceedings of the Workshop on Memory Systems Performance and Correctness (MSPC '14). ACM, New York, NY, USA, Article 5, 3 pages. https://doi.org/10.1145/2618128.2618136

[21] Benjamin Ransford, Jacob Sorber, and Kevin Fu. 2011. Mementos: System Support for Long-running Computation on RFID-scale Devices. SIGARCH Comput. Archit. News 39, 1 (March 2011), 159–170. https://doi.org/10.1145/1961295.1950386

[22] E. Sardini and M. Serpelloni. 2011. Self-Powered Wireless Sensor for Air Temperature and Velocity Measurements With Energy Harvesting Capability. IEEE Transactions on Instrumentation and Measurement 60, 5 (May 2011), 1838–1844. https://doi.org/10.1109/TIM.2010.2089090

[23] E. Sazonov, H. Li, D. Curry, and P. Pillay. 2009. Self-Powered Sensors for Monitoring of Highway Bridges. IEEE Sensors Journal 9, 11 (Nov 2009), 1422–1429. https://doi.org/10.1109/JSEN.2009.2019333

[24] Michal Spivak and Sivan Toledo. 2006. Storing a Persistent Transactional Object Heap on Flash Memory. In Proceedings of the ACM Conference on Language, Compilers, and Tool Support for Embedded Systems (LCTES).

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

[26] K. Vijayaraghavan and R. Rajamani. 2010. Novel Batteryless Wireless Sensor for Traffic-Flow Measurement. IEEE Transactions on Vehicular Technology 59, 7 (Sep. 2010), 3249–3260. https://doi.org/10.1109/TVT.2010.2050013