I/O Branch Bugs on Intermittent Systems

Milijana Surbatovich Emily Ruppel

1 Introduction

Batteryless, energy harvesting devices are positioned to deliver an exciting new class of applications enabled by maintenance free sensing and computing. Batteryless devices harvest energy from their environments and store it in a small buffer such as a capacitor rather than charging a battery. The buffered energy is used to operate a microprocessor as well as an array of peripherals that gather and communicate data. As a result, these devices operate only intermittently as energy is available. Once all of the energy in the buffer has been used, the device powers down and execution is terminated. Multiple systems have been developed to support long running computations by continuing the execution across reboots. The intermittent runtime systems developed in prior work use careful book-keeping of program state to correctly perform the continuations without custom hardware [1, 3, 5, 4, 2, 6].

When an execution is interrupted by a power failure, intermittent runtime systems restore the most recent consistent state from a checkpoint [5, 6], or task boundary [1, 4, 3] and re-execute the segment of code that was interrupted. All of these systems allow intermittently executed programs to make progress and prevent persistent state from being corrupted by re-execution, however, their guarantees in terms of sensor input are limited. Specifically, if *control flow* decisions are made based on non-idempotent operations, such as reading input from a sensor, these systems allow updates from multiple branches to persist. This is a problem because re-executing the code and taking different branches on each execution may result in writes to different variables, leaving the previously written memory in an inconsistent state. Bugs that can result from the partially written state are difficult for programmers to diagnose because they require the programmer to reason about how partial execution of different branches of control flow will affect the future state of the program.

Instead, our project uses a compiler pass to statically identify writes to non-volatile memory that may leave memory in an inconsistent state if repeated reads from the sensor produce different values. The pass first detects data dependences between variables in the program and I/O operations. Variables with data dependences on I/O are considered tainted. Then the pass detects control flow that is dependent on tainted values. The pass searches for writes to non-volatile memory in each of the basic blocks along the different control flow paths. After filtering the writes to reduce the number of false positives, the pass reports each of the remaining writes as potential bugs.

2 Background

Prior work in intermittent systems seeks to instrument the minimum number of writes to non-volatile memory to reduce the overhead of the intermittent runtime. In doing so, the runtimes only instrument updates to non-volatile memory that are involved in write-after-read (WAR) dependences. However, ensuring the consistency of memory involved in WAR dependences still leaves the program open to bugs caused by non-idempotent behavior.

2.1 Existing Intermittent System Support

Before discussing the range of non-idempotent operations that can occur on an intermittent device, we will present an example of why prior work on intermittent runtime systems fails to address the problem of non-idempotent control flow. Figure 1 shows a section of code instrumented with `dino_task` boundaries [3]. DINO uses task boundaries to form (supposedly) idempotent tasks. At each boundary, the volatile state of the program is checkpointed in non-volatile memory, and each non-volatile variable that is involved in a WAR dependence before the next task boundary is written to the `dino_nv_buffer`. The buffer of non-volatile variables ensures that a power failure does not cause the value of the WAR variables to become inconsistent.

In the example, the `dino_nv_buffer` is empty because no variable has a WAR dependence. As a result, updates to `steady` and `blink` are written directly to persistent memory. On the first execution attempt, the `get_temp` call returns a value that is less than the threshold, so `steady` is set before a power failure occurs. On reboot, the sensor returns a value that is greater than the

threshold, so `blink` is also set. After falling out of the if-else statement, the assertion that both variables cannot be set fails and the program terminates with an error.

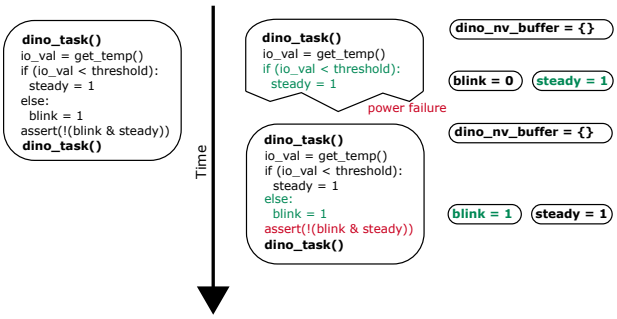

Figure 1: Attempt to execute the code shown at left using the DINO runtime. The state of the DINO buffer and the non-volatile variables, blink and steady are shown to the right.

Any runtime system that relies strictly on WAR analysis to preserve idempotent execution [3, 4, 6] will experience the same failure mode because the updates made along uncommitted branches will be allowed to persist.

2.2 A discussion of non-idempotent behavior in intermittent programs

In Figure 2, we present a flow chart of the taxonomy of non-idempotent behaviors and their potentially buggy consequences.

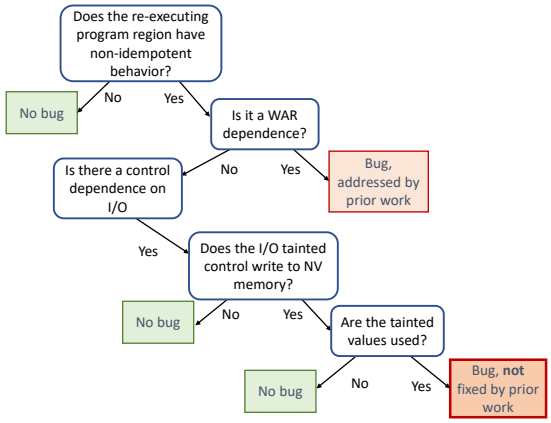

Figure 2: Taxonomy of non-idempotent behavior and consequences

Question 1: Does the program have non-idempotent behavior? Idempotent code can be re-executed arbitrarily without any changes to the visible output [6, 3]. Most new bugs introduced by computing under intermittent power are due to memory inconsistencies caused by the unexpected re-execution of code regions. If a program does not exhibit non-idempotent behavior, it will not have any of the bugs we are studying.

Questions 2 and 3: What is the cause of the non-idempotency? Non-idempotent behavior generally has two sources: write-after-read (anti) dependencies and I/O operations.[1] Code re-execution after a WAR dependence creates direct data-flow that did not exist in the original program, and I/O creates effects that cannot be undone. Bugs off of WAR dependencies have been extensively studied in prior work, and several runtime systems exist to fix them [6, 4, 3]. While there could be other bugs stemming from the use of irrevocable I/O operations, this paper specifically explores memory inconsistencies stemming from branches that are control dependent on non-idempotent input.

Questions 4 and 5: When can an I/O dependent branch cause a bug? Not all I/O dependent branches necessarily lead to buggy behavior. To make memory be in an inconsistent state, non-volatile variables must be written to on at least one side of the branch, as they will still be visible to the program on reboot. Furthermore, on a power fail and re-execution of the non-idempotent region, such a tainted variable has to be the reaching definition for some use, either in expected program execution or along the back-edge introduced by re-executing. We discuss this further in Section [false positives]. If the targets of the I/O dependent branch do write to non-volatile memory, however, and there is some use of these tainted variables, this can cause bugs that have not been addressed by prior work.

3 Evaluation

4 Implications

5 Surprises and Lessons Learned

6 Conclusion

References

[1] A. Colin and B. Lucia. Chain: Tasks and channels for reliable intermittent programs. In *Proceedings*

[1] Besides WAR dependencies and I/O, there are some less common causes of non-idempotency, such as pointer-based stack operations, as noted in prior work by Van Der Woude et al. [6].

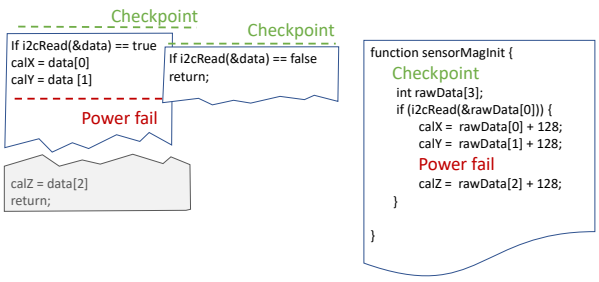

Figure 3: Bug in magnetometer initialization. Power failing while updating the calibration fields can cause the calibration data to become inconsistent, corrupting any future magnetometer reads that use the calibration.

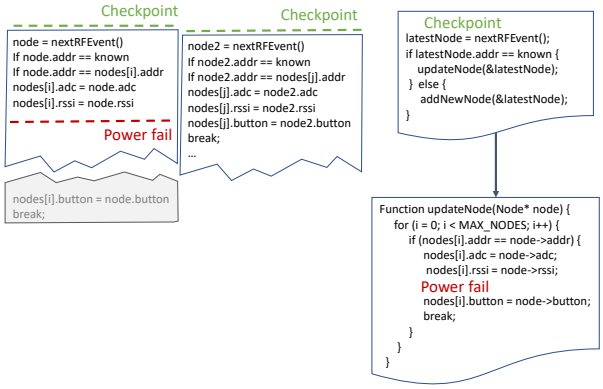

Figure 4: Bug in WSN concentrator. Power failing while updating the node structure can cause the node list to become inconsistent, corrupting the payload ("button" field) or using out-of-date timing information

of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA), 2016.

[2] J. Hester, K. Storer, and J. Sorber. Timely execution on intermittently powered batteryless sensors. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*. ACM, 2017.

[3] B. Lucia and B. Ransford. 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 2015, pages 575–585, New York, NY, USA, 2015. ACM.

[4] K. Maeng, A. Colin, and B. Lucia. Alpaca: Intermittent execution without checkpoints. *Proc. ACM*

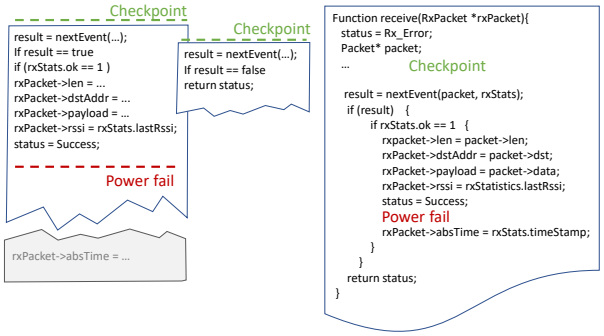

Figure 5: Bug in RF EasyLink Receiver. Power failing before returning a correctly read packet can cause the status field to be set to success, even if the next event fails. The function will return an incorrect status value, potentially crashing the larger application.

Program. Lang., 1(OOPSLA):96:1–96:30, Oct. 2017.

[5] B. Ransford, J. Sorber, and K. Fu. Mementos: System support for long-running computation on RFID-scale devices. In *ASPLOS*, Mar. 2011.

[6] J. Van Der Woude and M. Hicks. Intermittent computation without hardware support or programmer intervention. In *Proceedings of OSDI'16: 12th USENIX Symposium on Operating Systems Design and Implementation*, page 17, 2016.