I/O Branch Bugs on Intermittent Systems

Milijana Surbatovich Emily Ruppel

Abstract

Batteryless energy harvesting devices are computing platforms that operate in environments where batteries are not viable for energy storage. Energy-harvesting devices operate intermittently, only as energy is available. Prior work developed strategies for continuing long-running computations across reboot cycles on energy harvesting devices. The strategies use a combination of checkpointing volatile state and versioning non-volatile state to make application code arbitrarily re-executable. However, prior work provides weak guarantees for *non-idempotent* operations such as reading from a sensor. In the presence of control flow that is dependent on non-idempotent operations, previous runtime systems allow updates from uncommitted paths through the code to persist. In this paper, we present a compiler pass that statically detects potentially inconsistent writes to memory caused by non-idempotent operations and warns the programmer. We use the pass to analyze suites of embedded device code and found bugs caused by inconsistent writes.

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, 5, 7, 6, 4, 9].

When an execution is interrupted by a power failure, intermittent runtime systems restore the most recent consistent state from a checkpoint [7, 9], or task boundary [1, 6, 5] 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 [5]. 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.

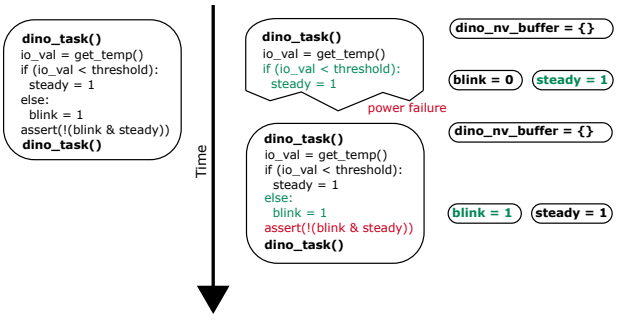

Figure 1: Attempt to execute the code shown at left using the DINO runtime. The state of the DINO buffer and the non-volatile variables, blink and steady are shown to the right.

Any runtime system that relies strictly on WAR analysis to preserve idempotent execution [5, 6, 9] 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.

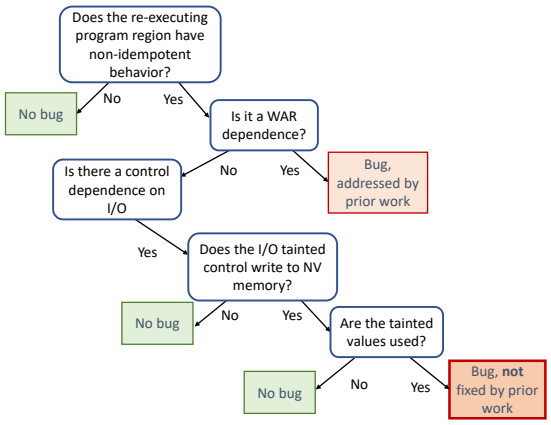

Figure 2: Taxonomy of non-idempotent behavior and consequences

Question 1: Does the program have non-idempotent behavior? Idempotent code can be re-executed arbitrarily without any changes to the visible output [9, 5]. 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 [9, 6, 5]. 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

[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. [9].

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 System Overview

The overarching algorithm is fixed-point - iterating through the functions, summarizing new information about the I/O flow, and iterating through functions again, until there is no new information to be gleaned. Psuedocode is presented in Algorithm 1

When the pass starts iterating, it doesn't know where the I/O enters the program, so all the functions are put in the working list, with a NULL I/O parameter–the I/O call is found internally, based on annotated information in the source code. Once the I/O source is found, the pass calls a function that traverses the data flow chain of the I/O instruction within the local function. In the chain, there are some instructions of interest: branches, which can lead to the bug of mismatched non-volatile write sets, and instructions that result in I/O tainted values flowing out of the current function that we're examining (an interprocedural flow **source**).

Traversing the data flow primarily uses LLVM's built-in def-use chains. From the definition of the tainted variable, we follow the uses in a depth first search, adding on any new store instructions to the visit stack. From the traversal function call, we return the list of interesting instructions found to $findTainted$. If the instruction was a branch, the pass calls the $check$ function to examine the non-volatile write sets of the branch target blocks to see if there is the possibility for the bug. Additionally, any variables written to on the tainted branch are control-dependent on the I/O variable, so the pass adds any globals to the interprocedural source list and continues the local traversal with the newly tainted control dependent variables.

Once the pass finishes traversing the tainted flow within the local function, it returns back to $searchFunction$ with the list of tainted sources of interprocedural flow.

Values can leave the local function by:

- being returned from the function
- being passed into a global variable
- being passed as a parameter into another function
- being stored into a call-by-reference parameter (which may be global itself

Algorithm 1 Iterative, fixed point taint tracking. While the algorithm is not truly interproeedural, it tracks the data and control dependencies with the local function and summarizes how the taint flows out of the function. The summary information is used to add functions to the working list, until the entire I/O tainted flow has been explored

```
function SEARCHFUNCTIONS(ModuleM)
    for all Function func in M do
        workinglist ← pair(func, NULL)
    end for
    while !workinglist.empty do
        currPair ← workinglist.next
        sources ← FINDTAINTED(currPair)
        sinks ← SUMMARIZE(sources)
        for all taintedPair in sinks do
            workinglist.add(taintedPair)
        end for
    end while
end function

function FINDTAINTED(func, ioVal)
    tainted ← SEARCHANNOTATION(func)
    visitlist ← tainted
    while !visitlist.empty do
        current ← visitlist.next
        result ← TRAVERSEDEPENDENCE(current)
        for all Instructions in result do
            if instruction == BranchInst then
                CHECK(instruction, visitlist)
            else
                Sources.add(instruction)
            end if
        end for
    end while
    return Sources
end function

function TRAVERSEDEPENDENCE(tainted)
    for all data dependencies of tainted do
        if instruction == (BranchInst or Source) then
            result.add(instruction)
        end if
    end for
    return result
end function

function CHECK(branch, visitlist)
    for all conditional target blocks on branch do
        for all Instructions in block do
            if instruction == store then
                bugReport[branch].add(instruction)
                visitList.add(instruction)
            end if
        end for
    end for
end function
```

If the tainted value was returned from the function F, we add all the callers of F to the stack, with the value where the return was stored as the tainted I/O value. (When adding new functions to the examination stack based on summary information, we pass in the tainted value as a parameter instead of searching for the annotations, but then search the chain in the same way.)

Likewise if the tainted value was passed into a global, we find the uses of the global, the parent Function of those uses, and add those to the stack, with the global as the tainted value. If the I/O was used as a function parameter, we find the callee function and the local version of the argument.

If I/O was stored into a pass-by-reference parameter, we find all the caller functions and their local value that the argument aliases to. (If the reference para refers to a global, we also add that global to the summary information.)

We add all these new function/tainted value pairs to the stack, and continue to iterate. (We do keep track of what pairs we have already examined, to prevent infinite looping). Eventually, there will be no new information being returned from the traversal function, and we have examined all the functions to which the tainted I/O has spread.

3.1 NV Write-set algorithm

To help with handling arbitrary complexity of control flow within the branches off of io, we precompute the may write sets of each function before hand, using a bottom up examination of the call graph.

What stores we decide to add the write sets does effect what possible bugs will be reported. For instance, one complexity that must be considered is how to handle stores on branches. Say we have a simple program with two nonvolatile variables, nv0 and nv1 that branches on an I/O tainted variable. On one side of the I/O branch we write to nv0 and nv1. On the other side, we write to nv0, but then we branch again. On one side of the branch we write to nv1, but on the other side we don't. Thus, in this example, one execution could have the I/O branch writing to the same nv set on both sides, and another execution doesn't.

There are a few ways to deal with these branches: don't add any stores within the branch basic blocks to the write set, add all the stores to the write set (what we do right now), or make two versions of the write set, adding a different side of the branch to each one. This last option is perhaps most attractive for accuracy, but it is difficult to handle arbitrary complexity, and has a much higher storage overhead.

Another problem with the first two approaches is that if we adjust the above example a little, we can have an I/O branch where both sides write to nv0, and on the second branch, one side also writes to nv1. In this case, adding all the stores with the branches detects more bug possibilities, whereas ignoring the branch under reports. So neither of the simpler solutions can be said to definitively under or over report bugs.

3.2 Filtering Pass

//TODO

3.3 High-level, English explanation of what the tool detects

- The final bug is always the disparate nv write sets after a branch off of an I/O tainted value
- We track the I/O through explicit (data) and implicit (control) flow
- Tainted I/O can leave a function through function calls, return values, pass by ref, and globals
- Initial I/O into the prog has to be annotated (not good or bad, just a thing)

3.4 Possible limitations of the tool

- Precomputed write sets
- Overhead?
- How complex can the src code get? – we should prob impose a limitation on branching levels or function depth
- Are there any other suspicious bug patterns? i.e., completeness
- Over or under report bugs? Right now tending towards conservatism

4 Evaluation

To evaluate the pass and demonstrate the presence of I/O dependent branch bugs in real code, we ran the pass on several suites of code for embedded devices. The three suites were MiBench [3], TI-RTOS driver code [8], and modified benchmarks written by the ABSTRACT research group for testing intermittent runtime systems on energy harvesting devices [6, 1, 5, 2]. We found that the

presence of the bug varies depending on how the code processes I/O and whether the developers were aware of the intermittent execution model.

To run the pass, we had to first ensure that the source code could be compiled with clang so that LLVM-IR could be emitted. To run MiBench and TI-RTOS code we had to change [insert stuff changed for MiBench and tirtos here]. The ABSTRACT benchmarks were all written to target MSP430 microcontrollers using TI's proprietary MSPGCC compiler, so MSP430 specific macros had to be removed from the benchmark code and the linked runtime. Next, we registered the I/O functions in each of the benchmarks so that the pass could track their updates. At that point the benchmarks are compiled with clang so that debug information is included in the LLVM-IR that is produced and handed to the pass described in Section 3. Then we analyzed any bug reports produced by the pass to determine the severity of the problem.

5 Implications

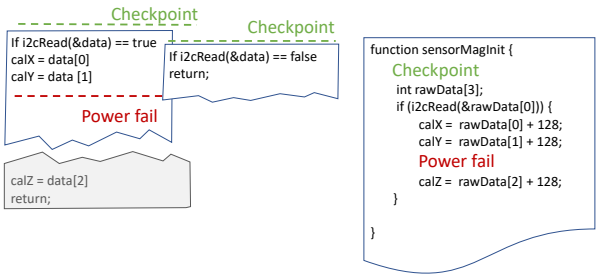

Figure 3: Bug in magnetometer initialization. Power failing while updating the calibration fields can cause the calibration data to become inconsistent, corrupting any future magnetometer reads that use the calibration.

6 Surprises and Lessons Learned

7 Conclusion

References

[1] A. Colin and B. Lucia. Chain: Tasks and channels for reliable intermittent programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2016.

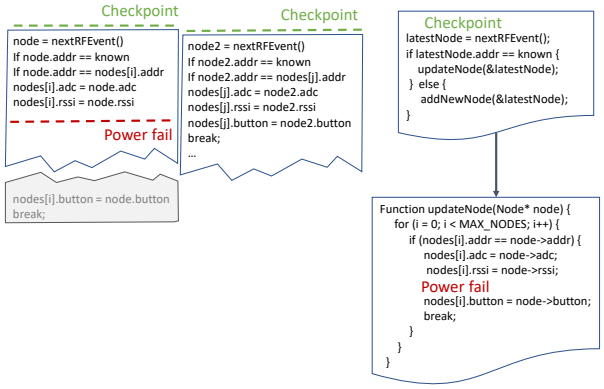

Figure 4: Bug in WSN concentrator. Power failing while updating the node structure can cause the node list to become inconsistent, corrupting the payload ("button" field) or using out-of-date timing information

[2] A. Colin, E. Ruppel, and B. Lucia. A reconfigurable energy storage architecture for energy-harvesting devices. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 767–781, New York, NY, USA, 2018. ACM.

[3] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pages 3–14. IEEE, 2001.

[4] 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.

[5] 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.

[6] K. Maeng, A. Colin, and B. Lucia. Alpaca: Intermittent execution without checkpoints. *Proc. ACM Program. Lang.*, 1(OOPSLA):96:1–96:30, Oct. 2017.

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

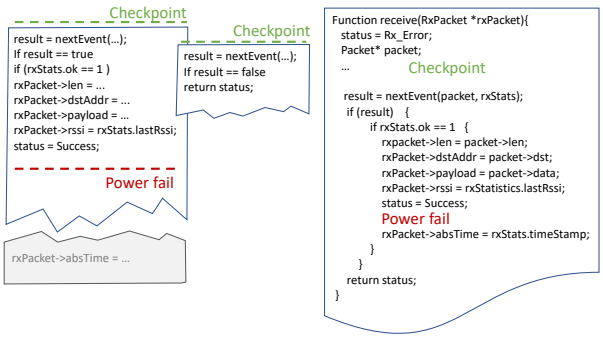

Figure 5: Bug in RF EasyLink Receiver. Power failing before returning a correctly read packet can cause the status field to be set to success, even if the next event fails. The function will return an incorrect status value, potentially crashing the larger application.

[8] TI Inc. Ti-rtos: Real-time operating system (rtos) for microcontrollers (mcu), 2017. Accessed: 2018-05-08.

[9] 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.