Mitigating Single-Event Functional Interrupts in the Peripheral Circuitry of Non-Volatile Memories in Radiation Environments

Daniel Puckett*
Sandia National Laboratories
1515 Eubank SE
Albuquerque, NM 87123
dpucke@sandia.gov

Henri Malahieude*
Sandia National Laboratories
1515 Eubank SE
Albuquerque, NM 87123
hvmalah@sandia.gov

Joseph D'Amico
Sandia National Laboratories
1515 Eubank SE
Albuquerque, NM 87123
jvdamic@sandia.gov

Josh Joffrion
Sandia National Laboratories
1515 Eubank SE
Albuquerque, NM 87123
jbjoffr@sandia.gov

***Abstract*—**Non-Volatile Memories (NVMs) are essential to computing systems, including systems present in radiation environments. Previous radiation tests on NVMs have revealed failure cases where the peripheral circuitry of an NVM is affected such that the NVM is inaccessible but no data is corrupted. We experienced this failure case, which we call lock-up, on an NVM we tested. Lock-up occurs for many microseconds at a time, which can lead to several kilobytes of data corruption. In this paper, we present a two-pronged mitigation strategy for lock-up to improve NVMs' reliability and availability and evaluate our mitigation strategy through simulations. First, we address NVMs' reliability by designing and evaluating several detection policies, which can detect when lock-up begins and ends consistently and with low overhead. Second, we address NVMs' availability by designing the Cache Window algorithm, which pre-loads data into a radiation hardened cache, enabling the processor to make forward progress while the NVM is in lock-up.

TABLE OF CONTENTS

1. INTRODUCTION

As mission goals for aerospace contracts have grown larger with the industry's maturity, so has the need to bring more complex computing capabilities to higher elevations. From airspace [1], to low-Earth orbit (LEO) [2], to geosynchronous orbit (GEO) [3], the aerospace industry faces a wide range missions and customers to service. Yet pervasive to the entire domain is the threat of radiation effects and their potentially catastrophic consequences [4].

While off-loading computation from the primary device to terrestrial assets is possible, at the very least a computer is necessary to send and receive data. Since missions require computers in radiation environments, it is important that these systems do not fail under radiation.

One essential component of computers is Non-Volatile Memory (NVM), which is composed of memory cells that store data and peripheral circuitry that controls access to the data. Traditional flash NVMs use charge-based memory cells; however, these memory cells are vulnerable to bit flips due to radiation. The memory cells of MRAMs, an emerging NVM based on magnetism, are naturally resilient to radiation. Thus, MRAMs are commonly used in aerospace applications that require high reliability like bootloaders [5], [6] and storing program data [7], [8].

Unfortunately, the peripheral circuitry of MRAMs is not naturally resilient to radiation because it is still fabricated with traditional CMOS. We have found this can lead to single-event functional interrupts (SEFIs) on MRAMs. We performed heavy ion radiation testing with an effective LET of 10.7 MeV cm^2/mg using the Texas A&M Cyclotron on a commercial MRAM. The MRAM experienced SEFIs where writes had no effect and reads returned incorrect values. After a short time, the MRAM returned to full functionality without a reset or power cycle. No data was corrupted during the SEFIs. We call this type of SEFI "lock-up".

If lock-up occurs undetected, then the MRAM's *reliability* (ability to read and write uncorrupted data) is reduced because it could operate on incorrect data or fail to store critical data. However, if the lock-up is detected and computation halted until lock-up ends, then the MRAM's *availability* (ability to respond to requests) is reduced.

Error correction codes have been used to protect data in radiation environments [11], [12]. However, they are insufficient to recover the several kilobytes of data that may be incorrectly read or written during lock-up.

Single event effects similar to lock-up have been documented on ReRAMs [9], [10] and possibly other MRAMs [10],

979-8-3315-7360-7/26/$31.00 ©2026 IEEE
* denotes equal contribution

although a reset or power cycle was required to return device functionality. This leads us to believe that lock-up may become a widespread phenomenon on non-traditional NVMs. To improve the reliability *and* availability of non-traditional NVMs under lock-up, we contribute the following:

- A model of lock-up on NVMs
- Policies to detect when lock-up begins and ends
- The *Cache Window* algorithm, which pre-loads data into a radiation-hardened cache, enabling the processor to make forward progress when the NVM is in lock-up
- Evaluations of the effectiveness and overhead of our detection policies and the Cache Window algorithm

2. NVM MODEL

We model the NVM based on the specifications of emerging NVMs [13], [14], [15]. Our NVM model has a single port and can only perform one read or write at a time. Like many NVMs, our NVM model has a read and write buffer known as the *page* and can be operated with or without it. Our NVM's page is 16 bytes, and each access to it is 4 bytes, allowing for a total of four accesses from the page. The four positions that can be accessed in the page are referred to as *page indexes*, and a block of memory will always be mapped to the same page index. In page mode, the first access outside of the currently open page will replace the page with a new page. It takes 30 nanoseconds to load the new values into the page and perform the requested operation and 10 nanoseconds to access a currently open page. In non-page mode, each access takes 30 nanoseconds. While we evaluate how well our detection policies detect lock-up on both page mode and non-page mode, we only evaluate overhead on page mode.

When the NVM is in lock-up, we have observed two different read behaviors. In one behavior, called "all zeros", the NVM always returns 0, but in the other behavior, called "open page", the NVM returns data that was in the page when lock-up began. We have not observed any affect from writes while the NVM is in lock-up.

In our NVM model, the duration of each lock-up is selected from a Gaussian distribution, at 75 microseconds on average with a standard deviation of 10 microseconds. The time between each lock-up is selected from a Poisson distribution, at 100 microseconds on average. Each distribution is generated with a separate permuted congruential generator for random numbers [16] which is included in Python's *default_rng* [17].

3. DETECTION POLICIES

In this section, we introduce several detection policies used to identify when lock-up periods begin and end, then briefly discuss their strengths and weaknesses. In these detection policies, we assume that the time between consecutive accesses is significantly shorter than the length of the lock-up period. We base this assumption on the duration of lock-up found in our radiation tests. If the assumption does not hold due to long delays between consecutive accesses, then programmers must manually launch one of the checks described below before the access. We also assume the NVM memory cells and CPU are rad-hard and only the NVM's peripheral circuitry is vulnerable.

Write-Verify Detection Policy

We first introduce a naive detection policy, *Write-Verify*, which detects lock-up by checking if the NVM is acting consistently. We start with this policy because it is similar to existing write-verify policies [18]. To check if a write occurred during lock-up, Write-Verify reads the same address immediately after writing to it, then verifies that the read value is equal to the written value.

Write-Verify has very low overhead because it only requires a single access, and that access will never open a new page. However, Write-Verify can fail to detect write accesses corrupted by lock-up when the written value is the same as the value returned during lock-up (e.g., writing a zero when the lock-up behavior is all zeros).

Canary Detection Policies

Since Write-Verify can fail to detect lock-up, we introduce another group of detection policies called *Canary*[2]. A Canary policy, denoted *Canary-N*, stores N different canary values at N addresses (each address in a different page of the NVM), then checks for lock-up by reading from each address and checking if it is the expected value. These addresses must be off-limits for applications, or the application could accidentally make Canary believe lock-up is occurring when it is not. Additionally, the Canary addresses and values must be stored in rad-hard memory. The intuition behind Canary is to test if the NVM can open a new page and correctly read the data from it – which cannot happen while the NVM is in lock-up.

Since this method is independent of the original access type, this policy detects lock-up just as effectively for reads as for writes. However, this class of policies has higher overhead than Write-Verify because Canary-N must access N different pages. Additionally, Canary-1 is vulnerable when lock-up has the open page behavior because the Canary address can be locked into the open page when lock-up begins.

Monitor Detection Policies

Since the NVM returns constant values to reads while it is in lock-up, we introduce a lightweight class of detection policies, *Monitor*, specifically for read accesses. A Monitor policy, denoted *Monitor-N*, silently observes a number of consecutive reads; if any of the reads give different values, then the first read could not have been in lock-up. If all of the reads give the same value, then Monitor-N executes a Canary-N check to verify the NVM is in lock-up. This verification is necessary because the application may naturally read a series of identical values from different addresses – consider highly sparse applications.

Monitor-N has an additional parameter, the *similar access threshold*, which is the number of consecutive reads with the same value before it executes a Canary-N check. In Sec. 5, we explore the optimal similar access threshold based on the level of sparsity in the application.

In general, Monitor-N has less overhead than Canary-N but similar effectiveness to Canary-N.

Conditional Detection Policies

Since the NVM frequently returns "0" in lock-up, then *Conditional-N*, which is only applicable to writes, uses

[2]Named after the phrase "Canary in a coal mine".

Canary-N to check for lock-up after writing "0" and Write-Verify after writing any other value.

This class of policies has a smaller set of failure cases than Write-Verify and has less overhead than Canary.

Varying check interval

Write-Verify, Canary-N, and Conditional-N all have a parameter called "check interval", which is the number of accesses between each check for lock-up. Increasing the check interval lowers the overhead of each policy, but increases the amount of re-work that must be done when lock-up is detected. We examine this tradeoff in Sec. 5.

Varying polling delay

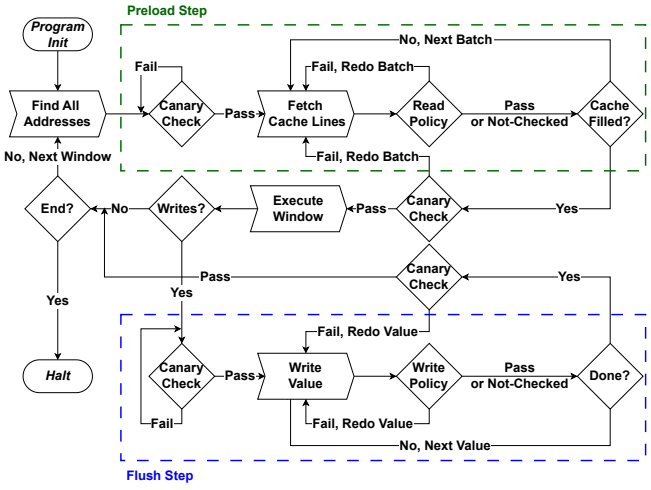
When a policy detects lock-up, it polls the NVM to check if lock-up has ended. The *polling delay* is the amount of time the policy waits between each check. Increasing the polling delay decreases the power consumption of the NVM but may result in a period of time where the CPU is unnecessarily idle, increasing latency. We examine this tradeoff in Sec. 5. Polling delays are manually defined constants, but future work could involve an exponential backoff on the delay.

Summary

Since we discuss many detection policies, we summarize each of them in Tbl. 1. Additionally, we summarize each configurable parameter for the detection policies in Tbl. 2.

4. Cache Window Algorithm

Figure 1. Flow Chart describing how a program using the *Cache Window* algorithm functions. A program is divided into *Windows* which preloads the needed data (and regions to write) into the cache on entrance, and flushes the data on exit. Note how the *Window*, the section of code, is only executed when all memory accesses are verified to only communicate to the cache.



We take from previous Failure-Atomic literature [19], [20], [21] and apply Failure-Atomic regions in reverse to our failure mode: the processor and its internal memory structures survive the failure and the memory is resumed. Using the cache, we can prepare the needed data regions before each section of code we call a *Window*. Once the cache has been fully prefilled and verified by the read policy, the processor can enter the window without worrying about not having the data to operate on. Furthermore, this allows a processor to operate through lock-up periods as if the NVM was still active. Not only is the cache able to hide the latency of accessing the external device, it can also hide temporary failures.

To verify the success of accesses before and after the window, the check interval of the detection policy is used to "batch" operations to the NVM and a canary acts as a "lock-up barrier" on the access. The write process is exactly like the read process, it begins with a canary read to prevent initial operations from happening during a lock-up, then each value is passed through the read or write policy to be verified (or not due to check interval alignment), and finally another canary read is used to validate the remaining values not yet checked. As long as the total time it takes to access the memory region is shorter than the average lock-up length, the read and write policy can have a relatively large check interval if fenced with canary reads to verify the access that do not fall within check intervals. Accesses that do not fall within a check interval will be the beginning and end of the sequence of accesses.

Constraints

The most obvious constraint on this algorithm is that code cannot be stored on the NVM. Lock-up sensitive devices cannot reliably provide data at all times, and unless there are architectural changes done to a processor (out of scope for this paper) there may be cases of incorrect machine code execution. Instruction prefetching can mitigate this challenge but is difficult to implement without operating system and hardware support. In environments without instruction prefetching, code should be placed on a more reliable memory device even if the Cache Window algorithm is not used.

This also means that the processor cannot be sensitive to radiation events. Should the cache, either instruction or data, be corrupted by a radiation event the correctness of the program is no longer maintained. The processor, the cache, and the device where the instructions are stored must be radiation-insensitive.

Lastly, the cache's size limits the possible division locations of a program. The more cache a processor has, the larger the windows can be and the more effective the algorithm. Conversely, the less cache a processor has the more often the algorithm will need to prefetch and flush for windows, increasing overhead. The process of dividing the program can be taken care of by a compilation pass more efficiently than manually by the developer, as previous Failure-Atomic literature has done [20], [21], but was not been implemented for our experiments.

Limitations

While forward progress is guaranteed by the cache window algorithm, a processor can remain frozen if the entrance of a window coincides with a lock-up period. Until the lock-up passes, the processor will not be able to continue execution on data unless an interrupt is taken to execute other code. This may not be suitable for all applications, and if it is not, a different solution should be explored

Another limitation affecting the types of programs the algorithm can protect is the requirement that a section of code cannot interact with more memory than what the cache can supply. The cache functions as a "window", into the data

Table 1. Summary of Detection Policies

Detection Policy	Access Type(s)	Description
Write-Verify	Writes	After write to address, reads from that address; verifies the read value equals the written value
Canary-N	Reads and Writes	At startup, stores N unique values at addresses from different pages; after an access, reads each address verifies each read has the expected value
Monitor-N	Reads	If multiple consecutive reads return the same value, issues a Canary-N check
Conditional-N	Writes	Uses Write-Verify if writing 0, otherwise uses Canary-N

Table 2. Summary of Detection Policy Configuration Parameters

Parameter	Applicable Policies	Description
Check Interval	All but Monitor-N	Number of accesses between checking if dropout started
Polling Delay	All	Time (ns) waiting between checking if dropout has ended
Similar Access Threshold	Monitor-N	Number of consecutive matching reads before issuing Canary-N check

region of a program. If the section of code needs to see more memory than can be opened for it, like a loop, the program cannot be divided into windows without losing performance. If the window needs to write to the data region, it also shrinks what can be prefetched. Output locations clog the cache and limit the input data that can be prefetched.

5. EVALUATION

Detection Policies

Methodology—We use five artificial applications to identify the strengths and weaknesses of each detection policy, listed in Tbl. 3. Each artificial application performs a series of reads and/or writes to a series of memory addresses. To expose the overhead of each detection algorithm, we model these applications with an ideal CPU; in other words, there is no delay between accesses, so their total latency is equal to the total time the NVM is active. The first applications perform a series of reads across a set of memory addresses (one sequentially and the other randomly). The next two applications perform a series of writes across a set of memory addresses (one sequentially and the other randomly). The last application perform a series of sequential writes, then a series of sequential reads across a set of memory addresses. Due to the simplicity of these applications, we can restart the application from any point in the application's runtime. In effect, there is a checkpoint after every memory access. Thus, when the detection policy detects lock-up, the application has to redo only the minimal amount of work. This implies that any negative effects due to checkpointing will be smaller in our artificial applications than in real applications.

The data read or written by each application can be configured in three ways. The first configuration, *normal*, uses uniform randomness to select each value between 0 and $2^{32} - 1$. The second configuration, *sparse*, sets each value to 0 with a 90% probability, and otherwise sets the value to a uniform random variable with a range between 0 and $2^{32} - 1$. The final configuration, *constant non-zero value* (CNZV), sets each value to a constant, non-zero value (we use 2) with a 90% probability, and otherwise sets the value to a uniform random variable with a range between 0 and $2^{32} - 1$. CNZV is designed to catch edge cases that appear due to accessing the same value multiple times.

We use the same seed for the random generator of each of these simulations so the lock-up begin and end times are the same for each run. Our lock-up duration and our mean time between lock-ups are randomly selected as discussed in Sec. 2.

We compare our detection policies to the *Ideal* detection policy, which knows precisely when lock-up will begin and end without incurring any overhead.

Effectiveness at Detecting Lock-up—In this section, we compare the effectiveness of each detection policy at detecting lock-up in a variety of test cases.

In addition to failures identified through these test cases, there are several other failure modes that require special intervention: If there is a significant gap in time between accesses, then lock-up could start undetected and finish after the next access but before the detection policy runs. Thus, the access could be corrupted and the corruption could be undetected. The solution for this, as mentioned in Sec. 3, is to manually insert checks before accesses that may be susceptible to this failure case. Additionally, if the check interval is greater than one, then the last access before the gap could be corrupted by lock-up, and the lock-up may end before the next check for lock-up. To avoid this failure case, manually place a detection check after the last access before a long gap between accesses.

The test cases we cover in this section vary the NVM behavior between page and non-page modes, and open page and zeros lock-up behaviors. They also vary application behavior between sequential and random accesses and the application data between normal, sparse and CNZV. There are a total of 24 test cases for reads and 24 test cases for writes. The configuration parameters are summarized in Tbl. 4.

Since some detection policies are only applicable to reads or writes, we show the failure modes for each detection policy on read-only applications in Tbl. 5 and on write-only applications in Tbl. 6. These tables also contain the largest percent of data corrupted that we observed in each failure mode. Additionally, some detection policies have significantly different error rates when check interval is large vs. small, so we list these separately.

First, we discuss the effectiveness of each detection policy on **reads**.

Canary-2 and Monitor-2 are the only read detection policies that pass every test case.

Canary-1 fails on test cases where the lock-up behavior is open page. This is because when lock-up starts while the Canary address is in the read buffer, then *every* access to

Table 3. Artificial Applications Used to Evaluate Detection Policies

Name	Description
Sequential Read	Reads from one million addresses sequentially
Random Read	Reads from one million addresses in a random order
Sequential Write	Writes to one million addresses sequentially
Random Write	Writes to one million addresses in a random order
Sequential Write-then-Read	Writes to one million addresses sequentially, then reads the same one million addresses sequentially

Table 4. Configuration Parameters for Artificial Applications

Name	Possible Values	Description
Data Configuration	normal, sparse, CNZV	Specifies if data is normally distributed (normal), is 90% sparse and 10% normally distributed (sparse), or is 90% a constant non-zero value and 10% normally distributed (CNZV)
Page Mode	page, non-page	Specifies whether or not the NVM is in page mode
Lock-up Mode	open page, zeros	Specifies if, during lock-up, the NVM returns data from the page open at the start of lock-up or zeros

Table 5. Failure modes and maximum percent of data corrupted over all the test cases for each read detection policy

Detection Policy	Check Interval	Failure modes (Max % Data Corrupted)
Canary-1	Small	Open page (27.7%)
Canary-1	Large	Open page (0.48%)
Canary-2	**Either**	**None**
Monitor-1	N/A	Open page and sparsity or CNZV (15.54%)
Monitor-2	**N/A**	**None**

Table 6. Failure modes and maximum percent of data corrupted over all the test cases for each write detection policy

Detection Policy	Check Interval	Failure modes (Max % Data Corrupted)
Write-Verify	Small	All zeros and sparse (0.42%) or open page and sparse or CNZV (0.33%)
Write-Verify	Large	All zeros and sparse (35.9%) or open page and sparse or CNZV (32.02%)
Canary-1	Small	Open page (27.67%)
Canary-1	Large	Open page (0.48%)
Canary-2	**Either**	**None**
Conditional-1	Small	Open page and sparse (0.19%) or open page and CNZV (0.33%)
Conditional-1	Large	Open page and sparse (0.45%) or open page and CNZV (32.02%)
Conditional-2	Small	Open page and CNZV (0.33%)
Conditional-2	Large	Open page and CNZV (32.02%)

that page index will be the Canary value. Thus, Canary-1 is frequently fooled into believing lock-up has not occurred. Canary-1's failure rate drops significantly when check interval is high because the Canary value is less likely to be in the read buffer when lock-up starts.

Monitor-1 fails on test cases where the lock-up behavior is open page and the application data is either sparse or CNZV. Since Monitor-1 issues a Canary-1 check whenever several reads to the same page index return the same value, when there are many identical values being read *correctly* then the Canary address is frequently in the open page. Thus, lock-up can begin while the Canary address is in the open page, fooling Monitor-1 into thinking lock-up has not occurred.

Next, we discuss the effectiveness of each detection policy on **writes**.

Canary-2 is the only write detection policy that passes every test case.

Write-Verify fails on test cases where the lock-up behavior is all zeros and the application data is sparse or where the lock-up behavior is open page and the application data is sparse or CNZV. These two failure cases have the same root cause: Write-Verify can fail to detect lock-up when the value returned during lock-up is the same as the value written. Note that when the check interval is small, the impact of a false negative is small because it only validates a small number of accesses. However, when the check interval is large, false negatives incorrectly validate a large number of accesses; this explains the significant increase in the percent of data corrupted for large check intervals.

Canary-1 fails on test cases with an open page for the same reason it can fail on read-only applications. Also similar to read-only applications, Canary-1 is less likely to fail with a large check interval. While it is true that false negatives are more impactful with a large check interval, this effect does not outweigh the drop in Canary-1's error rate due to a large check interval.

Conditional-1 fails on test cases where the lock-up behavior is open page and the application data is sparse or CNZV. It has two different failure cases: a) When the lock-up behavior is open page and the application data is sparse, then Conditional-1 frequently calls Canary-1, leading to the same failure case as Canary-1. However, much less data is corrupted in this failure case than in Canary-1. This is because as soon as a non-zero value is written, then Conditional-1 will perform a Write-Verify check instead of a Canary-1 check, and Write-Verify will detect the lock-up. b) When the lock-up

behavior is open page and the application data is CNZV, then the CNZV is likely to be caught in the open page when lock-up begins. If the next value to write is the CNZV, then Write-Verify will not catch that it was not written. The percent of data corrupted in case b) increases by several orders of magnitude when the check interval is large because false negatives have an increased impact with large check intervals.

Conditional-2 fails on test cases where the lock-up behavior is open page and the application data is constant non-zero values. This is due to the same failure case as b) in Conditional-1, so it shares the same behavior when check interval is large.

In **summary**, to consistently detect lock-up, use Monitor-2 or Canary-2 for reads and Canary-2 for writes. If infrequent errors are acceptable, then Canary-1 with a large check interval can be used for both reads and writes. If infrequent errors are acceptable and there are few CNZV in the application data, then Conditional-2 can be used for writes. If infrequent errors are acceptable and the application data is not sparse and has few CNZV, then Monitor-1 can be used for reads and Write-Verify and Conditional-1 can be used for writes.

Overhead on Reads—In this section, we evaluate the overhead of Canary-1, Canary-2 and Monitor-2 on the sequential read-only application while the NVM is in page mode. The trends identified in this section are applicable to other application behaviors and non-page mode.

First, we show how Canary-1 and Canary-2's latency change with respect to Ideal's latency as we increase their check interval in Fig. 2. We use the all zeros lock-up behavior so that Canary-1 does not ignore any lock-up periods, which would unfairly reduce its latency. While increasing the check interval initially decreases latency because it reduces the number of Canary-1 and Canary-2's accesses, at some point latency begins to increase again because more and more valid accesses must be redone since they were not verified by the detection policy before lock-up started. Based on this experiment, the optimal value for check interval is 600 for Canary-1 and 500 for Canary-2; however, these values will likely change for other applications and NVMs. With these values of check interval, Canary-1 adds an overhead of 0.95% of the latency of Ideal and Canary-2 adds an overhead of 2.3%. We have verified that the shape of the latency vs. check interval curve holds across lock-up behaviors and page modes.

Figure 2. Latency of Canary-2 vs. its check interval compared to Ideal on the sequential read-only application while the NVM is in page mode.

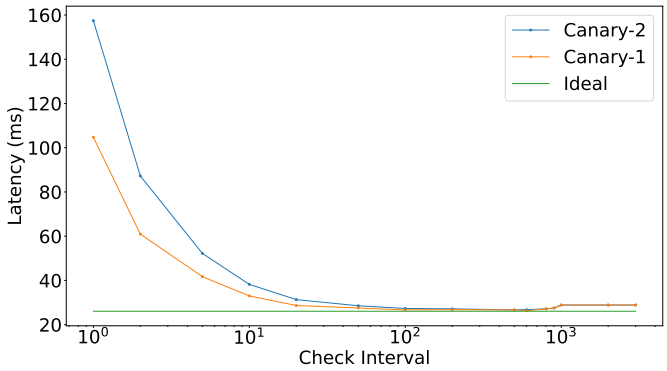
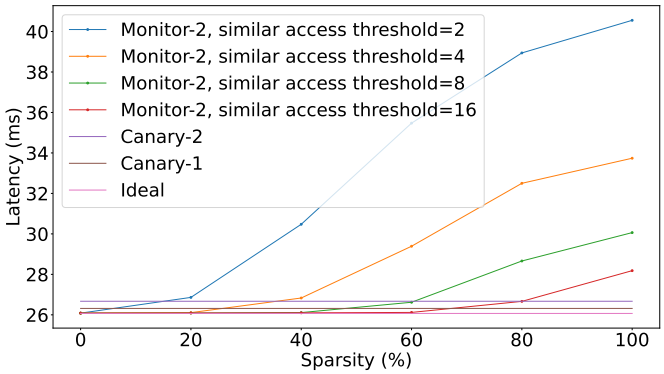


Figure 3. Latency of Monitor-2 with various similar access thresholds compared to Canary-2 with a check interval of 500 and Canary-1 with a check interval of 600 on the sequential read application in page mode with lock-up verses percent of sparsity in the application.



Next, in Fig. 3 we show how Monitor-2's latency varies as the application sparsity increases and as we increase the similar access threshold. At 0% sparsity, Monitor-2 adds a latency overhead of less than 0.06% over Ideal. However, at 100% sparsity, even with a similar access threshold of 16 Monitor-2 has an overhead of 8.1%. Thus, for applications with significant sparsity, Canary-1 and Canary-2 have less overhead than Monitor-2. Otherwise, Monitor-2 has less overhead than Canary-1 and Canary-2. We have verified that the shape of these curves holds across lock-up behaviors and page modes.

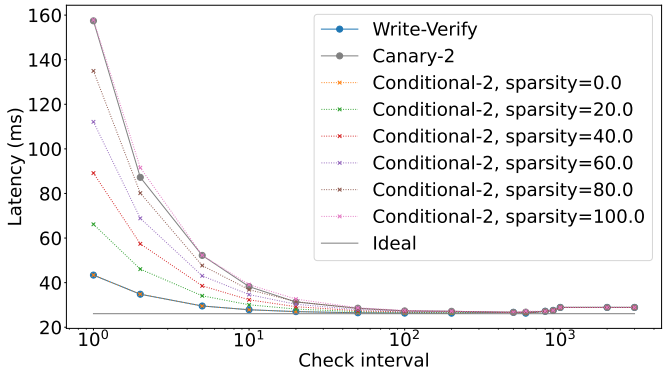
If 100% reliability is needed, then Canary-2 and Monitor-2 are the only options for reads. If the application data is highly sparse, use Canary-2 and sweep check interval to identify the setting with the lowest overhead. Otherwise, use Monitor-2 with a large similar access threshold. If some errors can be tolerated, such as in approximate computing, and the data is highly sparse, then lower overhead can be achieved using Canary-1 instead of Canary-2. Note that there is no reason to use Monitor-1; when it has low error, Monitor-2 has less than 0.1% overhead.

Overhead on Writes—In this section, we evaluate the overhead of Canary-2, Write-Verify, and Conditional-2 on the sequential write-only application while the NVM is in page mode. The trends identified in this section are applicable to Canary-1, Conditional-1, other application behaviors, and non-page mode.

In Fig. 4, we plot the latency of Write-Verify, Canary-2, Conditional-2, and Ideal verses check interval. We plot Conditional-2 several times with varying levels of application sparsity. As check interval increases, the latency of Write-Verify, Canary-2, and Conditional-2 decreases up until a point, after which their latency increases. The decrease in latency is due to fewer detection accesses, but the increase in latency is due to valid accesses being redone because they were not verified by the detection policies before lock-up started. The latency of Conditional-2 is for the most part between Write-Verify and Canary-2; it behaves more like Write-Verify with lower application sparsity and more like Canary-2 with higher application sparsity. The check interval with lowest latency is 500 for Canary-2 and 600 for Write-Verify, leading to overheads of 2.3% and 0.87%, respectively.

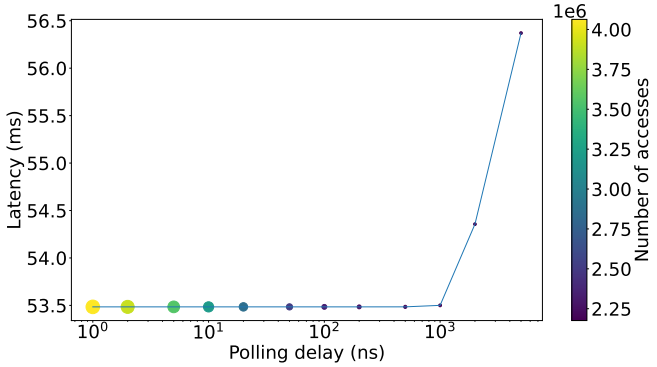
If 100% reliability is needed, then Canary-2 is the only option for writes. Sweep check interval to identify the setting with the lowest overhead. If infrequent errors can be tolerated, then the other detection policies typically have lower overhead than Canary-2.

Figure 4. Latency of Write-Verify, Canary-2, Conditional-2, and Ideal verses check interval and percent sparsity on the sequential write-only application in page mode with lock-up.



Varying the polling delay—In this section, we explore the costs and benefits of increasing the polling delay on Canary-2 while checking for lock-up to end. Specifically, in Fig. 5, we show the latency and number of NVM accesses on the sequential write-then-read application in page mode with lock-up verses the polling delay. The number of accesses at each datapoint is shown by the color and size of the point. Increasing the polling delay decreases the number of accesses from over four million to just above two million, which represents significant power savings. However, after the polling delay passes one millisecond, the latency significantly increases because the processor is idling while the NVM is available. To reduce power consumption without increasing latency, the polling delay should be the largest value that does not significantly increase latency. For this application, that point is at one ms, with an 86% reduction in accesses and a 0.03% increase in latency.

Figure 5. Latency and number of accesses of Canary-2 on the sequential write-then-read application in page mode with lock-up verses the polling delay.



Extrapolating Simulation Results— The simulation results are based on artificial applications, so the specific numbers referenced in this section will change, although the trends will stay the same. Since the simulator assumes applications are completely memory-bound, it over-emphasizes the impact of detection policies. Real-world applications will likely be less affected by detection policies. Additionally, real-world applications are typically constrained on where they can insert checkpoints, and there is likely overhead to saving and restoring checkpoints. Because the artificial applications do not have these overheads, the simulation results likely underestimate the impact of lock-ups when the check interval is greater than one.

Cache Window Algorithm

To show the effectiveness of the cache window algorithm, we simulate a matrix multiplication benchmark. Matrix multiplications represent the most common memory-bounded operation widely used in Machine Learning and Computer Vision tasks [11].

Benchmark Setup—Lock-ups in the simulated environment for the Matrix Multiplication benchmark behave by returning zeros on reads and doing nothing on writes. This was chosen since this was observed as the most common behavior of our physical models. Returning zero also allows us to track if a detection policy is failing to produce non-corrupted data, despite initalizing them in their most aggressive forms. Probabilities on lock-ups is as discussed in Sec. 2. All experiments were conducted with page-mode enabled on the NVM device.

Three detection policies were selected to be tested in this environment. Firstly, the Ideal policy to show the performance of the benchmark without any overhead. Secondly, a Canary-1 read policy with a Conditional-2 write policy at a check interval of 500 as we recommend. Lastly, a Monitor-2 read policy with a Canary-2 write policy also at a check interval of 500 as we recommend.

Parameters for the processor were based off of the SiFive FU740 RISC-V Core, which has a max clock frequency of 1 GHz, a max Instruction Per Cycle rate of 2, an L1 Data-Cache of 32KiB, and load instructions take 4 Cycles Per Cache Hit [22]. Advanced eXtendable Interface (AXI) Link Delay towards the NVM device was also estimated, with each access taking 6ns. These parameters combine to produce a best-case scenario state of execution when using a non-volatile component with potential lock-ups.

Each detection policy advances the simulation by exactly one pipeline instruction delay and an AXI delay. The pipelined instruction delay for the RISC-V Processor is:

$$\textbf{Clk Period} \cdot \textbf{Inst. per Cycle} = \frac{1}{1\textbf{GHz}} \cdot 2 = 0.5\textbf{ns}$$

Total CPU delay on communicating with the NVM device in the fastest processing speed is 6.5ns.

The base experiment chosen was a Read-Only version of the matrix multiplication benchmark with 512x512 Matrices, values calculated are not written back to the device. Variations on this base are as follows:

- Read-Write, each value computed is written back to the NVM
- Short Lock-ups, lock-up periods were reduced to an average 1us and a standard deviation of 100ns
- Slow Processor, the RISC-V Processor was reduced from 1GHz Clock Rate and 2 IPC to 100MHz and 0.5 IPC

Figure 6. Simulated Runtime of the Matrix Multiplication Benchmark.

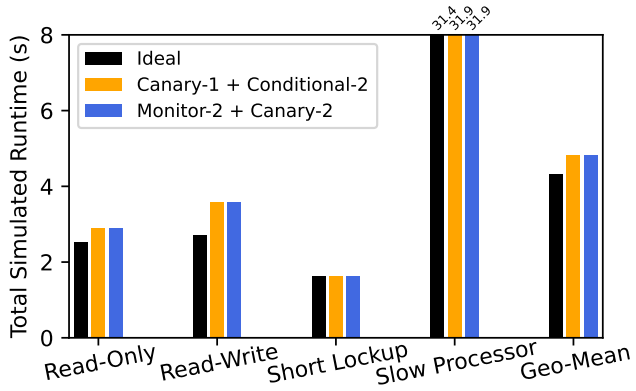
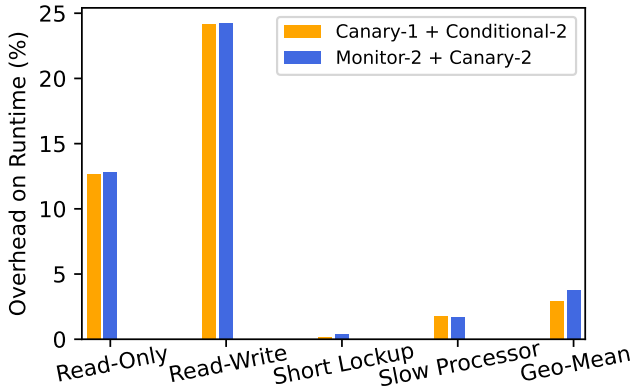


Figure 7. Percent of time each configuration was executing the detection policy.



All values are initialized into the simulated NVM before simulation and then the application preloads the values into the cache during execution, computes the matrix multiplication per individual element, and compares it to the true value. In no configurations, except Canary-1 + Conditional-2 during Short Lock-ups, was there corrupted data.

Tracked metrics include the total simulated runtime of the application (Figure 6), the percent of that time spent on the detection policy (Figure 7), and the percent of time the application's execution of a window overlapped with a lock-up period compared to the rest of the runtime (Figure 8). Respectively these measures indicate performance of the configuration, the overhead of the detection policy, the potential efficiency of the Cache Window algorithm in the configuration. Each metric is then averaged using a geometric mean, which follows the equation:

$$\textbf{Geo-Mean} = \sqrt[k]{n_0 \cdot n_1 \cdot \textbf{...} \cdot n_k}$$

The single-threaded simulation software was ran on computer with an i7-8665U running at 1.9 GHz, and 32 GBs of 2400 MT/s RAM. GPU Acceleration was not used for the simulation.

Total Runtime—Simulated Total Runtime, Figure 6, includes time that the application waited for data at the entrance of a window (the prefill cache stage) due to a lock-up. If a detection policy has a higher overhead, or a slower lock-up detection rate, the application spends more time loading verified data for the program. Slower resumptions of the windows translate to longer total execution times, exposing the application to further lock-ups.

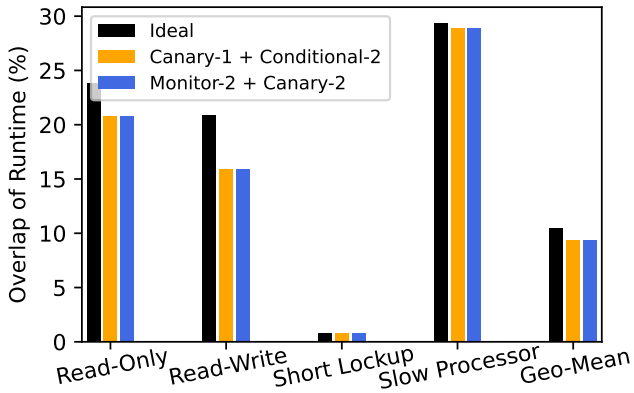
The Slow Processor configuration suffers the most from this negative feedback loop, as it has an average of ten more seconds of lock-up. The Ideal mitigation in the slower processor configuration experienced 125,459 outages compared to the faster processor in Read-Only having 10,063 outages. However, much of the runtime is due to the slower processor and is not due to the detection policies utilized ($<5\%$ overhead).

Overhead—On dense workloads, such as this benchmark, the likelihood of repeated reads are extremely low but not zero. Monitor-2 detection policy has a higher overhead due to the chance of repeated values in the data, see Figure 7. We don't see this issue with Canary-1 + Conditional-2 because overhead is not a function of input data.

We explored increasing the repeated value threshold for Monitor, however saw no noticable difference in performance. The increased overhead may represent how the total execution time was reduced, therefore the percentage scales with it.

Overlap/Operate-Through

Figure 8. Percent of Time each configuration executed instructions while a lock-up was occurring.



One of the advantages of the Cache Window algorithm is the possibility of the processor being able to continue execution during a lock-up. Figure 8 shows a relation between the speed of the processor, the lock-up period length, and the overlap percent. Shorter lock-ups for a high performance processor do not correlate to a higher overlap percentage. However, slower processors can preempt lock-ups if they're able to preload fully.

Reliability

Canary-1 + Conditional-2 fails to be as reliable as Monitor-2 + Canary-2. While there were no errors in almost all

simulations, Canary-1 + Conditional-2 resulted 39089 errors ($\sim 15\%$ of written values), in the Shorter Lock-up configuration while no other detection policy had errors.

A slower processor does not make errors more likely because the primary bottleneck on the check interval is memory access time. It takes the NVM $10\textbf{ns} \cdot 500$ **accesses** $= 5000\textbf{ns}$ to load an entire check interval in the best-case scenario. In total, the processor spends $\sim$ 100ns executing preloads for the data in a check interval of the fast processor. The slow processor spends $\sim$ 300ns for the same interval, which is not significant compared to the check interval load time.

The errors seen in the Short Lock-up configuration can be explained by the average lock-up time of the configuration (1us) being much lower than the total load time of one check interval (5.1us). Monitor can catch the error because its max detection delay is 20-60ns (two repeated reads), while Canary-1 has to wait 5.1us (the entire check interval) before catching the error. Reducing the check interval can increase Canary-1's reliability, however it would not solve for the open buffer lock-up case. The overhead of the two configurations is close enough that the increase in reliability with Canary-1 for a trade-off in overhead would not be worth the trouble.

6. Future Work

Much of this work has been based off of physical devices and behaviors observed when testing them. While we were able to collect valuable data to make the simulations possible, the detection policies described in this paper were not yet in a working state by the time of the radiation test. Our upcoming studies will involve implementing these policies into our physical devices and exploring differences that arise from real-world tests compared to simulation results.

As we transfer our algorithms onto physical devices, we plan to implement each detection policy in hardware rather than the CPU. This may reduce the latency of lock-up checks.

Furthermore, this paper has been assuming single-core bare-metal computing tasks which have complete control over the cache. The Cache Window Algorithm will need to be adjusted for more modern use-cases such as multi-core and operating system environments. Since the Cache Window Algorithm manages memory using software, either the location that verified data is placed in will need to change, or a new hardware component functioning like a type of cache will be needed to accelerate the algorithm.

An additional challenge with operating system environments is that delays between accesses may be unpredictable (due to events such as context switches). Future work may address this problem by adding a hardware module which periodically checks if the NVM is in lock-up.

Future work may include evaluating our detection policies and the Cache Window Algorithm on real applications instead of artificial applications. These evaluations will allow us to have increasing confidence in the effectiveness and performance of our detection policies and the Cache Window Algorithm.

Another avenue to explore is implementing the Cache Window Algorithm as a compiler pass. For now, due to the simulation environment, the preload and flush steps were manually inserted. Manually inserting the steps is time-consuming on the part of the developer and is error-prone due to difficulties in knowing how much memory each section of code interacts with. A compiler pass with very minimal alias-analysis would be able to automate the process, also allowing for unmodified source code to be used on lock-up sensitive devices.

Lock-ups may not only affect NVMs, but also other I/O devices and Volatile Memories. An exploration of whether these mitigations are effective on those devices would provide insight on how lock-ups can affect electrical components overall.

7. Conclusion

In this paper, we for the first time model lock-up as a failure mode in NVMs, which can impact the reliability and availability of the NVM. We provide novel mitigation policies to improve the reliability and availability of NVMs which face lock-up. To address reliability, we create and evaluate several detection policies. We find that Canary-2 and Monitor-2 can have 100% reliability with a small overhead. We also present several other detection policies that have less reliability but a smaller overhead. To address availability, we present a novel Cache Window algorithm which exploits the cache's ability to hide external latency to also hide temporary external device failures. We also show that the best detection policy for the Cache Window algorithm is Monitor-2 on reads and Canary-2 on writes, with no reported errors and equivalent overhead to less reliable detection policies.

References

[1] A. Hopkins, T. Smith, and J. Lala, "Ftmp—a highly reliable fault-tolerant multiprocess for aircraft," *Proceedings of the IEEE*, vol. 66, no. 10, pp. 1221–1239, 1978.

[2] Q. Li, S. Wang, C. Xu, X. Ma, M. Xu, A. Zhou, R. Xing, B. Yang, Z. Zhu, Y. Zhang, and X. Liu, "Exploring real-time satellite computing: From energy and thermal perspectives," in *2024 IEEE Real-Time Systems Symposium (RTSS)*, 2024, pp. 161–173.

[3] B. L. Edwards and D. J. Israel, "A geosynchronous orbit optical communications relay architecture," in *2014 IEEE Aerospace Conference*, 2014, pp. 1–7.

[4] Y. Baruah, S. Roy, S. Sinha, E. Palmerio, S. Pal, D. M. Oliveira, and D. Nandy, "The loss of starlink satellites in february 2022: How moderate geomagnetic storms can adversely affect assets in low-earth orbit," *Space Weather*, vol. 22, no. 4, p. e2023SW003716, 2024, e2023SW003716 2023SW003716. [Online]. Available: https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1029/2023SW003716

[5] O. Gonçalves, G. Prenat, and B. Dieny, "Radiation hardened mram-based fpga," *IEEE Transactions on Magnetics*, vol. 49, no. 7, pp. 4355–4358, 2013.

[6] C. Horch, D. Garbe, and K. Schäfer, "A reliable supervisor system utilizing an fd-soi fpga and mram for cots-based payload data processing," in *2025 European Data Handling Data Processing Conference (EDHPC)*, 2025, pp. 1–4.

[7] R. Merl and P. Graham, "A low-cost, radiation-hardened single-board computer for command and data handling," in *2016 IEEE Aerospace Conference*, 2016, pp.

1–8.

[8] R. Merl, E. Cox, R. Dutch, P. Graham, S. Larsen, J. Michel, D. Milby, K. Morgan, and K. Tripp, "Leon4 based radiation-hardened spacevpx system controller," in *2020 IEEE Aerospace Conference*, 2020, pp. 1–10.

[9] S. Roffe and A. D. George, "Evaluation of algorithm-based fault tolerance for machine learning and computer vision under neutron radiation," in *2020 IEEE Aerospace Conference*, 2020, pp. 1–9.

[10] C. Oh, H. Youn, and V. Raj, "An efficient algorithm-based fault tolerance design using extended rearranged hamming checksum," in *Proceedings 1992 IEEE International Workshop on Defect and Fault Tolerance in VLSI Systems*, 1992, pp. 237–246.

[11] D. Chen, H. Kim, A. Phan, E. Wilcox, K. LaBel, S. Buchner, A. Khachatrian, and N. Roche, "Single-event effect performance of a commercial embedded reram," *IEEE Transactions on Nuclear Science*, vol. 61, no. 6, pp. 3088–3094, 2014.

[12] G. Korkian, D. León, F. J. Franco, J. C. Fabero, M. Letiche, Y. Morilla, P. Martín-Holgado, H. Puchner, H. Mecha, and J. A. Clemente, "Single event upsets under proton, thermal, and fast neutron irradiation in emerging nonvolatile memories," *IEEE Access*, vol. 10, pp. 114 566–114 585, 2022.

[13] "Parallel interface mram," 2025. [Online]. Available: https://www.everspin.com/parallel-interface-mram

[14] "Space grade parallel sram memory," 2025. [Online]. Available: https://www.avalanche-technology.com/document-page/1gbit-8gbit-x32-parallel-mram-memory-space-grade/

[15] T. Eshita, W. Wang, K. Nakamura, S. Mihara, H. Saito, Y. Hikosaka, K. Inoue, S. Kawashima, H. Yamaguchi, and K. Nomura, "Development of ferroelectric ram (fram) for mass production," in *2014 Joint IEEE International Symposium on the Applications of Ferroelectric, International Workshop on Acoustic Transduction Materials and Devices & Workshop on Piezoresponse Force Microscopy*, 2014, pp. 1–3.

[16] M. E. O'Neill, "Pcg: A family of simple fast space-efficient statistically good algorithms for random number generation," Harvey Mudd College, Claremont, CA, Tech. Rep. HMC-CS-2014-0905, Sep. 2014.

[17] "Numpy random generator," available at https://numpy.org/doc/stable/reference/random/generator.htm

[18] H. Sun, C. Liu, N. Zheng, T. Min, and T. Zhang, "Design techniques to improve the device write margin for mram-based cache memory," in *Proceedings of the 21st Edition of the Great Lakes Symposium on Great Lakes Symposium on VLSI*, ser. GLSVLSI '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 97–102. [Online]. Available: https://doi.org/10.1145/1973009.1973030

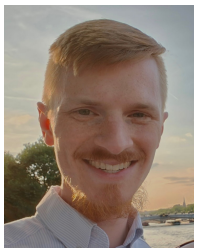
[19] M. Alshboul, H. Elnawawy, R. Elkhouly, K. Kimura, J. Tuck, and Y. Solihin, "Efficient checkpointing with recompute scheme for non-volatile main memory," *ACM Trans. Archit. Code Optim.*, vol. 16, no. 2, May 2019. [Online]. Available: https://doi.org/10.1145/3323091

[20] Q. Liu, J. Izraelevitz, S. K. Lee, M. L. Scott, S. H. Noh, and C. Jung, "ido: Compiler-directed failure atomicity for nonvolatile memory," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 258–270.

[21] Y. Xu, J. Izraelevitz, and S. Swanson, "Clobber-nvm: log less, re-execute more," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 346–359. [Online]. Available: https://doi.org/10.1145/3445814.3446730

[22] "Sifive u74-mc core complex manual 21g1.01.00," 2021. [Online]. Available: https://starfivetech.com/uploads/u74mc_core_complex_manual_21G1.pdf

BIOGRAPHY



Daniel Puckett is a PhD student in Computer Engineering at Texas A&M. He received his B.S. in Computer Engineering from Texas A&M and his M.S. in Electrical Engineering from Rice University. He is currently a research and development intern at Sandia National Laboratories. During his four years of academic research experience and over five years of internship experience, he has built expertise in computer architecture, machine learning, radiation hardening, and neuromorphic computing.

Henri Malahieude is a Ph.D. Student studying Electrical and Computational Engineering at the University of Colorado Boulder. He received his B.S. in Computer Science from the University of California, Riverside. Currently he is serving as a Year-Round Intern for the Microsystems Engineering, Science, and Applications (MESA) organization at Sandia National Laboratories.

Joseph D'Amico is a senior staff member at Sandia National Laboratories, where he researches radiation effects on advanced microelectronic systems. He has a BS from Rice University and an MS and PhD from Vanderbilt University, all in electrical and computer engineering. He has worked on projects ranging from low-level integrated circuit layouts to high-level machine-learning software, but his most recent research has focused on enabling the use of emerging commercial non-volatile memories in extreme radiation environments by using algorithmic hardening techniques.