Persistent Memory Test Framework

by

Caleb Toney

A creative component submitted to the graduate faculty
in fulfillment of the requirements for the degree of
MASTER OF SCIENCE

Major: Computer Engineering

Program of Study Committee:
Dr. Henry Duwe, Major Professor

The student author, whose presentation of the scholarship herein was approved by the program of study committee, is solely responsible for the content of this creative component. The Graduate College will ensure this creative component is globally accessible and will not permit alterations after a degree is conferred.

Iowa State University
Ames, Iowa
2019

Copyright © Caleb Toney, 2019. All rights reserved.

TABLE OF CONTENTS

List of Figures and Tables

ABSTRACT

This creative component introduces a trace based testing framework for persistent memory programs. Using a number of tools and techniques, the framework checks for expected output, hangs, and errors. The framework builds on the ideas put forth in previous persistent memory testing frameworks and could be the base of a full featured test framework in the future. The framework introduces a trace generator, memory map generator, and bug checker in conjunction with a program that recovers from power failures by accessing persistent memory. We show that the framework is able to detect synthetic hangs and recovery code errors.

INTRODUCTION

Persistent memory is a form of byte-addressable nonvolatile memory used in computer systems. Currently, volatile memory such as DRAM is used primarily to temporarily store data or program code for the processor. Persistent memory has many advantages over volatile memory, the biggest of which is data protection through power failures. Recent years have seen persistent memory get both cheaper and faster and become accordingly more common in computing systems. Over time, persistent memory will likely begin to replace DRAM as the performance gap closes. As persistent memory becomes more common, a new set of challenges must be solved. For example, as persistent memory transactions are executed, properties such as atomicity, durability, consistency, and isolation must be ensured. Further, programming for persistent memory is difficult as there are many different persistent memory programming models. Errors in programming can lead to insidious and difficult to detect bugs.

Many proposals have attempted to address these challenges through both software and hardware solutions. Hardware solutions are often focused on increasing performance, but can also ensure properties such as durability. Software solutions often attempt to ease the difficulty of programming for persistent memory through libraries, programming models, and file systems. These solutions seek to increase functionality and efficiency and reduce memory consistency errors.

When programming mistakes do happen, they can be very difficult to detect as they often occur only in very specific power failure scenarios. Several testing frameworks have been developed to detect and warn programmers against potential consistency errors as a solution to this problem. Each framework seeks to find bugs through program annotation, runtime tracing, static testing, and many other methods. Each framework succeeds to varying degrees at quickly and efficiently detecting bugs that would otherwise be difficult to detect.

This creative component seeks to create a testing framework building on the ideas and concepts of previous research. The framework uses dynamic, trace-based crash consistency logic to detect errors in a program’s output after undergoing crash recovery. The framework allows for simulated power failure at any point in a program’s runtime. The framework produces a trace, memory map, and error detection code for a given test program. This creative component was able to produce some promising results by detection of synthetic bugs in a test program. Although the framework is still in its early stages, this creative component can be the basis for an efficient persistent memory test framework.

BACKGROUND & RELATED WORKS

Persistency

Persistent memory is memory that can survive system failure or power loss and thus retain critical data. Nonvolatile memory could potentially become a replacement for existing DRAM as new technologies are introduced. In the past, nonvolatile memory was too expensive for wide scale adoption, but is quickly becoming prevalent in the server market. As costs come down, persistent memory will continue to become more common in servers and other computing systems.

Early persistent memory systems used a variety of technologies to provide for persistency, but often relied on copy-on-write or write-through buffer technologies. One early example called eNVy [1] used a copy-on-write flash array implementation to provide for persistent memory at a fraction of the cost of battery-backed SRAM. In addition to hardware solutions, software solutions are necessary to provide guarantees about the consistency and validity of data. One early software solution called RVM [2] provided a Unix library to map and guarantee the consistency of Unix virtual memory regions. Early technologies have been continuously improved and iterated to achieve cheaper, faster products.

More recent technologies being used in nonvolatile memory include phase-change memory (PCM) and spin STT-MRAM. PCM is a technology built using an alloy that has two phases that have high and low resistance respectively [16]. As the temperature of the alloy changes, it changes between its amorphous and polycrystalline phases. It has a significantly faster write time than flash based technologies and can be altered at the bit level. Additionally, it is a backend technology that could be embedded in many technology nodes. STT-MRAM or spin-transfer torque MRAM works by flipping the spin of electrons using a spin-polarized current [17]. Two of the newest and most promising types of persistent memory are NVDIMM and 3DXPoint [3]. 3DXPoint is sold by Intel under the name Optane and is a layered storage solution based on bulk resistance changes. It sits on the memory bus and connects to a processor's onboard memory controller. It can function as main memory without persistence or can be used as persistent memory via memory-aware applications or a separate file system. It functions slower than traditional DRAM with an average random load time of 305 ns vs 81 ns respectively [3]. Optane's primary use, however, is as a byte-addressable persistent memory that gets mapped into the user space using mmap(). Users can then load and store directly to that address space. The performance advantage of 3DXPoint in this scenario varies widely based on application use, but does speed up nonvolatile memory accesses significantly.

Current software technologies attempt to optimize access times and accessibility to programmers of nonvolatile memory systems. For example, Mnemosyne [4] is a lightweight programming interface designed for programming to persistent memory. NOVA [6] is a file system designed to optimize persistent memory performance and provide consistency guarantees. Additionally, many other solutions exist to optimize both the programming process and efficiency of persistent memory. Often those systems use a form of logging such as undo logging, redo logging, or journaling to guarantee crash consistency by maintaining two versions of data. Optimistic crash consistency [15] uses a journaling file system that decouples durability and ordering of writes to create a new commit protocol. This protocol allows for efficient and guaranteed data storage. Other examples of journaling file systems include Linux ext3 and

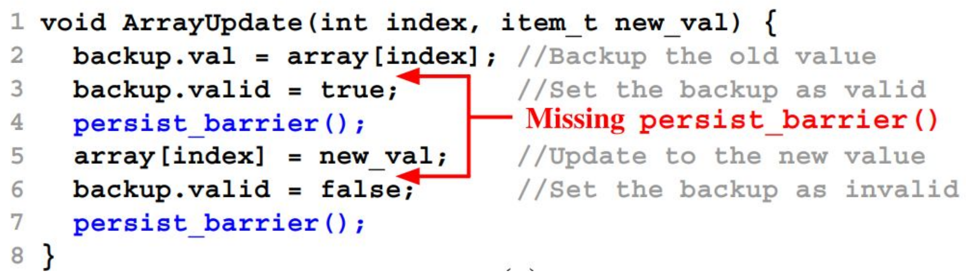
NTFS. Software can significantly reduce persistent write overhead by efficiently ordering and accessing persistent memory writes.

Persistent data relies on four different properties described by the acronym ACID [5]. Each letter in ACID stands for a consistency property necessary to guarantee error-free persistent memory access. Atomicity is the ability of memory transactions to ether complete successfully or fail completely, resulting in no partial data corruption. Consistency is the ability of data to remain in consistent, valid states. This means that a memory transaction will never be placed in an invalid, unreadable state. Consistency does not necessarily guarantee that a transaction is correct. Isolation is the guarantee that memory will be in the same state after transaction reordering that it would have been if the accesses had been executed sequentially. This is necessary as systems reorder memory accesses for efficiency. Durability is the ability of data to survive system failures such as power failure. Each of these properties have varying degrees of necessity to persistent memory systems, and many proposed ideas focus on guaranteeing one or two of the properties. In a system in which persistency can be guaranteed, all four properties must be guaranteed.

Examples of persistent memory bugs

Persistent memory bugs can be both difficult to detect and lead to errors in program execution. Part of the reason they are so difficult to detect is that they can vary widely in what causes them [7]. They can be caused by improper reordering of writes, improper use of cache flushes, recovery code coding errors, failure to log file system modification, and many other issues. Additionally they can be caused by the programmers themselves or by a library the programmer may be using.

Improper reordering of writes means that a program or file system has violated the isolation property of the ACID model guarantees. This could be caused by missing or misplaced ordering or durability instructions such as sfence and mfence. The fence instructions allow a programmer to use a consistency model such as sequential or relaxed. If the programmer improperly uses these instructions, transactions may be reordered such that they produce a different result than they would have if executed sequentially. One example given by PMTest [8] can be seen in figure 1. In this example, there needs to be a persist_barrier after creating a backup copy of array[]. Without the correctly placed persist_barrier, lines can be rearranged incorrectly and thus the array could become unrecoverable in the event of a power failure. These bugs could occur when a programmer is using low-level functions as part of their program. These low-level primitives are very complex to use even when a programmer has a strong understanding of crash consistency programming. As such, it is easy for a programmer to accidentally create a bug in their persistency code.



```
void ArrayUpdate(int index, item_t new_val) {
  backup.val = array[index]; //Backup the old value
  backup.valid = true;       //Set the backup as valid
  persist_barrier();
  array[index] = new_val;    //Update to the new value
  backup.valid = false;      //Set the backup as invalid
  persist_barrier();
}
```

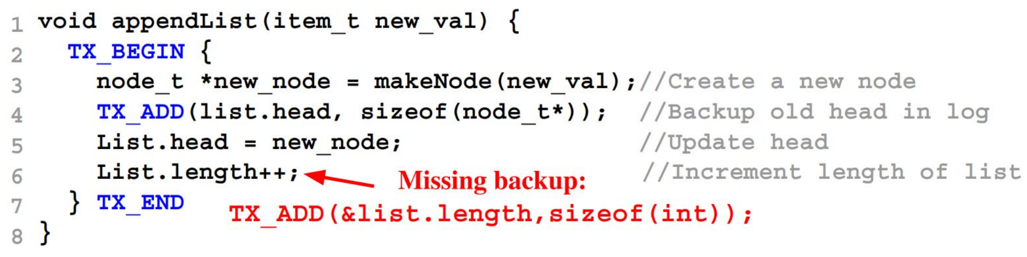
Figure 1: Example of potential improper write reordering

Recovery code coding errors are some of the simplest and easiest to understand types of memory persistency bugs. As an example, say a program has a while loop that says “while (x !=

10000){++x;}”. If x is a persistent value and a power failure were to occur when x is 10,000 before the while condition had been checked, the program could enter an infinite loop. If the program started its recovery inside the loop and thus incremented x again before checking the while loop condition, x would then be 10,001 and never satisfy the exit condition. Another example is described in the results section in which the recovery code incremented a for loop before completing the necessary writes and thus output an incorrect set of results.

A specific example of a complicated persistent memory bug potentially detectable by an automated test framework is described by Yat[7]. In this case, a linked truncate list was being used for recovery. Two separate threads were deleting inodes at the same time. Inode1 was deleted by thread 1 and added to the truncate list. Before this is committed, thread 2 starts deleting inode2 and is committed just as a power failure occurs. During recovery, the uncommitted first transaction is reverted but the committed second transaction is not reverted. Since they are linked lists, inode2’s memory is never freed. This bug would be very difficult to detect manually as it requires power failure at a very specific point. The ability to simulate power failure at a number of different points in a test framework allows for detection of bugs that require specific failure points.

In many cases a programmer will use a library or transactional interface rather than programming persistency code themselves. An example of a persistency bug created using a transactional interface given by PMTest [8] is shown in figure 2. In this example, the programmer is using the TX_BEGIN abstraction, but forgets to backup the list.length. While abstraction out to a library or transactional interface makes it significantly easier for the programmer, it is still possible that the programmer misuses the interface. It is also possible that the library itself contains persistency bugs.



```
void appendList(item_t new_val) {
  TX_BEGIN {
    node_t *new_node = makeNode(new_val);//Create a new node
    TX_ADD(list.head, sizeof(node_t*));  //Backup old head in log
    List.head = new_node;                //Update head
    List.length++;                       //Increment length of list
  } TX_END
}
```

Figure 2: Example of persistency bug using transactional interface

These examples of persistency bugs show the varied and complex nature that can cause recovery errors. Their precise timing and varied conditions illustrate why consistency bugs are difficult to detect via traditional debugging means. Automated frameworks allow for potential detection of these difficult bugs that are otherwise difficult to find.

Bug Avoidance

Developing for persistent memory is both difficult and bug prone. While testing frameworks have been created to check for bugs, much research has been done to try and prevent the bugs in the first place. Programming libraries, interfaces, guarantees, file systems, and hardware design are all methods researchers have used to deal with the unique challenges of persistent memory programming. One of the earliest persistent memory programming frameworks was described in Lightweight Recoverable Virtual Memory (RVM) [12] from 1993. Their approach sought to give the programmer specific control of atomicity, permanence, and serializability. They specifically valued simplicity to allow for easier programming without bugs. It is a Unix library that is only around 10,000 lines long and allows for guarantees about validity and consistency of data.

A second, more recent, solution called Hands-off Persistence System (HOPS) [11] focuses on achieving both performance and bug free persistency by providing high level ISA primitives. These primitives allow an application to separate durability and ordering constraints in programming. The HOPS framework then automatically enforces those constraints for the programmer. While this methodology is useful and proven, it requires hardware modification in order to be put in practice.

The previously discussed Mnemosyne [4] is an interface designed specifically for programming with persistent memory. Storage-Class-Memory (SCM) is a term referring to memory that provides both the interface of memory (load and store instructions) in conjunction with the persistence of disks. The purpose of Mnemosyne is to provide an interface to allow data structures to be made persistent without converting them to a serial format. It works by providing segments of virtual memory that are stored in SCM. It also supports low level operations that can consistently update data as well as durable memory transactions to enable consistent, in-place updates. Mnemosyne works using a set of small libraries that run on traditional processors. It allows programmers to designate data as critical. Critical data will then be saved to non-volatile memory. By taking a simple approach for programmers, Mnemosyne attempts to help avoid bugs, but relies on the assumption that in-flight operations are atomic rather than explicitly guaranteeing it.

A specific challenge for persistent memory consistency is encrypted memory systems. Kolli [14] seeks to lower the burden on the programmer in encrypted systems through creation of a programming model and proposal of several small hardware modifications. Data security and encryption are extremely important topics in today's world. Kolli observed that the key to maintaining crash consistency in an encrypted system is ensuring the atomicity of both data and its associated counter. This means that either both the data and the counter are made persistent, or neither are made persistent. To achieve this, they suggest what they call selective counter atomicity to achieve crash consistency and maintain performance. Their model allows for relaxed atomicity requirement that allows reordering and buffering of instruction in select windows of execution. The model allows for encryption and high performance while maintaining crash consistency guarantees in a way that reduces the burden on the programmer.

Each of these works and research are designed to improve or maintain efficiency of persistent memory writes while also making it easier for the programmer to interface with the persistent memory. They allow for guarantees when used correctly, but generally don’t take into account the effectiveness and understanding of the programmer. In scenarios where the programmer makes a mistake, testing frameworks can help find the resulting bugs.

Related Testing Frameworks

A number of different persistent memory testing frameworks have been proposed and created in recent years. Some prominent examples include crash hoare logic [9], Yat [7], and PMTest [8]. Each of these seeks to solve the same problems being addressed in this creative component using a number of different approaches. Crash hoare logic was an idea created to test the FSCQ file system. The FSCQ file system was able to certify that its implementation meets specification including crash recovery by using crash hoare logic. The primary purpose of crash hoare logic is a way for file system developers to formally certify that persistent memory operations function correctly. To do this, they certify that both preconditions and postconditions of an operation are correct. In a crash scenario, they certify by considering crash conditions, address space, and recovery execution. All operations are certified automatically in the framework, but the primary contribution is that the model considers these conditions as inputs.

Crash hoare logic is a useful certification method, but it is inflexible and largely useful only with file systems.

Yat is a trace based testing framework that functions similarly to the design of this creative component. Yat attempts to test apps for bugs caused by improper reordering of writes and then run a program's persistent memory recovery mode through a high number of scenarios to search for bugs. In order to test for improper reordering of writes, Yat reorders persistent memory writes in every order in which they could be executed to the persistent memory hardware. By doing this, Yat is able to check whether any of the potential write orders cause invalid or incorrect results. These reorders can largely be enforced by fence instructions such as sfence or mfence. When operating, Yat functions in two phases. In the first phase, Yat records a trace of an executing app within the address range of persistent memory. The trace includes writes, cflush instructions, and fence instructions. In part two, the trace is cut into segments divided by persistent memory barriers in the trace. Each segment is then reordered and replayed in every possible combination. Finally, Yat runs recovery code and an application specific verification checker. In other publications, Yat has been described as thorough but slow. The writers themselves found that Yat is effective at finding both simple and complex bugs related to write reordering and expected output. They also noted that many bugs were detectable by manual trace examination without having to use replay.

PMTest is a testing framework that takes a very different approach than YAT. PMTest's stated goal is to create a testing framework that can detect when a programmer improperly uses a library or low level primitive. They specifically focus on being flexible with multiple computer architectures and being fast in execution. To achieve this, PMTest uses a form of program annotation. It specifically adds two functions: isOrderedBefore() and isPersist(). They also implement some higher level checkers that build on the two low level checkers. In order for these to be useful, the programmer has to manually insert them into their code. The framework then uses the checkers at runtime to confirm that persistent writes programmed into the code match up with the PMTest checkers. If not, the framework will return warnings. This tool functions as a valuable sanity check that allows programmers a second method of confirming their code makes sense. The problem with this approach is that if a programmer misunderstands how to use the PMTest checkers, the framework can return false negatives or miss actual bugs entirely. The framework relies on the assumption that the programmer fully understands how to use it and where to code the checkers in relation to nonvolatile memory instructions. If a programmer fundamentally misunderstands how to use low level persistent memory instructions, they will likely also misunderstand how to use low level PMTest checkers. This framework makes complete sense in an ideal world, but fails to account for a high probability of human error when using it. If used properly, the framework provides a very fast, effective checking mechanism.

A final tool that is considered state of the art by other testing frameworks [8] is called pmemcheck [10]. It is a prototype library only available on gitHub as of this writing. Pmemcheckis built as a Valgrind tool. In order to use it, a programmer has to add a few lines of code to their program telling pmemcheck which parts of memory are persistent memory. The framework then checks for writes that were not made persistent as they were supposed to be. It also detects flushing errors and writes that were overwritten before being made persistent. One of the biggest advantages of this framework is that it is free and open source. Additionally, it requires little setup or knowledge of the underlying framework to be use correctly. One of the pmemcheck's drawbacks is that it is considered slow. It clocks in at around 20x slower than the PMTest framework.

Each of these testing frameworks attempts to solve many of the same problems as this creative component. They all take a different approach and achieve varying results. Many of them are building blocks for the approach presented herein.

Test Framework

Overview

In an attempt to provide a solution for automated persistent memory bug detection, the groundwork for a trace based test framework was developed as the primary contribution of this creative component. The framework uses concepts from the testing frameworks described above as a basis for a new automated testing framework.

Figure 3 shows the lifecycle of a test run using the persistent memory test framework. It starts by running a program being tested for persistent memory bugs underneath the memory trace generator. The trace is then fed to the memory map generator that records the last write at each virtual memory location. The memory map is then fed into the test program's recovery mechanism. Finally, a bug detection program searches for signs of mismatches or failures during the program run. The gray portions show potential future expansion in the form of iterative execution. With the addition of iterative execution, the framework will test many different memory maps based on simulated failure points at each persistent memory write in the memory trace.

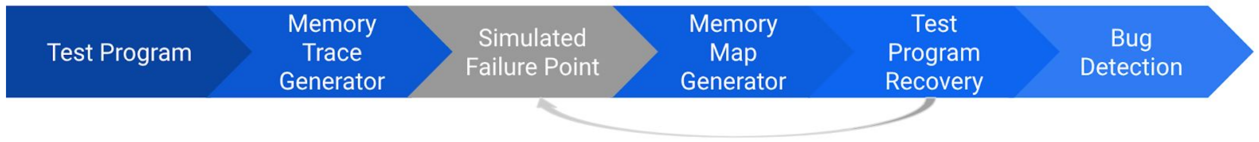


Figure 3: Test Framework Layout

The test program used during both development and testing of the framework is a simple program to write an array into nonvolatile memory using the mmap() function. It takes inputs for recovery mode and the memory mapped file and outputs the array both to stdout and as a binary file. The test program is written in C and includes a full recovery mechanism.

The memory trace generator is a tool written using Intel's PIN tool [18]. PIN instruments a program using just-in-time dynamic compiling and can be used for a number of different modifications and applications of a program. In this context, PIN is used to record the virtual address, size, instruction pointer, and value of each write transaction created by the test program. The PIN tool doesn't have any explicit inputs, but it runs around the test program. It outputs two binary files. The first contains the virtual address, size, and instruction pointer. The second contains the values of each write. The PIN tool is written in C++.

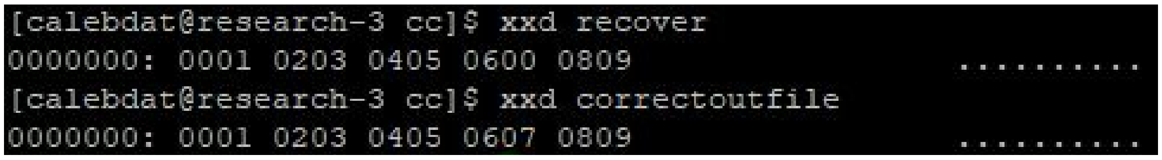
The memory map generator is a tool that takes as inputs the trace files produced by the trace file generator as well as a simulated power failure location. It then parses the trace files to produce a memory map containing the last value written into each virtual memory location before the simulated power failure.

The test program recovery mechanism is part of the original test program. It is the code that runs after a crash or power failure to try to generate the same output it would have had the interruption not occured. This is made possible by the program storing critical data in persistent memory. Without persistent memory, the program would have no information about where it failed and thus have to start over. This recovery mechanism is specific to the program being tested. It is not really possible to create a generalized solution as each program needs to recover differently using different data.

The final part of the test framework is the bug detection mechanism. This program takes as an input the test program that includes recovery. It runs that test program and its recovery code as a child process then checks for 3 different types of failures. First, it checks the output produced after the simulated power failure against the output produced by the run in which the trace was simulated. It compares the two outputs and lets the user know where any potential mismatches are. The second error the program attempts to check for is crashes and hangs. To do this, the program takes in a completion time expectancy and uses that time as an upper bound for execution time. If the recovery has not completed in the specified time, the program will kill the process and warn the user that the program likely hung. The final error the bug detection program looks for is error messages produced by the program.

Results

The testing framework was able to produce limited positive results as a proof of concept. The test program was designed to create a persistent memory array of 0-9 in the virtual address space. After a power failure, the recovery mechanism seeks the last value written to the memory mapped array and continues writing the array from that point. In the example below, a bug was intentionally put in the recovery mechanism that caused the program to continue writing the array from a point one extra number after the last successfully written number. The expected values of the persistent array can be seen below the command “xxd correctoutfile”. It shows the values 00-09 written in memory. Below the line “xxd recover” is shown the values written to memory using the intentionally bugged code. In this scenario, power failed after writing value 6 into memory. The recovery mechanism then resumed writing at value 8 and skipped value 7 entirely. This resulted in the output 0001 0203 0405 0600 0809. A clearer example of expected and actual output can be seen in table 1 below. The full code for the test program can be seen in Appendix A.



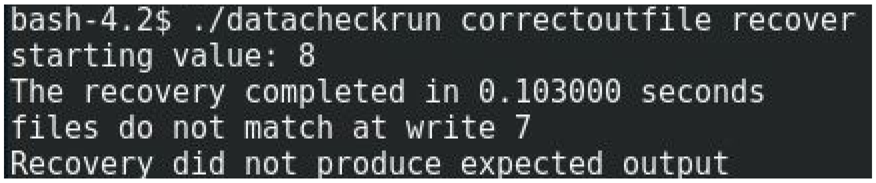
```
[calebdat@research-3 cc]$ xxd recover
0000000: 0001 0203 0405 0600 0809               ..........
[calebdat@research-3 cc]$ xxd correctoutfile
0000000: 0001 0203 0405 0607 0809               ..........
```

Figure 4: Comparison of binary persistent memory files

Expected Output	Recovery Output
0001 0203 0405 0607 0809	0001 0203 0405 0600 0809

Table 1: Expected output vs Recovery Output in recovery code error scenario

The results shown above were created as a child process of the bug detection code. The code takes as input the expected output and the name of the array file that the recovery program is writing to. It then runs the recovery process and monitors the results. In figure 5, the starting value is supposed to be the first value written to the array during recovery. In the scenario shown in figures 2 and 3, the recovery code chose the starting value as 8 rather than 7. The recovery then completes and the bug detection program notes the completion time to ensure a reasonable value. It then compares the recovered output with the expected output. In this scenario, value 7 was never written and thus the files do not match. The bug detection code noted the error and printed the location of the mismatch. In order to detect this bug naturally, the framework would likely have to repeat the recovery and test framework using many different points in the trace as a simulated failure point. This could significantly increase the run time as each test would take approximately a tenth of a second.



```
bash-4.2$ ./datacheckrun correctoutfile recover
starting value: 8
The recovery completed in 0.103000 seconds
files do not match at write 7
Recovery did not produce expected output
```

Figure 5: Output of bug detection program

This simple test program and automated bug detection scenario allow for a valid proof of concept. As noted in Yat [7], one of the most common types of persistent memory bugs is errors in the recovery code that change the expected output. This example result shows the framework successfully detecting such a bug. While this result is not comprehensive, it is a promising result for the potential validity and usefulness of the framework.

Limitations

The results and methods presented herein are a promising beginning to an automated test framework. They do, however, have some limitations. First, the results found here were intentionally placed, synthetic bugs. While these results certainly verify potential usefulness, they do not show actual real world confirmed usefulness yet. The results focus on persistent memory recovery bugs as presented in Yat [7].

The methods contained herein are currently useful, but do not contain a mechanism for repeated testing at each possible failure point. By only testing one potential location at a time, a user has to be lucky in the power failure time they chose to detect naturally occurring bugs. A more comprehensive method would be to run the test framework with a simulated failure after every write. A method that is both efficient and more comprehensive would be to run the framework with a simulated failure after every persistent memory write. These limitations do not preclude the results from being useful, but should be examined with some measure of caution.

Future Work

Although this paper outlines the beginnings of a framework to detect memory consistency bugs, there is still a lot of work that needs to be done for it to be a fully fleshed out, useful system. One of the first important steps is to make the framework less application specific. As a proof of concept and building block, the framework achieves its goal of finding and squashing a memory persistency bug. In order to achieve that in a short time, however, certain methods were programmed with the test application in mind rather than for a broad set of use cases. Most of these could be generalized fairly easily to allow for a much wider set of usage scenarios for the toolset.

A second area of future expansion would be further testing on additional programs and scenarios. For this creative component, the testing was largely done with a simple test program specifically designed for this framework. For further proof of concept and bug detection, a wider selection of persistent memory aware programs could be used. Specifically, a persistency bug has been manually identified in the sql program N-Store. Using the framework to recreate and automatically detect the same bug would go a long way towards ensuring the functionality in real-world scenarios.

A third area of future expansion would be targeting only memory that is persistent. The current system creates and parses a trace file into a memory map file. It does not treat memory that is volatile and memory that is nonvolatile differently. One way to reduce runtime overhead is to target only persistent memory operations. Writes that are between two persistent memory locations essentially do not change the results of a recovery operation. By targeting only persistent memory writes, we would be achieving two things. First, targeting persistent memory transactions allows us to target the transactions that are most likely to cause persistency bugs. Second, by removing a significant portion of transactions from consideration, we can reduce the framework runtime.

A fourth area of future work is the addition of a repetition framework. Unless a user gets lucky, a single run on a single trace is not very useful for many programs. Programs often rely on sets of inputs that change and modify how the program runs. A single trace is unlikely to cover all possible scenarios and thus likely to miss potential bugs. A potential solution is an automated repetition process based on a set of predefined test program inputs. This would allow the test framework to cover a much wider set of scenarios that would result in much more comprehensive and complete bug detection tests.

A final area of future expansion is further honing and developing of consistency bug detection rules and methods. The framework currently tests for basically three things. First, it tests whether the recovered output matches the expected output. Second, it tests whether the recovery completed in a "reasonable" time frame. Finally, it checks whether the program produced any error messages. The idea of a "reasonable" timeframe is currently a subjective input by the framework user. It is not scientific or based on any kind of research. Further research and development into detection of stalls or hangs would allow for a much more scientific detection method. Additionally, research could be done into the best way to determine how long to consider a "reasonable" run time is when accounting for runtime differences from system variation and calls. Ideally, a model could be created to determine a time for each program that minimizes both false positives and false negatives. Further, detection methods have been identified for finding potentially nonatomic writes in a trace. For example, pmemcheck uses the pattern store->flush->sfence->pcommit->sfence to check for atomicity of persistent memory writes [10]. Similar patterns could be implemented as checks into the framework for persistent memory writes.

Conclusion

Persistent memory is nonvolatile memory that can be used to backup critical data through power failure scenarios. The use of persistent memory comes with a new set of programming challenges and potential bugs. Examples of bugs include recovery code errors, nonatomic persistent memory writes, and improper reordering of writes. To help combat these concerns, researchers have developed libraries, file systems, and programming models. In addition, several sets of research proposed and built test frameworks.

The primary contribution of this creative component is the design and programming of a basic test framework with room for expansion. The test framework is a trace-based dynamic testing framework that currently primarily tests for recovery code bugs. It checks for error codes, hangs, and mismatches in expected output. Early tests showed promising results as the framework was able to detect synthetically placed bugs that led to hangs and output mismatches. In the future, this framework could be expanded to use more complex bug checking models and allow for rapid repetition. The current framework helps fill a hole in the test framework space for a fast trace based testing framework and lays the promising foundation of a full-featured test framework.

References

1. M. Wu and W. Zwaenepoel, “eNVy,” *ACM SIGPLAN Notices*, vol. 29, no. 11, pp. 86–97, 1994.
2. M. Satyanarayanan, H. H. Mashburn, P. Kumar, D. C. Steere, and J. J. Kistler, “Lightweight recoverable virtual memory,” *ACM Transactions on Computer Systems*, vol. 12, no. 1, pp. 33–57, 1994.
3. S. Swanson, “Early Measurements of Intel’s 3DXPoint Persistent Memory DIMMs,” *ACM SIGARCH*, 15-Apr-2019. .
4. H. Volos, A. J. Tack, and M. M. Swift, “Mnemosyne,” *ACM SIGPLAN Notices*, vol. 47, no. 4, p. 91, 2012.
5. R. Ramakrishnan and J. Gehrke, *Database management systems*. Brantford, Ontario: W. Ross MacDonald School Resource Services Library, 2017.
6. J. Xu and S. Swanson, “14th USENIX Conference,” in *NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories*. ISBN 978-1-931971-28-7
7. P. Lantz, S. Dulloor, S. Kumar, R. Sankaran, J. Jackson, and Intel Labs, “USENIX ATC '14,” in *Yat: A Validation Framework for Persistent Memory Software*. 978-1-931971-10-2
8. S. Liu, Y. Wei, J. Zhao, A. Kolli, and S. Khan, “PMTest: A Fast and Flexible Testing Framework for Persistent Memory Programs,” *ASPLOS '19*, Apr. 2019.
9. H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. Zeldovich, “Using Crash Hoare logic for certifying the FSCQ file system,” *Proceedings of the 25th Symposium on Operating Systems Principles - SOSP 15*, 2015.
10. PMDK. An introduction to pmemcheck. http://pmem.io/2015/07/17/pmemcheck-basic.html, 2015
11. S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton, “An Analysis of Persistent Memory Use with WHISPER,” *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS 17*, 2017.
12. M. Satyanarayanan, H. H. Mashburn, P. Kumar, D. C. Steere, and J. J. Kistler, “Lightweight recoverable virtual memory,” *ACM Transactions on Computer Systems*, vol. 12, no. 1, pp. 33–57, 1994.
13. A. Kolli, J. Rosen, S. Diestelhorst, A. Saidi, S. Pelley, S. Liu, P. M. Chen, and T. F. Wenisch, “Delegated persist ordering,” *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

14. S. Liu, A. Kolli, J. Ren, and S. Khan, “Crash Consistency in Encrypted Non-volatile Main Memory Systems,” *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018.
15. V. Chidambaram, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, “Optimistic crash consistency,” *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles - SOSP 13*, 2013.
16. “Phase-Change Memory (PCM),” *ST*. [Online]. Available: https://www.st.com/content/st_com/en/about/innovation---technology/PCM.html. [Accessed: 22-Apr-2019].
17. “STT-MRAM: Introduction and market status,” *MRAM-info*, 19-Feb-2019. [Online]. Available: https://www.mram-info.com/stt-mram. [Accessed: 22-Apr-2019].
18. MAD\losnat, “Pin - A Dynamic Binary Instrumentation Tool,” *Intel® Software*, 25-Jun-2018. [Online]. Available: https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool. [Accessed: 22-Apr-2019].

Appendix A

```c
#include<string.h>
#include<stdio.h>
#include<stdlib.h>
#include<sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/types.h>

#define NUMINTS (10)
#define FILESIZE (NUMINTS * sizeof(char))
void err_quit (char *msg)
{
        printf(msg);
        return;
}

int main(int argc, char* argv[]){
        int fdin, fdout;
        char *src, *dst;
        struct stat statbuf;
        int mode = 0x0777;
        char *map;
        int startingi = 0;
        int last = 0;
        //int baseval = 1354;
        char baseval = 0;

        if (argc!=3)
                err_quit("usage: a.out <recovermode> <tofile> <recoverfile>\n");

        int inputarg = atoi(argv[1]);
        if(inputarg == 1){ //if 1, assume power failure and enter recovery
                /*open input file*/
                if ((fdin = open(argv[2], O_RDWR)) < 0)
                {
                        printf("can't open %s for reading\n", argv[1]);
                        return 0;
                }

                //parse input file looking for write values
                char parsedval;
                unsigned char *f;
                struct stat s;
                int j = 0;
                int size;
                int status = fstat (fdin, & s);
                size = s.st_size;
                 f = mmap (0, FILESIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fdin, 0);
                for(j=0;j<size;j++){
                        parsedval = f[j];
                        if(parsedval == baseval){
                                last = baseval;
                                baseval++;
                        }
                }
                if(last<NUMINTS){
                        startingi = last + 1;
                        printf("starting value: %d\n",startingi);
                        for(startingi;startingi<size;startingi++){
                                f[startingi]=startingi;
                        }
                }else{
                        startingi = NUMINTS;
                }
        }else{
                startingi = 0;

        /*open/create the output file*/
        if ((fdout = open (argv[2], O_RDWR | O_CREAT | O_TRUNC, (mode_t)0600)) <0){
                printf("can't create %s for writing\n", argv[2]);
                return 0;
        }

        /* go to location corresponding to last byte */
        if(lseek (fdout, FILESIZE-1, SEEK_SET) == -1)
        {
                printf("lseek error\n");
                return 0;
        }

        /*write dummy byte at last location */
        if (write(fdout, "", 1) != 1){
                printf("write error\n");
                return 0;
        }

        map = mmap(0, FILESIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fdout, 0);
        if (map == MAP_FAILED){
                close(fdout);
                printf("Error mapping file\n");
                return 0;
        }
        char i = 0;
        for (i=startingi; i<=NUMINTS;i++){
                map[i]=i;
                printf("%d ", map[i]);
        }
        printf("\n");
        if(munmap(map, FILESIZE) == -1){
                perror("Error unmapping the file");
        }

        close(fdout);
        }
}
```