Crash-Consistency Persistent Memory Bug Repair

Yile Gu
University of Michigan
yilegu@umich.edu

Shenyi Wang
University of Michigan
wshenyi@umich.edu

Abstract

Persistent memory (PM) is a promising emerging hardware that has better latency than flash SSDs and cheaper cost than DRAM. Despite its powerfulness, it is still challenging to develop programs under PM due to the presence of crash-consistency bugs.

In this project, we propose a new crash-consistency bug repair tool for PM applications. We utilize bug traces from crash-consistency bug detection tool Squint [15] to generate fixes. The bug fixes are dependent on the transaction mechanism provided by Persistent Memory Development Kit (PMDK). In terms of fix generation, we first analyze traces returned from Squint to propose valid intervals for transaction wrapper. Then we insert transaction via source-to-source transformation. Evaluations show that our tool is able to fix non-trivial crash-consistency bugs with minimized manual efforts.

1 Introduction

In recent years, the increasing gap between the performance of CPU cache and DRAM memory has triggered the development of a new hardware: persistent memory (PM). PM's access speed sits between cache and traditional memory, which leads to its duality characteristic: either as memory or persistent storage. However, PM requires application developers to explicitly perform flush and memory fence operations, which may lead to crash-consistency bugs due to carelessness. Crash consistency bugs are problematic for crash-consistent applications, as such bugs can cause data loss or data corruption. As a common practice, crash consistency bugs are mainly fixed manually, which takes much time and human power, since crash-consistency bugs are often hard to reason about.

So far, crash-consistency bugs can be divided into two categories. One is durability bugs (also called missing flush/fence bugs). Such bugs are caused by an omission of a cache-line flush or memory fence on first-generation PM platforms. The other category of crash-consistency bugs is application-specific bugs, which are caused by improper ordering between updates to semantically-related PM data and arise regardless of whether applications include appropriate PM ordering instructions.

To tackle durability bugs, previous work Hippocrates [14] was designed to find and automatically fix PM durability bugs, and many other works can also fix such bugs. However, there are limited works focus on application-specific bugs. Prior work Squint [15] is designed to find application-specific bugs but can't fix them. Our project aims to add repair mechanism for Squint to help it automatically fix application-specific bugs. The core idea of our project is to use transaction mechanism provided by PMDK to form transaction blocks that can be rolled-back. The way we specify block region is to use the event trace provided by Squint to help locate interval ranges of inconsistency bugs and insert transaction macro between intervals to keep them consistent.

Our project is implemented in two parts. The algorithm part is to get temporal insertion position based on Squint and the insertion engine part uses Clang front-end to finish source-to-source transformation by inserting final transaction macro. We evaluate our project on test programs to show that it is able to generate bug fixes correctly and efficiently, and the overhead added to programs is minor.

In the rest of this paper, we first provide background on persistent memory, PMDK, and Squint. Then we describe the details of our interval generation algorithm and transaction macro insertion method. Next, we provide high-level implementation of our project. After that we evaluate our project on chosen test programs. Finally, we introduce related works in crash-consistency bugs repair, discuss limitations and future work, and conclude our project.

2 Background

2.1 Persistent Memory Background

Persistent memory (PM) is a novel hardware which allows programs to access data as memory and is directly byte-addressable, where the contents are non-volatile, preserved across power cycles. PM is used in conjunction with memory and storage. Systems containing persistent memory can outperform legacy configurations, providing faster start-up times, faster access to large in-memory datasets, and often improved total cost of ownership.

However, programming PM applications is error-prone. In order to persist data that are cached in volatile CPU, developers need to manually *flush* cache lines to guarantee that updates reach PM. Moreover, flushing cache line is a weakly order process, so developers must use memory *fences* to regulate the ordering of updates to persistent memory.

PMDK. The Persistent Memory Development Kit (PMDK) is a suite of open-source libraries that have been developed for various use cases. Each library is tuned, validated to production quality, and thoroughly documented. These libraries provide stable APIs so developers can rapidly implement PM features into applications without worrying about the hardware implementation details or generational changes.

PMDK provides a powerful mechanism for handling crash-consistency issues: *transaction*. Transactions wrap persistent memory addresses into blocks for protection and backup values in selected blocks in case of rollback action. Transaction operations here is the same mechanism people usually used in database operation. if a program crashes during processing a transaction block, transaction mechanism will roll back the program to the state before processing this block. Figure 1 provides the lifecycle of transaction mechanism defined in PMDK.

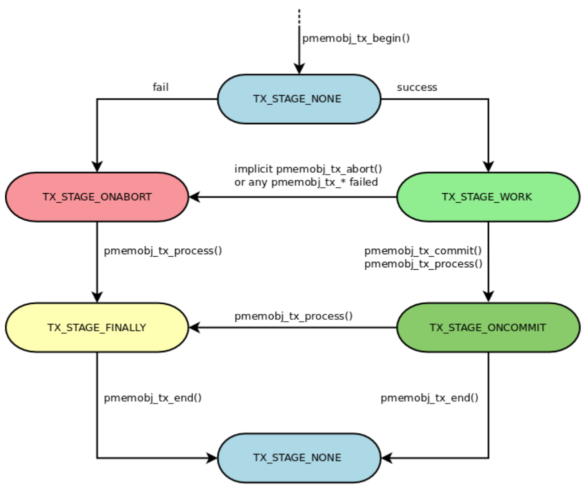

Figure 1: The lifecycle of PMDK transaction mechanism from [17]

2.2 Crash-Consistency Bugs in PM

The weakly ordered nature of updates from volatile memory to persistent memory may cause data inconsistency. Before data become persistent on PM, they're still volatile on CPU. If crashes happen before or during data reaching persistent memory, those data which should have become persistent are still volatile due to weakly ordered updates. These kinds of problems are called crash-consistency bugs and they can be fixed by correctly using flush/fence primitives. There are two categories of PM crash-consistency bugs:

(1)Durability Bugs. Such bugs are also called *missing flush/fence bugs*. They arise due to an omission of a cache-line flush or memory fence on first generation of PM platform. The main idea of fixing durability bugs is intuitive – inserting those missing flush/fence. For example, Hippocrates [14] can automatically detect and fix missing flush/fence without harming program's performance. However, PM platforms equipped with eADR can automatically fix missing flushes and we will see the impact of durability bugs decrease as PM hardware continues to improve.

(2)Application-specific Bugs. Application-specific bugs are caused by semantically-related PM data which are updated in an improper ordering. Here the proper ordering is application-specific, which means the "correct" ordering derives from semantics that embedded in specified programs.

Listing 1 illustrates an example bug in this class. Even though we call *PERSISTENT* operation for every updates, crash happen at line **5** still lead to a crash-consistency bug. The correct ordering here is first update *array_size* and then allocate new array with new size, Otherwise, if crash happens at line **5**, an out of index illegal memory access may happen after recover, since the real size of *array* is inconsistent with the recording in *array_size*.

```
1 foo(){
2     delete my_structure->array;
3     my_structure->array = new int[new_sz];
4     PERSISTENT(my_structure->array);
5     // Bug: crash before size is updated
6     my_structure->array_size = new_sz;
7     PERSISTENT(my_structure->array_size);
8 }
```

Listing 1: An example of application-specific bug.

2.3 Squint: Crash-Consistency Bug Detection

To help software developers better detect crash-consistency bugs in PM applications, various PM testing tools have been proposed in recent years. One of the major problems of such tools is that it is difficult to achieve high test accuracy efficiently. As a result, Squint [15] is designed to tackle this problem.

Squint is a robust PM crash-consistency testing tool that scales PM model checking to large PM applications. Squint identifies update mechanisms of the application under test by identifying ordering constraints between PM updates and constructing dependency graphs that obey the causality of updates. To prevent the ordering explosion problem, Squint reduces update patterns by grouping them into representative update mechanisms, which allows the tool to scale to large applications while having good testing coverage.

3 Design

3.1 Bug Repair Pipeline Overview

Figure 2 summarizes the overall pipeline of our crash-consistency bug repair tool. It has three main components: bug detector, fixer and verifier. The bug detector executes target program and generates pairs of trace events and crash-consistency labels indicating whether

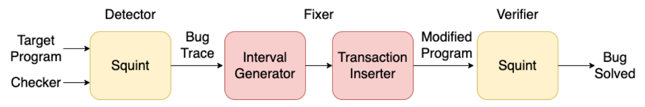

Figure 2: The overall architecture of our bug repair tool

the program under test is buggy. The fixer digests this information and produces fixes that modify the source code program directly. The modified program is compiled again and fed into the verifier to test if the existing crash-consistency bug has been successfully fixed.

In terms of inputs, the users will provide a target program to test crash-consistency and a checker program for detecting crash-consistency violations. If the target program contains crash-consistency bugs, our bug repair tool outputs a modified source code program that has been cross-checked by the verifier to be crash-consistency bug-free. If not, the program remains unmodified.

3.2 Bug Detector and Verifier

Given a target program and a checker program, we invoke Squint as our bug detector. Squint first runs pmemcheck [17] on the target program to generate a trace of PM operations which contain stores, cache-line flushes and memory fences. Then Squint converts the trace into *persistence graphs* that reflect PM updates and their corresponding orderings. The graphs are optimized into representative subgraphs by grouping PM updates from the same data type and removing redundant graphs. Finally Squint invokes pmreorder [10] to perform exhaustive model checking on these optimized subgraphs.

Squint outputs ranges of trace events that it has tested all of the possible orderings on, and the corresponding crash-consistency labels. The fixer consumes these outputs and produces bug fixes on the target program. After that, we again invoke Squint as the verifier to ensure that crash-consistency bugs in the program have been fixed successfully.

3.3 Transaction Interval Generation

In terms of transaction lifecycle shown in Figure 1, for simplicity, our transaction created by the fixer will only contain the *TX_STAGE_WORK* block. This is because that other blocks such as *TX_STAGE_ONABORT* are

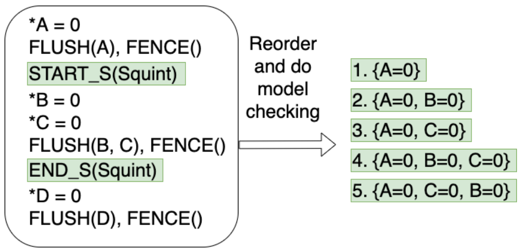

Figure 3: A code snippet that Squint tests on, modified from [15]

usually designed for either logging or bookkeeping in the case of crash failures. Such bookkeeping operations need auxiliary information about variables defined in the program and cannot be directly inferred from the trace records of flushes, fences and stores.

3.3.1 Difference between Reorder Interval and Transaction Interval

To explain how to define an accurate interval for transaction, let's start by looking at an example. Figure 3 shows a code snippet that Squint tests on. In this case, Squint has reduced one *persistence graph* into a representative subgraph which contains two updates $*B = 0$ and $*C = 0$. Squint inserts *START_S* and *END_S* markers indicating the range to perform exhaustive reordering by pmemreorder. Let the range between these two markers be reorder interval, and let the range generated by the fixer be transaction interval defined by *START_Tx* and *END_Tx* markers. One of the key findings is that the reorder interval and the transaction interval may be different.

To see this difference, we will walk through the process how pmemreorder detects crash-consistency bugs. At first, pmemreorder persists all states before *START_S* marker, and generates ordering by ordered set enumeration to test on. In our example, there will be 5 possible orderings between *START_S* and *END_S*.

If any of the 5 possible orderings generates a crash state that is not consistent, then Squint will mark this interval containing updates on B, C as inconsistent. Now if we consider inserting transaction to solve this crash-consistency bug, we have to define the locations for markers $START\_Tx$ and END_Tx. A trivial solution will be to place the two markers at exactly where $START\_S$ and END_S are.

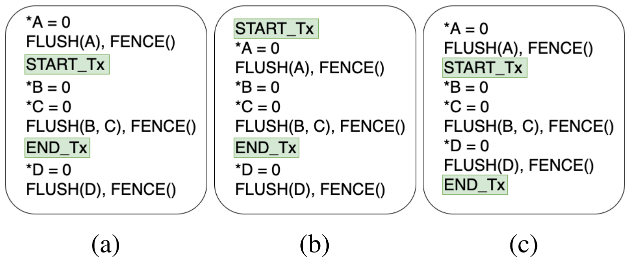

Figure 4: Three possible situations for transaction markers (a) within inconsistent interval (b) before inconsistent interval (c) after inconsistent interval

Define the notation $\rightarrow$ to reflect the causality relation "happen before" [6]. For example, $X \rightarrow Y$ means X happens before Y. Now if variables B and C possess causality relation either $B \rightarrow C$ or $C \rightarrow B$, then our trivial solution will work. This is because that the causality relation will cause either $\{A = 0, C = 0\}$ or $\{A = 0, B = 0\}$ to be inconsistent. However, after the transaction insertion, the 5 possible orderings are reduced to only 3: $\{A = 0\}$, $\{A = 0, B = 0, C = 0\}$ and $\{A = 0, C = 0, B = 0\}$. All of the 3 orderings are consistent and the crash-consistency bug is solved.

However, the situation gets more complicated if the causality relation is, for example, $B \rightarrow A || C$. In this case, putting a transaction around updates on B and C will not work, because one of the 3 possible ordering left, $\{A = 0\}$, will still be inconsistent after the transaction is inserted. To solve this specific crash-consistency bug, a $START\_Tx$ marker must be placed before update on variable A. Similarly, if the causality relation is $D \rightarrow B || C$, then an END_Tx marker must be placed after update on variable D.

Figure 4 summarizes the three possible situations to locate markers for transaction intervals. Given an inconsistent interval returned from Squint, the transaction may be inserted directly at the inconsistent interval, the $START\_Tx$ marker may be before $START\_S$ marker, or the END_Tx marker may be after END_S marker.

3.3.2 Locating Start and End of the Transaction

To properly define the starting point and ending point of the transaction, we have to make use of more information from the trace events. The key insight is that if an interval is inconsistent, we have to find a starting point such that all of the updates before it have been tested to be consistent and an ending point such that all of the updates after it have been tested to be consistent.

Formally, define a trace of length n to be $T = [te_0, te_1, ..., te_{n-1}]$, where te_i indicates the trace event at timestamp i. Each trace event may be a flush, fence or store event. The trace returned from pmemcheck during the test will be a global sequence containing all of the possible trace events, denote it as T_{global}. Squint's partial model checking can be considered as running consecutive subsequence of the global trace T_{global}. Let the subsequence be $T_s = [te_i, te_{i+1}, ..., te_{i+m-1}]$, and let the corresponding crash-consistency result returned from pmreorder be y, then given a global trace T_{global}, output res from Squint will be a set of subsequence traces with crash-consistency labels: $res = [(T_s^0, y_0), (T_s^1, y_1), ...(T_s^n, y_n)]$.

If there exists a subsequence $T_s^k = [te_k, te_{k+1}, ..., te_{k+t-1}]$ such that label y_k indicates crash-inconsistency, our fixer will be initiated. To find an appropriate starting point for transaction interval, ideally, we want to find a trace event te_i such that $i \leq k$ and trace $T_i = [te_0, te_1, ...te_i]$ is consistent. However, finding this starting point requires us to traverse backward linearly. And each time pmemreorder tests crash-consistency on this given trace, there is an exponential time complexity that hinders performance greatly.

3.3.3 Heuristic Algorithm for Transaction Interval Finding

To tackle this performance issue, we introduce a heuristic algorithm to help speed up the generation process. Define $\subseteq$ to be the envelope operation. If there exist two subsequences of traces $T_m = [te_m, te_{m+1}, ...te_{m+i-1}]$ and $T_n = [te_n, te_{n+1}, ...te_{n+j-1}]$, such that te_m happens at least as early as te_n (i.e. $m \leq n$) and te_{n+j-1} happens at least as early as te_{m+i-1} (i.e. $n + j - 1 \leq m + i - 1$), then T_n is enveloped by T_m, i.e. $T_n \subseteq T_m$.

Since Squint has tested many instances of subsequences of global trace, we could utilize this information without incurring additional pmreorder testing. Given an inconsistent interval $T_f = [te_f, te_{f+1}, ..., te_{f+i-1}]$, we want to find a consistent tuple in Squint's result $(T_c = [te_c, te_{c+1}, ..., te_{c+j-1}], y_c) \in res$ such that T_c is tested strictly before T_f (i.e. $c + j - 1 \leq f$) and the ending point of T_c is as late as possible (i.e. $c + j - 1$ as large as possible). To do so, define the subsequence from timestamp 0 to f (head of the inconsistent interval) to be $T_{0-f} = [te_0, te_1, ..., te_f]$, we first find the candidate set of subsequence traces $S = [T_c^0, T_c^1, ...T_c^k]$ where each subsequence is enveloped by T_{0-f} (i.e. $T_c^k \subseteq T_{0-f}$), then we choose the subsequence in candidate set with the largest ending point timestamp. This ending point of trace event will be used as the starting point for our transaction insertion, denote it as te_{start}, because we know to the best our knowledge that until this point the program is still crash-consistent and the crash-consistency bug is not involved yet.

We use the similar approach to find the ending point for our transaction interval. Suppose global trace has n events, then define the subsequence from tail of the inconsistent interval timestamp $f + i - 1$ to $n - 1$ to be $T_{f-end} = [te_{f+i-1}, te_{f+i}, ..., te_{n-1}]$. We find the candidate set $E = [T_E^0, T_E^1, ...T_E^k]$ where each subsequence is enveloped by T_{f-end} (i.e. $T_E^k \subseteq T_{f-end}$). Then we choose the smallest starting point within the candidate set to be the ending point of our transaction interval, denoted as te_{end}. This is again because we know to the best our knowledge that until this point the program has returned to be crash-consistent, and the crash-consistency bug has been all involved.

We pair (te_{start}, te_{end}) as the proposal of interval for transaction insertion. The stack information in the trace events contains corresponding source file and line that will be utilized to insert transaction.

3.3.4 Transaction Interval Reduction

One of the optimizations for transaction interval generation is that it is possible to further shrink down length of our proposal. This is because that Squint constructs test cases that are separated by data types, which may not be the minimized interval of inconsistency.

For example, consider a proposal of transaction interval (i.e. a specific subsequence of trace events) $T_m = [te_m, te_{m+1}, ...te_{m+i-1}]$. If we want to make the starting point of this interval more accurate, we could generate set S of test cases linearly from the original starting point: $S = [(te_m), (te_m, te_{m+1}), (te_m, te_{m+1}, te_{m+2}), ...,$ where each element in the set is a subsequence of T_m that involves the original starting point. We then invoke pmreorder to test all cases in the set. To optimize the starting point, we pick the longest test case in set S that is consistent, say $(te_m, te_{m+1}, ..., te_{m+k})$, and assign the trace event with greatest timestamp te_{m+k} to be the new starting point. It is safe to do so because we know that until te_{m+k} the program is still crash-consistent. The similar approach can be taken to optimize the ending point.

3.4 Transaction Insertion

The process of transaction insertion is a source-to-source transformation by inserting PMDK transaction macro into source file. So, we use Clang [7] front-end as our tool to analyze abstract syntax tree (AST) of source file and transform source file to a modified version. First, we use temporal positions obtained from Section 3.3.4 to approximately locate the position to insert transaction macro. Next, we need to modify the inserting position to an appropriate place based on AST context in order to keep the correctness of programs' semantics.

3.4.1 Assumption

Based on our observation, intervals usually just span multiple lines and are usually included in the same function scope. It's rational to induce this observation to the following assumptions:

- *START_Tx* and *END_Tx* that are used to indicate transaction blocks will not appear in different files.
- In order to specific rollback regions of transaction operation, *START_Tx* and *END_Tx* will not appear in different stack levels of function calls.

With these two assumptions, we could consider that *START_Tx* and *END_Tx* will only appear in the same function scope.

3.4.2 Transaction Macro in PMDK

As mentioned in Section 3.3, PMDK provides transaction mechanism that wraps program code into transaction block by predefined transaction macro. For simplification, we only use macro *TX_BEGIN* and *TX_END*, and they are regarded as final insertion position. *START_Tx* and *END_Tx* are regarded as temporal insertion position respectively. We derive final insertion position through temporal insertion position. The derived relations are shown as below.

After determining final insertion position, the next is to insert transaction macro into source file. Listing 2 illustrates insertion process by pseudo code.

```
TX_BEGIN(pop){
    specify_memory_region_for_rollback();
    /* transaction block */
}TX_END
```

Listing 2: Pseudo code for usage of transaction macro.

3.4.3 Correctness of Semantics

Since we insert macro *TX_BEGIN* and *TX_END* to specific transaction blocks, we need to avoid semantic errors after insertion. Inserting macros in different scope will cause semantic errors because insertion may destroy original semantic structure, leading to compiling errors.

Given *START_Tx* and *END_Tx*, we use Clang front-end to analyze local AST structure on temporal insertion position and derive final insertion position.

Listing 3 illustrates an error of inserting macro in different program scope.

```
foo(){
  // Suitable position for TX\_BEGIN
  if(/* condition */){
    TX_BEGIN(pop){
    specify_memory_region_for_rollback();
    /* code */
  }
  }TX_END
}
```

Listing 3: An error for macro insertion.

4 Implementation

We implement our project as two part. The first part is the algorithm to get temporal insertion position and the second part is the insertion engine to finish transaction macro insertion. The algorithm part was implemented in ≈ 300 lines of Python and the insertion engine part was implemented in ≈ 500 lines of C++. The algorithm part uses Squint [15] to generate PM trace, during this process it utilizes pmemreorder to reduce temporal insertion region. The engine part uses Clang front-end to perform AST analysis.

5 Evaluation

The evaluation of our bug repair tool is carried out in mainly three dimensions: correctness, performance and efficiency. For correctness, we use our own test examples and test cases from PMDK to see if transaction insertion

Test Case	Before	After
Out of Order	5 bugs	1* bug
Array Alloc (PMDK v1.8)	11 bugs	0 bug
Array Realloc (PMDK v1.8)	1 bug	0 bug
Array Free (PMDK v1.8)	5 bugs	5 bugs

Table 1: Correctness evaluation of the target program after transaction insertion

Test Case	Before	After
Out of Order	0.041s	0.049s
Array Alloc (PMDK v1.8)	0.064s	0.075s
Array Realloc (PMDK v1.8)	0.064s	0.065s
Array Free (PMDK v1.8)	0.067s	0.069s

Table 2: Performance comparison of the target program before and after transaction insertion

is able to fix crash-consistency bugs. For performance, we test the runtime overhead of adding transaction to the program. For efficiency, we evaluate how much time the tool needs to generate the intervals and fix the bugs.

5.1 Correctness

We design a crash-consistency bug test case called "Out of Order", which is referenced from [19]. In this test case, a counter for a customized structure is updated before the structure itself is updated, creating a crash-inconsistent image if the crash happens between the two updates. To demonstrate the generalizability of our tool, we also pick test cases for "Array" structure from PMDK. We run Squint on the original programs and the modified programs after transaction insertion.

Table 5.1 shows the comparison of number of bugs detected by Squint. In test cases "Out of Order", "Array Alloc" and "Array Realloc", our bug repair tool successfully solves crash-consistency bugs by inserting transactions. Although in test case "Out of Order" there still lefts 1 bug detected by Squint after our bug fix, we investigate the bug information generated by Squint and find out that it is due to a crash happened at PM file creation, which is technically not a crash-consistency bug we are interested in.

For test case "Array Free", our repair tool does not generate any transaction interval despite the presence of bugs. We take a look at the bug information and find out that stack traces for these bugs lead to library files in PMDK. However, our tool currently only supports generating bug fixes in the files specified by users.

5.2 Performance

Next we evaluate how much runtime overhead transactions will add to the program. For each test case we have used in Section 5.1, we measure the execution time of the original program and the modified program after inserting transactions.

Table 2 summarizes the performance comparison between two versions of programs. The transactions added produce minor execution overhead for the target programs. In the best case , for test case "Array Realloc", the execution time for modified program takes longer than the original by 1.6%. In the worst case, for test case "Array Alloc", the execution time for modified program takes longer than the original by 19.5%.

5.3 Efficiency

Finally we evaluate how much time it takes for our repair tool to generate transaction intervals and fix the program. For each test case used in 5.1, we measure the process time of using heuristic algorithm to generate intervals, reducing intervals via pmreorder and inserting transactions to the program separately. Table 3 summarizes the process time of each submodule.

We observe from the results that reduction phase takes the longest process time. Compared to reduction, the time it takes to generate intervals and insert transactions is negligible. This is expected because pmreorder uses exhaustive model checking to emulate all possible orderings within the given range. A possible optimization is to provide an option for the users indicating whether or not to skip reduction process, because reduction on intervals only affects performance of our bug fix but not the correctness.

6 Related Works

Bug detection. The research community has proposed many systems on application testing and bug detection [1–3, 12, 13]. PMTest [11] is a trace-validation framework, where each PM operation produces a trace

Test Case	Heuristic	Reduction	Insertion	Total
Out of Order	0.057s	6m32s	0.124s	6m32s
Array Alloc (PMDK v1.8)	0.011s	10m50s	0.357s	10m50s
Array Realloc (PMDK v1.8)	0.013s	3m41s	0.103s	3m41s
Array Free (PMDK v1.8)	N/A	N/A	N/A	N/A

Table 3: Execution time comparison for each stage of our algorithm

event which is asynchronously validated to detect a durability bug. Some other tools are based on binary instrumentation. Pmemcheck [17] is a binary instrumentation tool designed by Intel for PMDK, which is based on valgrind. Agamotto [16] is a generic and extensible system for discovering misuse of PM in PM applications. Squint [15] detects application-specific bugs in PM using model checking on update mechanisms that are used by applications.

Bug repair. Many works have been done on automated bug repair [4, 5, 8, 9, 18]. They either target at performing general-purpose repair or solving specific issues such as concurrency bugs. Hippocrates [14] is an automated PM durability bug fixing system with no harm to program after repair. Hippocrates produces fixes that are functionally equivalent to developer fixes.

7 Discussion

There still exist several drawbacks for our bug repair tool. Currently our tool only fixes crash-consistency bugs via source code modification, supporting C/C++ languages. This limits the types of programs our tool is able to solve. What is more, it is difficult to combine durability bugs fixed by Hippocrates with our fixes because Hippocrates generates fixes on Intermediate Representation (IR). A solution to make our tool more programming-language agnostic is to also insert transactions at IR by LLVM Pass. However, this is challenging because we are not aware of how transaction mechanism provided by PMDK is translated into IR during compiling.

It is also possible that the starting and ending points of our transaction interval are in different functions and even different files. In our bug repair tool, we use stack information to find an optimized fix that lies in the same function. If no such fix is found, we skip the interval generation as the candidate fixes may affect correctness of the program. This is partly because that Squint generates test cases by data types, so consistent intervals may be far from an inconsistent interval. One solution is that we use inconsistent interval as the transaction interval when it contains partially crash-consistent test cases. However, the bug fixes we generate may not be useful in this case.

8 Conclusion

In this project, we propose a new crash-consistency bug repair tool for PM applications. We take advantage of both trace generated from Squint and transaction mechanism provided by PMDK. We design algorithm to determine the interval for transaction insertion efficiently, and use source-to-source transformation to repair the program. We evaluate our tool on crash-consistency bug test cases to demonstrate its effectiveness. We believe that our bug repair tool could help advance the development of comprehensive PM bug fixing toolchain.

References

[1] Recon: Verifying file system consistency at runtime. In *10th USENIX Conference on File and Storage Technologies (FAST 12)*, San Jose, CA, February 2012. USENIX Association.

[2] Daniel Fryer, Dai Qin, Jack Sun, Kah Wai Lee, Angela Demke Brown, and Ashvin Goel. Checking the integrity of transactional mechanisms. In *12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 295–308, Santa Clara, CA, February 2014. USENIX Association.

[3] Travis Hance, Andrea Lattuada, Chris Hawblitzel, Jon Howell, Rob Johnson, and Bryan Parno. Storage systems are distributed systems (so verify them that Way!). In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI*

20), pages 99–115. USENIX Association, November 2020.

[4] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. Automated atomicity-violation fixing. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, page 389–400, New York, NY, USA, 2011. Association for Computing Machinery.

[5] Guoliang Jin, Wei Zhang, and Dongdong Deng. Automated Concurrency-Bug fixing. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 221–236, Hollywood, CA, October 2012. USENIX Association.

[6] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, jul 1978.

[7] Chris Lattner. Clang: a c language family frontend for llvm. September 26, 2007.

[8] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 3–13, 2012.

[9] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, 2012.

[10] Weronika Lewandowska. Pmreorder basics. Feb 2015.

[11] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan. Pmtest: A fast and flexible testing framework for persistent memory programs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 411–425, New York, NY, USA, 2019. Association for Computing Machinery.

[12] Ashlie Martinez and Vijay Chidambaram. CrashMonkey: A framework to automatically test FileSystem crash consistency. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, Santa Clara, CA, July 2017. USENIX Association.

[13] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. Finding Crash-Consistency bugs with bounded Black-Box crash testing. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 33–50, Carlsbad, CA, October 2018. USENIX Association.

[14] Ian Neal, Andrew Quinn, and Baris Kasikci. Hippocrates: Healing persistent memory bugs without doing any harm. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, page 401–414, New York, NY, USA, 2021. Association for Computing Machinery.

[15] Ian Neal, Andrew Quinn, and Baris Kasikci. You can find persistent memory bugs if you squint hard enough! 2022.

[16] Ian Neal, Ben Reeves, Ben Stoler, Andrew Quinn, Youngjin Kwon, Simon Peter, and Baris Kasikci. AGAMOTTO: How persistent is your persistent memory application? In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1047–1064. USENIX Association, November 2020.

[17] PMDK. An introduction to pmemcheck. 2015.

[18] Seemanta Saha, Ripon K. Saha, and Mukul R. Prasad. Harnessing evolution for multi-hunk program repair, 2019.

[19] Steve Scargall. *Programming Persistent Memory: A Comprehensive Guide for Developers*. 01 2020.