Detecting and Repairing Bugs in Persistent Concurrent Programs

Tooba Khan, Srivatsan Ravi, Chao Wang
University of Southern California

1 Introduction

The availability of byte-addressable Persistent Memory (PM) has sparked increased efforts towards designing persistent concurrent data structures capable of recovering from system crashes due to power failures. For concurrent programs, in particular, it can be challenging for programmers to ensure proper data persistence, which is crucial for maintaining consistency and guaranteeing the correct order of commits.

Our work aims to find the missing *CLFLUSHOPTs*(efficiently flushes cache lines, optimizing cache performance) and *SFENCEs*(ensures the correct ordering of STOREs in the PM). While there are attempts to address this PM debugging challenge, they all rely on programmers to write and debug software; furthermore, there is little work on debugging automation. To overcome this challenge, we propose a *trace-based analysis* method for detecting and repairing PM bugs. Given an execution trace and a desired property, our method first generates a symbolic formula $\Phi_{program} \wedge \neg\Phi_{property}$ to encode the persistency-related program behaviors and check them against the property. It guarantees that the symbolic formula is satisfiable *iff* there exists a PM bug. Our method also formulates and solves the repair problem similarly using an off-the-shelf constraint solver.

2 Problem statement

The example Consider the example in Listing 1, which has two threads and three shared variables x, y and z stored in the PM. If we assume that the sequential and concurrency-related constraints encode the *desired* program behavior and thus must always be enforced, then it is easy to notice possible violations of persistency constraints.

Let S_i be the i-th statement of a thread in Listing 1. Within the same thread, the *CLFLUSHOPT* instructions in S_3 and S_4 ensure that x and y are persisted in a sequential order. Furthermore, the *SFENCE* instruction in S_5 ensures that x and y are persisted before any statement after S_5 in Thread 1.

Due to the write-read conflict between the write of x in S_2 and the read of x in S_6, we must ensure that the write of y in S_6 takes effect after the write of x in S_2 takes effect. Otherwise, the PM may be put in an inconsistent state in the event of a power or system failure (where x has an old value but y has a new value). Unfortunately, the current implementation does not prevent the write of y in S_6 from taking effect before the write of x in S_2 in the PM, since $S_7 - S_8$ in Thread 2 may be executed before $S_4 - S_5$ in Thread 1.

```
Thread 1                          Thread 2
1  y = 0                          1
2  x = A                          2
3  clflushopt(y)                  3
4  clflushopt(x)                  4
5  sfence()                       5
6                                 6  y = x
7                                 7  clflushopt(y)
8                                 8  sfence()
9  if x==y:                       9
10    z = 0                       10
11    clflushopt(z)               11
12                                12 r = z
13                                13 clflushopt(r)
```

Listing 1: Motivating example.

Our method is designed to detect and then repair this PM bug. One possible repair that our method returns would be making $S_1 - S_5$ and $S_6 - S_8$ critical sections, e.g., by using the mutex lock and unlock primitives.

The execution trace The input to our method is the trace of a buggy program execution. Each event of the trace, denoted $trace[i]$, consists of four elements:

- **trace[i][0]** OPCODE *(which may be LOAD, STORE, FLUSH, and FENCE)*
- **trace[i][1]** Target variable *(variable to be stored in case of STORE and FLUSH and a temporary variable in case of load)*
- **trace[i][2]** Value to be loaded *(value of importance in case of Loads)*
- **trace[i][3]** Thread ID

In the trace $\langle S_1, S_2, S_6, S_7, S_8, S_3, S_4, S_5, S_9, S_{10}, S_{11}, S_{12}, S_{13}\rangle$, if a crash occurs right after S_7, the PM will be put into an inconsistent state. Using this inconsistent state for failure recovery will cause y to have the new value but x to have the old value (but x should have been equal to y). Thus, at S_9, the condition will evaluate to $False$ instead of $True$. This inconsistent

state will be further propagated and reflect in the value of r and S_{12}.

3 Methodology

Central to this work is calculating persistent time intervals to find the STOREs subsequently accessed by LOADs or STOREs, potentially resulting in PM violations. **Defining the persistent time** We define the persistent time to be a time interval $[lb, ub]$ that represents the range when a written value is persisted in the PM. The written value may be persisted anytime after executing the corresponding *CLFLUSHOPT* instruction and before executing the corresponding *SFENCE*. Thus, the lower bound lb is defined as the time step of the first corresponding *CLFLUSHOPT*. The upper bound ub is defined as the time step of the first corresponding *SFENCE*. The written value is guaranteed to be persisted in the PM before *SFENCE*.

Consider two types of common PM bugs: *durability* violations and *crash consistency* violations: *Durability* bug means a PM STORE is not followed its *CLFLUSHOPT*, in which case the written value is not guaranteed to show up in the PM. In other words, if a crash occurs, the written value may be lost. *Crash consistency* bug means two PM STOREs should be persisted in a certain order, but the order is not enforced properly. For example, if the write of y to PM depends on the value of x written to PM earlier, then we expect $UB_x < LB_y$. Otherwise, the PM may be put into an inconsistent state in the event of a power or system failure.

Computing the persistent time Our method centers around computing the persistent time interval for all statements in the execution trace. After computing the persistent time intervals, using them to detect PM bugs will be straightforward.

To compute the persistent time intervals, we need to find the program counter values for each statement. For example, since we have 2 threads in our running example, to maintain the sequential and concurrency constraints of our current trace, S_1, and S_2 in Thread 1 will have the program counter values 1 and 2, respectively. However, the program counter value of S_6 can range from 3 to 6. Nevertheless, the only value of program counter for S_6 that does not violate PM constraint is 6.

After finding the possible values of program counter for each statement, the next step is to find the lower and upper bounds of persistent time for each LOAD and STORE. For every STORE(S_i) statement, the lower bound is the value(or range of values) for program counter when a *flush* is first encountered for S_i. Similarly, the upper bound is the value(or range of values) of program counter for the first *fence* statement. For every LOAD, the lower bound is the value(or range of values) of its program counter. The upper bound on the persistent time of a LOAD statement does not have any significance, so we assign it the value "∞".

We then model our bug detection using logical constraints, denoted $\Phi_{program} \wedge \neg\Phi_{property}$. We model three kinds of constraints in $\Phi_{program}$: sequential, concurrency, and persistency. The sequential constraints are used to fix the ordering of statements executed by a single thread, while the concurrency constraints are used to fix the ordering that should be maintained between two or more threads to ensure correct program behavior. The persistency constraints encode the correct behavior of the PM. We model two types of bugs in $\neg\Phi_{property}$: durability violations and crash consistency violations.

Durability constraint is encoded as follows:

$UPPER\_BOUND(S_i) \leq l \quad \forall S_i \in STORE$, where, l = last statement in the program.

Crash consistency constraint is encoded as follows: For $S_i \in STORE$ and $L_i \in LOAD$ such that L_i reads from S_i: $UPPER\_BOUND(S_i) \leq LOWER\_BOUND(L_i)$

Bug repair *Repair* of *durability* is achieved by the insertion of *CLFLUSHOPT* instructions in the code. We insert *CLFLUSHOPT* instructions for all memory locations that need to be written to the PM thus eliminating all *durability* bugs.

Repair of *crash consistency* bugs is achieved by two ways. The first method eliminates *Crash consistency* bugs within a thread by inserting appropriate *SFENCES* to ensure a particular ordering between STOREs.

The second method eliminates *crash consistency* bugs arising due to interleaving. This is achieved by locking the access to shared variables. The lock ensures that before a context switch is made and another thread can access the shared data, it should be persisted in the PM. In our example, the execution sequence $\langle S_1, S_2, S_6, S_7, S_8, S_3, S_4, S_5, S_9, S_{10}, S_{11}, S_{12}, S_{13}\rangle$ is repaired by locking the statements from $S_2 - S_5$.

Implementation and results We have implemented our approach as a tool. It uses LLVM to generate traces of concurrent programs. We then encode the persistency constraints and use the Z3 theorem prover to solve them. We repair the original programs using the violated PM constraints by adding appropriate locks, *CLFLUSHOPTs* and *SFENCEs*.

We are conducting thorough experiments, including examining promising results from tests conducted on simplified toy examples. Using our trace-based bug detection and repair, we will conduct experiments involving well-known concurrent data structures like Redis and Memcached.