Checking Robustness to Weak Persistency Models

Hamed Gorjiara  
University of California, Irvine

Weiyu Luo  
University of California, Irvine

Alex Lee  
University of California, Irvine

Harry Xu  
University of California, Los Angeles

Brian Demsky  
University of California, Irvine
Persistent Memory

- Non-volatile, byte-addressable, and high-speed
Persistent Memory

- Non-volatile, byte-addressable, and high-speed

- Challenge:
  - The program needs to guarantee crash-consistency
Persistent Memory

Supporting crash consistency

- Ordering: existence of processor cache
Persistent Memory

Supporting crash consistency

- Ordering: existence of processor cache

Program Code

1. \( x = 1 \)
2. \( y = 2 \)
Persistent Memory

Supporting crash consistency

- Ordering: existence of processor cache

<table>
<thead>
<tr>
<th>Program Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. ( x = 1 )</td>
</tr>
<tr>
<td>2. ( y = 2 )</td>
</tr>
</tbody>
</table>
Persistent Memory

Supporting crash consistency

- Ordering: existence of processor cache

Program Code

1. x = 1
2. y = 2
Persistent Memory

Supporting crash consistency

- Ordering: existence of processor cache
- Durability: using cache instructions
State-of-the-Art

- Constructive approaches
- Testing and checking frameworks
State-of-the-Art

- Constructive approaches
- Testing and checking frameworks

+ Systematically transforming programs
  - Only on lock-free data structure
  - Inject unnecessary flush and fence$^{[2,3,4]}$
  - Memory overhead$^{[1]}$

[1] Mirror: Friedman et. al. PLDI'2021
[3] Dananjaya et. al. ASPLOS’20
[4] Venkataraman et. al. FAST’11
State-of-the-Art

- Constructive approaches
- Testing and checking frameworks

- Bugs with visible symptoms\textsuperscript{[5,6]}
- User annotation or heuristics\textsuperscript{[1-4]}
- Manual inspection of long execution trace\textsuperscript{[1-6]}

\textsuperscript{[1]} Witcher  \quad \textsuperscript{[5]} Jaaru
\textsuperscript{[2]} PMTest  \quad \textsuperscript{[6]} Yat
\textsuperscript{[3]} PMDebugger
\textsuperscript{[4]} XFDetector
Strict Persistency

- A persistency model where
  - Persistency memory order = Volatile Memory order

```
1. x = 1
2. y = 2
```
Strict Persistency

- A persistency model where
  - Persistency memory order = Volatile Memory order
- Naïve implementation
  - Using flush instructions after each memory operation

Program Code

1. \( x = 1 \)
2. \( \text{Flush}(&x) \)
3. \( y = 2 \)
4. \( \text{Flush}(&y) \)
Key Observation

Typical correct use of flush instructions in PM programs ensures:

Program executions under \textcolor{red}{weak persistency} semantics are equivalent to those under \textcolor{green}{strict persistency} semantics
Robustness

**Pre-crash Execution**
- T1: x=1, Rx=2, y=3, y=4
- T2: x=2, y=5

**Post-crash Execution**
- Rx = 2
- Ry = 3

Weak Persistency Models
Robustness

### Weak Persistency Models
- **Pre-crash Execution**
  - T1: \(x=1\), \(Rx=2\), \(y=3\), \(y=4\)
  - T2: \(x=2\), \(y=5\)

- **Post-crash Execution**
  - T1: \(Rx = 2\), \(Ry = 3\)
  - T2: \(Rx = 2\), \(Ry = 3\)

### Strict Persistency Models
- **Pre-crash Execution**
  - T1: \(x=1\), \(Rx=2\), \(y=3\)
  - T2: \(x=2\), \(y=3\)

- **Post-crash Execution**
  - T1: \(Rx = 2\), \(Ry = 3\)
  - T2: \(Rx = 2\), \(Ry = 3\)

Timeline
Robustness

Identical Post-crash Execution

Weak Persistency Models

Strict Persistency Models
Robustness

Timeline

Weak Persistency Models

Pre-crash Execution

T1
- x=1
- Rx=2
- y=3
- y=4
- y=5

T2
- x=2

Post-crash Execution

Rx = 2
Ry = 3

Timeline

Strict Persistency Models

Pre-crash Execution

T1
- x=1
- Rx=2
- y=3

T2
- x=2
- Rx=2

Post-crash Execution

Rx = 2
Ry = 3

Reads-from Relation
Robustness

Timeline

Pre-crash Execution

T1

x=1
Rx=2
y=3
y=4

T2

x=2

Post-crash Execution

Rx = 2
Ry = 3

Sequenced-before Relation

Timeline

Pre-crash Execution

T1

x=1
Rx=2
y=3
y=4

T2

x=2

Post-crash Execution

Rx = 2
Ry = 3

Weak Persistency Models

Strict Persistency Models
Robustness

T1
x=1
Rx=2
y=3
y=4
y=5

T2
x=2

Pre-crash Execution

Post-crash Execution

TSO Ordering

T1
x=1
Rx=2

T2

Pre-crash Execution

Post-crash Execution

Pre-crash Execution

Post-crash Execution

Weak Persistency Models

Strict Persistency Models
Robustness

A program is **robust** to a weak persistency model:

- For *any* crash event and *any* post-crash execution under the weak persistency model, there **exists** some execution under strict persistency model that is equivalent to it.
Robustness

A program is robust to a weak persistency model:

- For any crash event and any post-crash execution under the weak persistency model, there exists some execution under strict persistency model that is equivalent to it.

Robustness is sufficient condition to assure correct usage of flush and fence operations
Robustness and Commit Store
Our Solution: PSan

Persistent Memory Sanitizer (PSan):

- Dynamically checks robustness for programs
- Detects bugs caused by missing flushes/fences
- Bug localization
- Suggests bug fixes
PSan Overview

- Built on top of Jaaru
PSan Overview

- Built on top of Jaaru
- Random vs. model checking mode
PSan Key Idea

- PSan computes a set of strictly persistent executions whose pre-crash executions are consistent with the post-crash execution.
PSan Key Idea

• Reason about the potential crash interval
  ○ Using constraints

![Diagram showing pre-crash and post-crash execution with variables x = 1 and x = 2, and Rx = ?]
PSan Key Idea

- Reason about the potential crash interval
  - Using constraints
PSan Key Idea

Pre-crash execution

Pre-crash Code

1. x = 1
2. y = 2
3. x = 3
4. y = 4
5. x = 5

Post-crash Code

1. r1 = y
   // r1 = ???
2. r2 = x
   // r2 = ???

Crash

x = 1  y = 2  x = 3  y = 4  x = 5
PSan Key Idea

Pre-crash execution

Pre-crash Code
1. x = 1
2. y = 2
3. x = 3
4. y = 4
5. x = 5

Post-crash Code
1. r1 = y
   // r1 = ???
2. r2 = x
   // r2 = ???

strictly persistent executions represented by the constraints
PSan Key Idea

Pre-crash Code
1. x = 1
2. y = 2
3. x = 3
4. y = 4
5. x = 5

Post-crash Code
1. r1 = y
   // r1 = ???
2. r2 = x
   // r2 = ???

strictly persistent executions represented by the constraints
PSan Key Idea

strictly persistent executions represented by the constraints
PSan Key Idea

Pre-crash execution

Pre-crash Code

1. x = 1
2. y = 2
3. x = 3
4. y = 4
5. x = 5

Post-crash Code

1. r1 = y
   // r1 = ???
2. r2 = x
   // r2 = ???

Robustness Violation

strictly persistent executions represented by the constraints
Supporting Multi-threaded Programs

- Each thread can make different progress when a program crashes
- Each thread requires its own potential crash interval constraints
- Deducing constraints
  - TSO ordering between stores to the same variable
Supporting Multi-threaded Programs

Pre-crash execution

Post-crash execution
Supporting Multi-threaded Programs

Pre-crash execution

T1
x = 1

T2
x = 2

tso

Post-crash execution

Rx = 1
Supporting Multi-threaded Programs

Pre-crash execution

\[ T_1 \]
- \( x = 1 \)
- \( \text{Flush(&x)} \)

\[ T_2 \]
- \( r_1 = x \)
- \( y = r_1 \)
- \( \text{Flush(&y)} \)

Post-crash execution

- \( R_x = ? \)
- \( R_y = ? \)
Supporting Multi-threaded Programs

Pre-crash execution

T1
- x = 1
- Flush(&x)

T2
- r1 = x = 1
- y = r1
- Flush(&y)

Post-crash execution

- Rx = ?
- Ry = ?
Supporting Multi-threaded Programs

Pre-crash execution

T1

x = 1

Flush(&x)

T2

r1 = x = 1

y = r1

Flush(&y)

Post-crash execution

Rx = 0

Ry = ?
Supporting Multi-threaded Programs

Pre-crash execution

T1

x = 1

Flush(&x)

T2

r1 = x = 1

y = r1

Flush(&y)

Post-crash execution

Rx = 0

Ry = 1
Supporting Multi-threaded Programs

Pre-crash execution

T1
x = 1
Flush(&x)

T2
r1 = x = 1
y = r1
Flush(&y)

Post-crash execution

Rx = 0
Ry = 1

Robustness Violation
Supporting Multi-threaded Programs

- Each thread can make different progress when a program crashes
- Each thread requires its own potential crash interval constraints
- Deducing constraints
  - TSO ordering between stores to the same variable
  - Happens-before relation
Supporting Multi-threaded Programs

Pre-crash execution

T1
- x = 1
- Flush(&x)

T2
- r1 = x = 1
- y = r1
- Flush(&y)

Post-crash execution

Rx = ?

Ry = ?
Supporting Multi-threaded Programs

Pre-crash execution

T1

T2

x = 1
Flush(&x)

r1 = x = 1
Flush(&y)

y = r1

Post-crash execution

Rx = 0

Ry = ?
Supporting Multi-threaded Programs

Pre-crash execution

T1
x = 1
Flush(&x)

T2
r1 = x = 1
y = r1
Flush(&y)

Post-crash execution

Rx = 0
Ry = 1
Supporting Multi-threaded Programs

Pre-crash execution

T1
- x = 1
- Flush(&x)

T2
- r1 = x = 1
- y = r1
- Flush(&y)

Post-crash execution

Rx = 0
Ry = 1

Robustness Violation

x = 1
Flush(&x)

r1 = x = 1
y = r1
Flush(&y)
Suggesting Fixes for Robustness Violations

- Defining a fix as a set of flush intervals

Two cases for robustness violations:

1. Reading from too old of store
2. Reading from too new of store
Suggesting Fixes for Robustness Violations

Pre-crash execution

Post-crash execution

Case 1: Reading from too old of store
Suggesting Fixes for Robustness Violations

Pre-crash execution

Post-crash execution

Case 1: Reading from too old of store
Suggesting Fixes for Robustness Violations

Pre-crash execution

Post-crash execution

Reads from a store that is too old

Case 1: Reading from too old of store

Ry = 4
Rx = 1
Suggesting Fixes for Robustness Violations

Pre-crash execution

Post-crash execution

Flush interval suggested by PSan for variable X

Case 1: Reading from too old of store
Suggesting Fixes for Robustness Violations

Pre-crash execution

Post-crash execution

Case 2: Reading from too new of store
Suggesting Fixes for Robustness Violations

**Case 2: Reading from too new of store**

Pre-crash execution

- $x = 1$
- $y = 2$
- $x = 3$
- $y = 4$
- $x = 5$

Post-crash execution

- $R_y = 2$
- $R_x = ?$

54
Suggesting Fixes for Robustness Violations

Pre-crash execution

Post-crash execution

Reads from a store that is too new

Case 2: Reading from too new of store
Suggesting Fixes for Robustness Violations

**Pre-crash execution**

- x = 1
- y = 2
- x = 3
- y = 4
- x = 5

Flushing interval suggested by PSan for variable Y

**Post-crash execution**

- Ry = 2
- Rx = 5

Case 2: Reading from too new of store
Suggesting Fixes for Robustness Violations

Pre-crash execution

T1

x = 1

Flush(&x)

T2

r1 = x

y = r1

Flush(&y)

Post-crash execution

Rx = 0

Ry = 1

Special case: Multi-threaded programs
Suggesting Fixes for Robustness Violations

Pre-crash execution

T1
- x = 1
- Flush(&x)

T2
- r1 = x
- y = r1
- Flush(&y)

Post-crash execution

Rx = 0
Ry = 1

Special case: Multi-threaded programs

Flush(&x)
Flush(&y)
Suggesting Fixes for Robustness Violations

Flush interval is empty!!

Special case: Multi-threaded programs
Suggesting Fixes for Robustness Violations

Pre-crash execution

T1

x = 1

Flush(&x)

T2

r1 = x

y = r1

Flush(&y)

Flush interval

Post-crash execution

Rx = 0

Ry = 1
Evaluation

Evaluated PSan on:

- A collection of data structure: RECIPE, CCEH, Fast Fair
- Popular real-world frameworks and applications: PMDK, Memcached, and Redis

PSan found 48 bugs of which 17 are new!
Evaluation

PSan found 3 types of bugs:

- Missing flush and fence operations
- Cache line alignment bugs
- Memory management bugs
Evaluation

PSan found 3 types of bugs:

- Missing flush and fence operations
- Cache line alignment bugs
- Memory management bugs

```cpp
btree::btree()
{
    root = (char*)new page();
    // clflush((char*)root, sizeof(page));
    height = 1;
    // clflush((char*)this, sizeof(btree), false, true);
}
```
Evaluation

PSan found 3 types of bugs:

- Missing flush and fence operations
- Cache line alignment bugs
- Memory management bugs
Evaluation

PSan found 3 types of bugs:

- Missing flush and fence operations
- Cache line alignment bugs
- Memory management bugs
  - Garbage collection, memory allocation components
Evaluation

- Negligible overhead compared to Jaaru
- Average **13.1s** to explore all executions revealing all bugs for each benchmark
Conclusion

Testing persistent memory program is **challenging**, and fixing persistency bugs is **difficult**!

**PSan**

- Presents **robustness**, a sufficient correctness condition
- Finds persistency bugs caused by missing flushes/fences
- Found **48** persistency bugs of which **17** are new
- Localizes persistency bugs and suggests fixes
- Available on: [plrg.ics.uci.edu/psan](http://plrg.ics.uci.edu/psan)