# 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

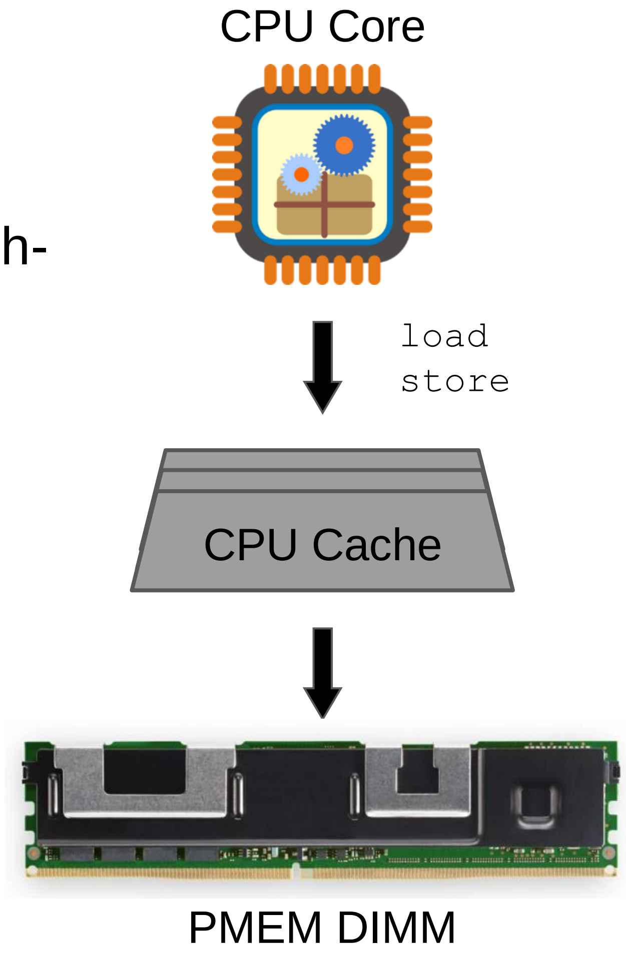
- Non-volatile, byte-addressable, and high-speed

PMEM DIMM

# Persistent Memory

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

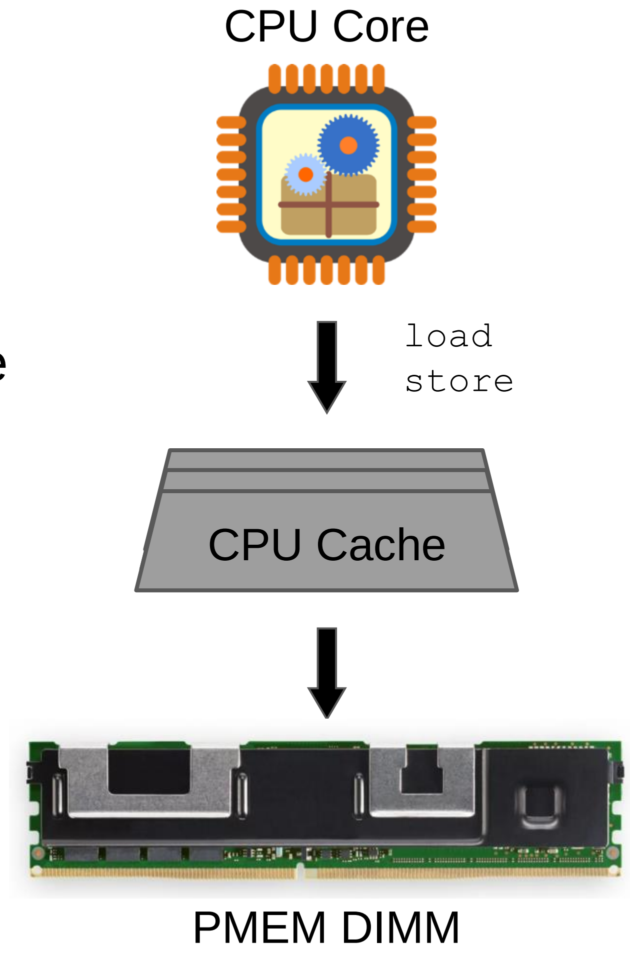
- Challenge:
    - The program needs to guarantee crash-consistency



# Persistent Memory

Supporting crash consistency

- Ordering: existence of processor cache



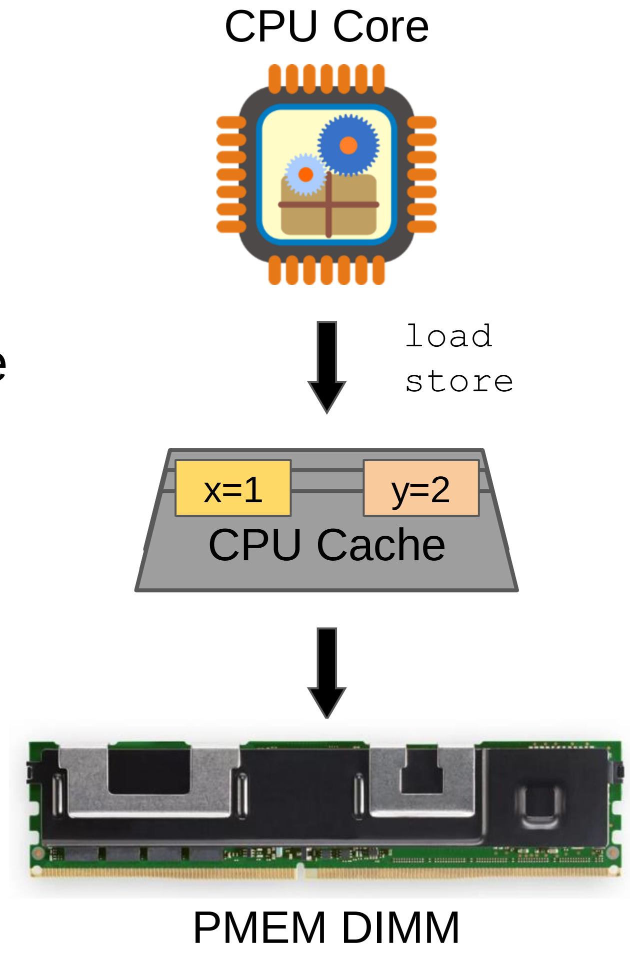
# Persistent Memory

## Supporting crash consistency

- Ordering: existence of processor cache

Program Code

```
1. x = 1
2. y = 2
```



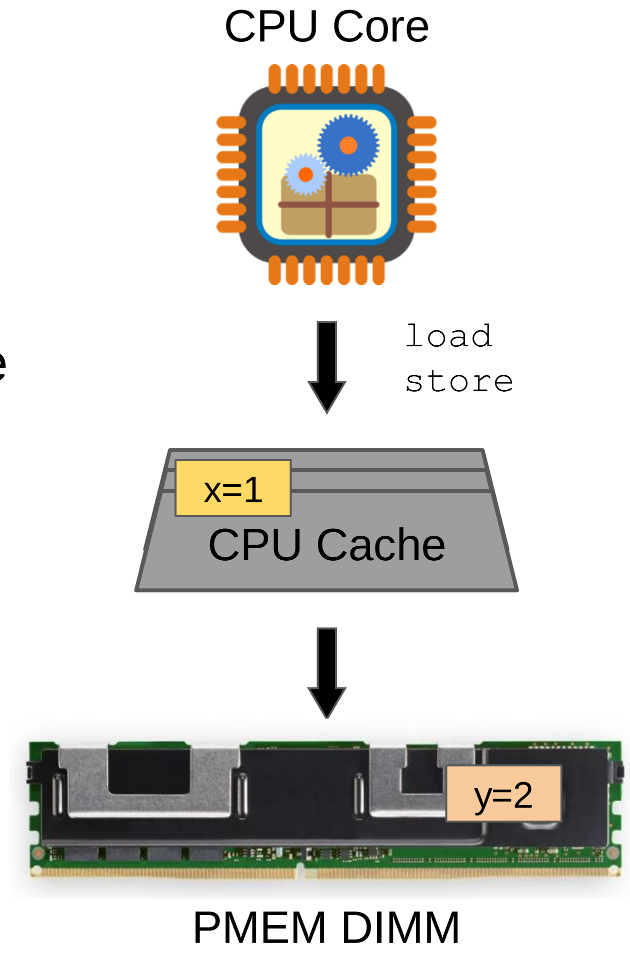
# Persistent Memory

Supporting crash consistency

- Ordering: existence of processor cache

Program Code

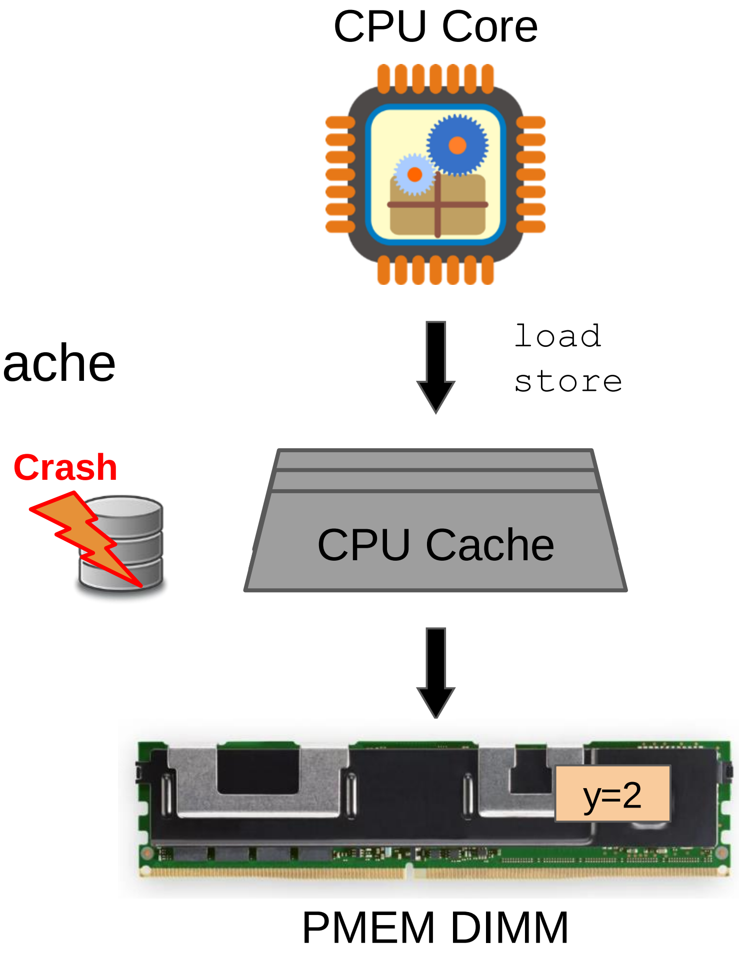
```
1. x = 1
2. y = 2
```



# Persistent Memory

Supporting crash consistency

- Ordering: existence of processor cache



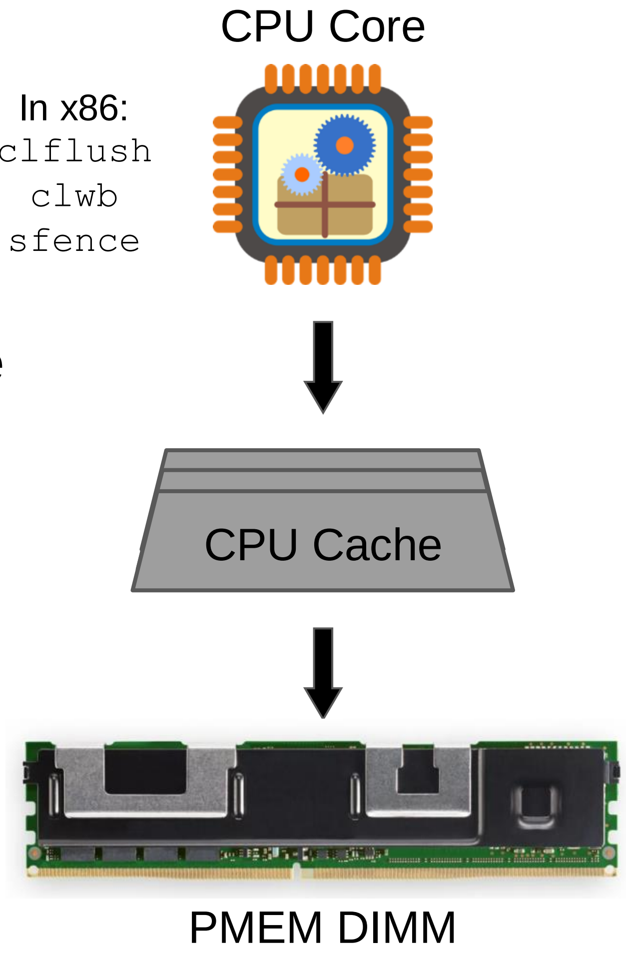

| Program Code |
|---|
| 1. x = 1<br>2. y = 2 |

# Persistent Memory

Supporting crash consistency

- Ordering: existence of processor cache
- Durability: using cache instructions

In x86:
`clflush`
`clwb`
`sfence`



# 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
[2] Izraelevitz et. al. DISC'2016
[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[5,6]
- User annotation or heuristics[1-4]
- Manual inspection of long execution trace[1-6]


[1] Witcher
[2] PMTest
[3] PMDebugger
[4] XFDetector
[5] Jaaru
[6] Yat

# Strict Persistency

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

**Program Code**

```
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

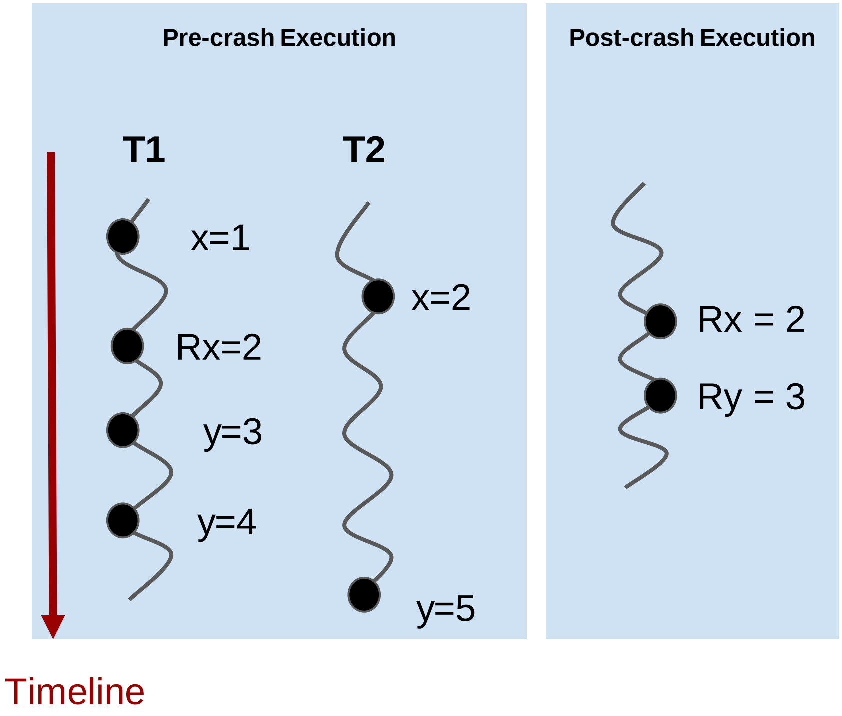
```
1. x = 1
2. Flush(&x)
3. y = 2
4. Flush(&y)
```

# Key Observation

Typical correct use of flush instructions in PM programs ensures:

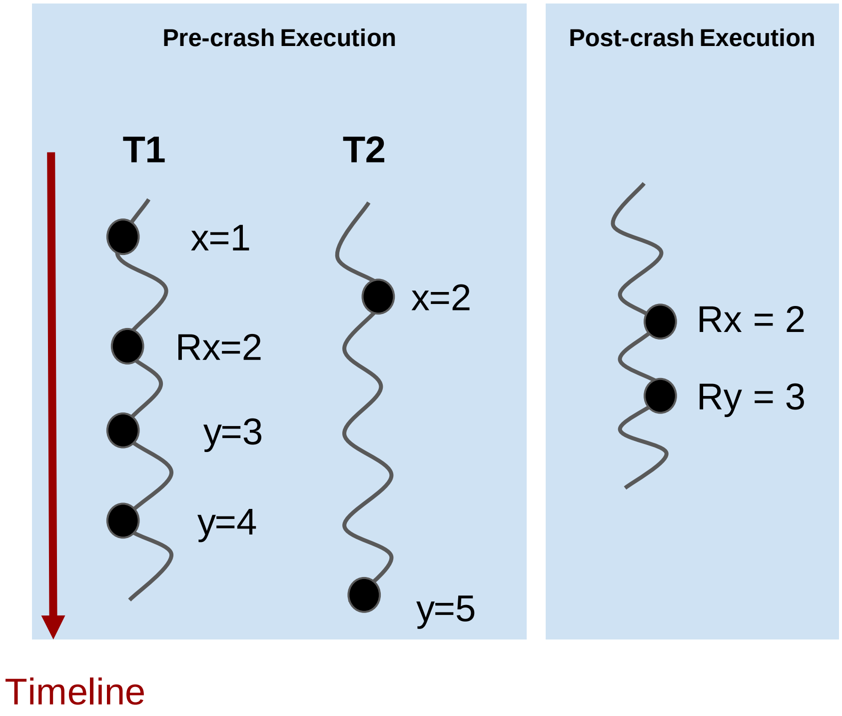
Program executions under weak persistency semantics are equivalent to those under strict persistency semantics

# Robustness

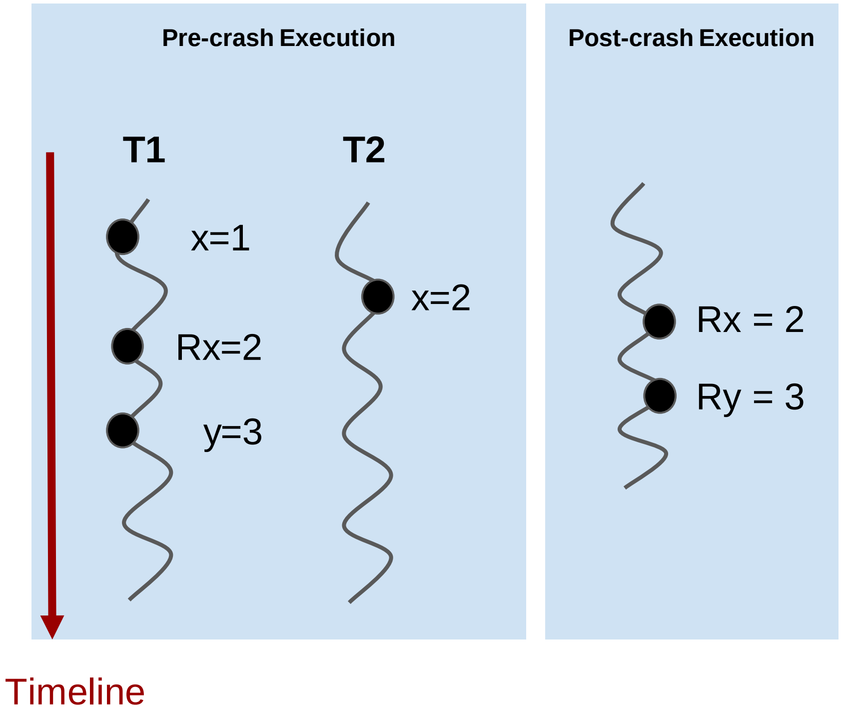



**Weak Persistency Models**

# Robustness



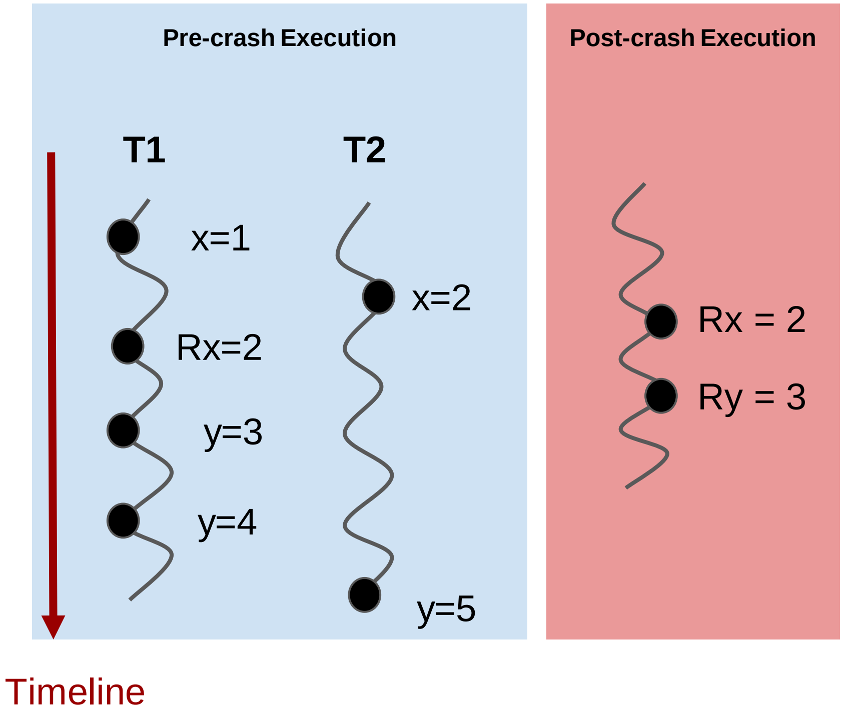

**Weak Persistency Models**



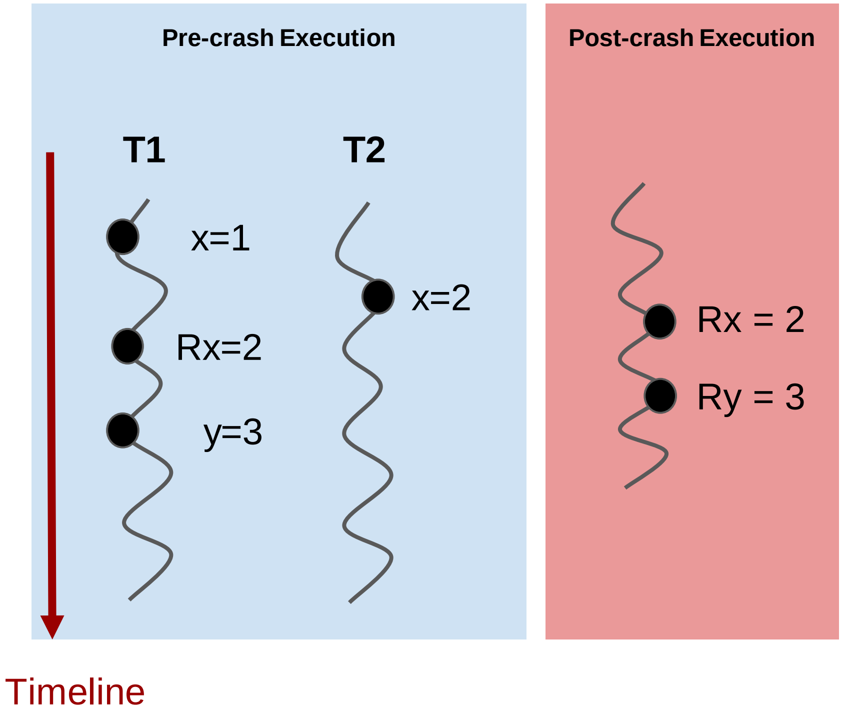

**Strict Persistency Models**

# Robustness

Identical Post-crash Execution



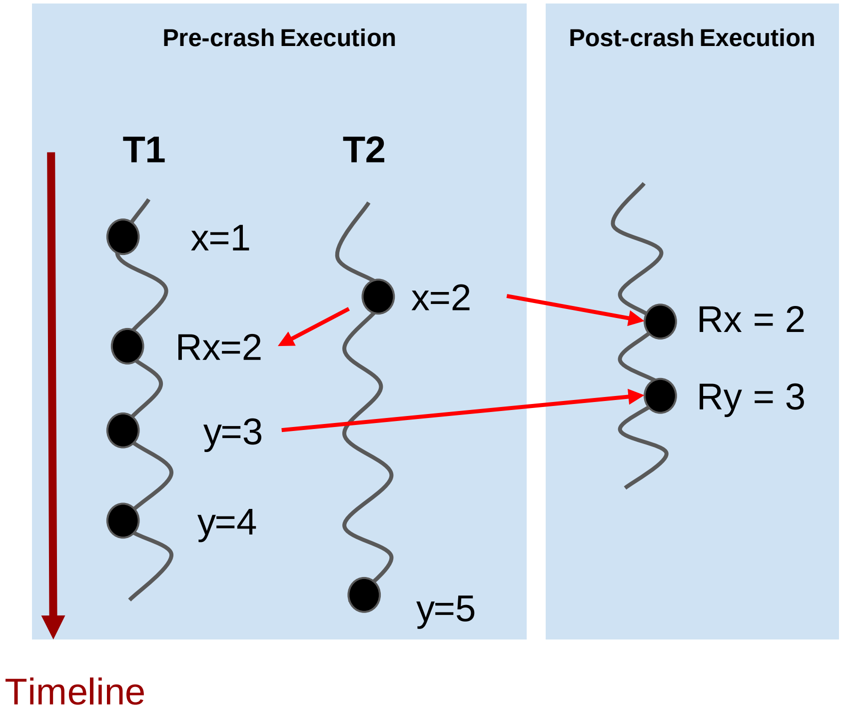

**Weak Persistency Models**



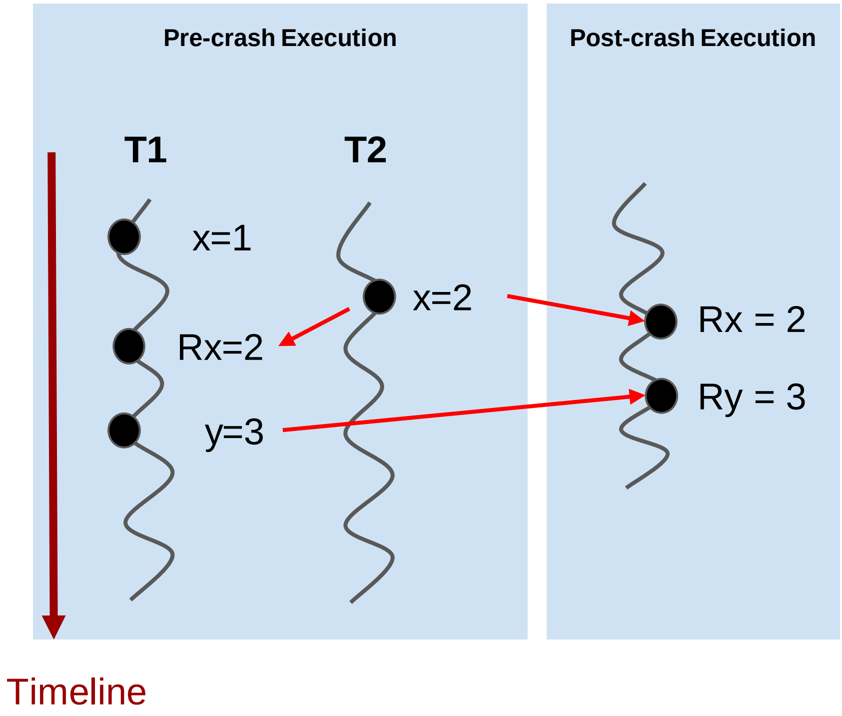

**Strict Persistency Models**

# Robustness

Reads-from Relation



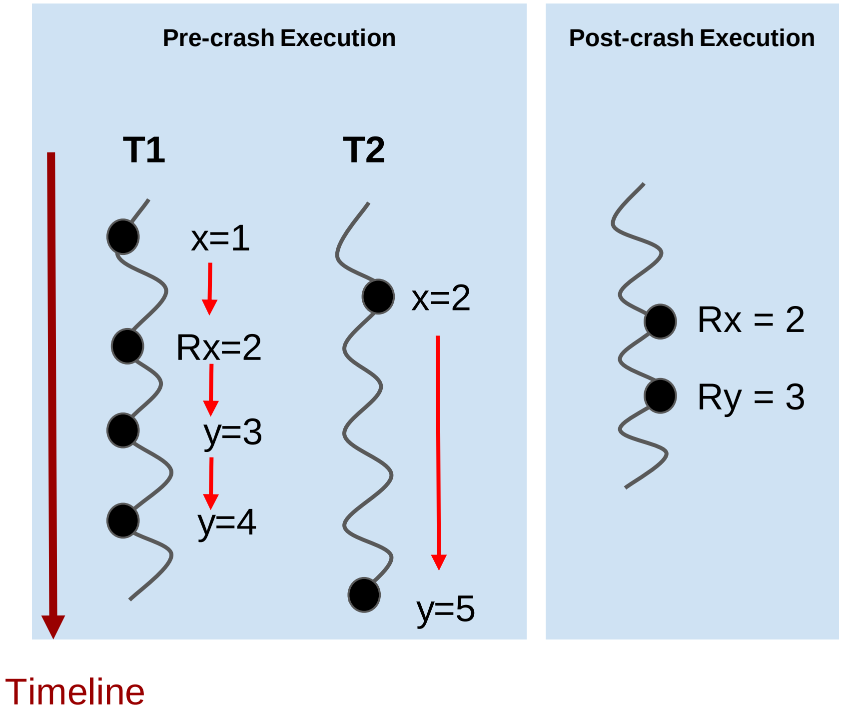

**Weak Persistency Models**



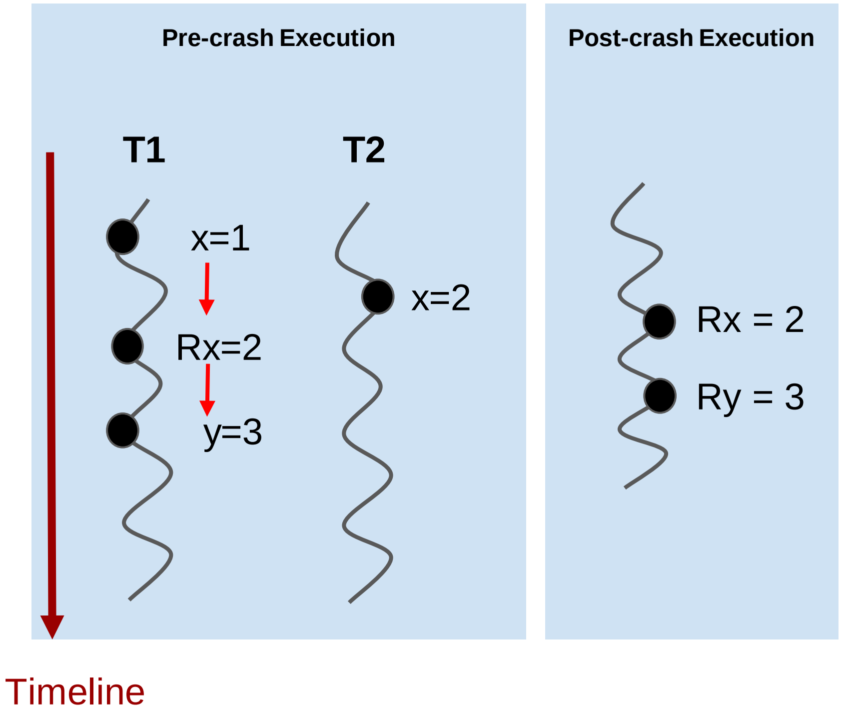

**Strict Persistency Models**

# Robustness

Sequenced-before Relation



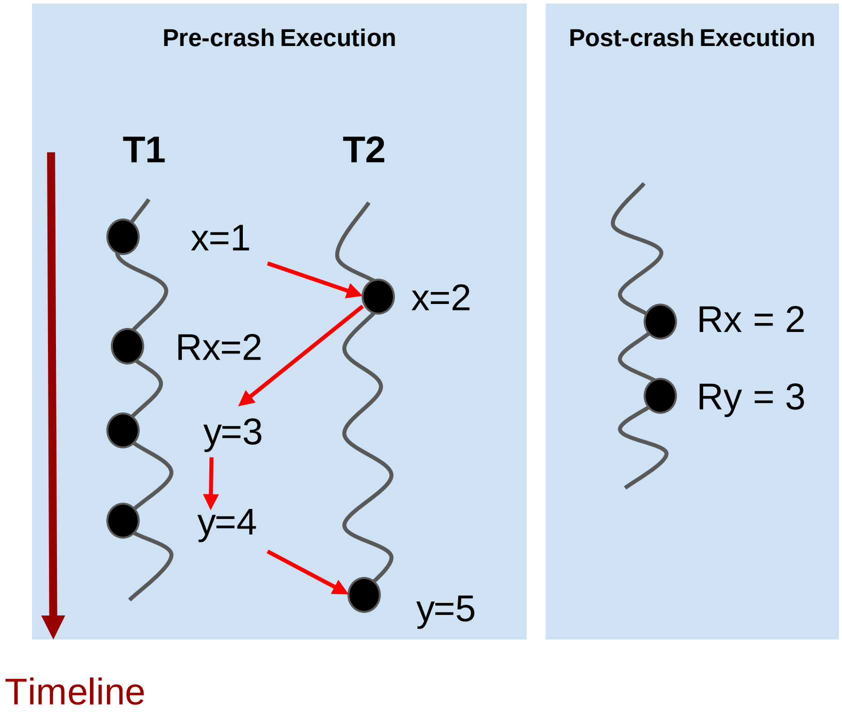

**Weak Persistency Models**



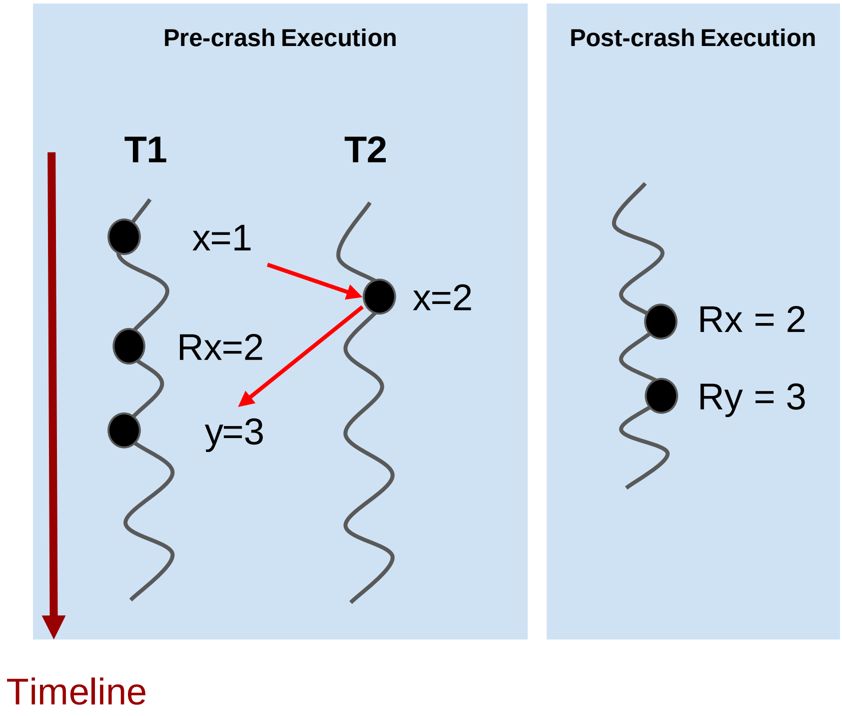

**Strict Persistency Models**

# Robustness

TSO Ordering




**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.

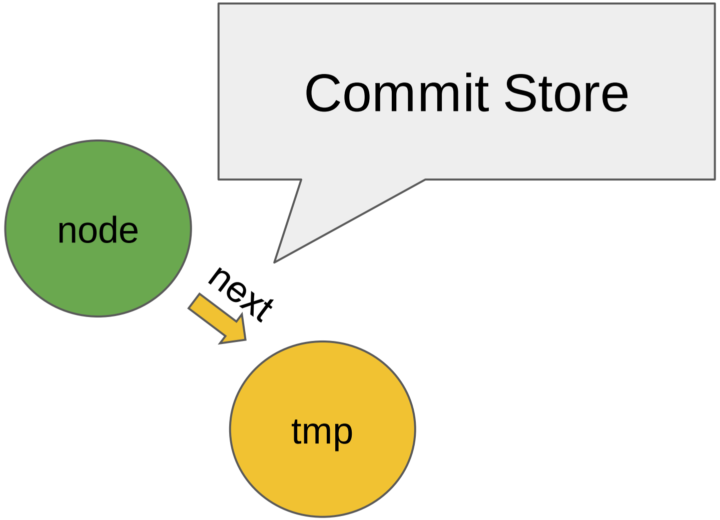
# Robustness

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



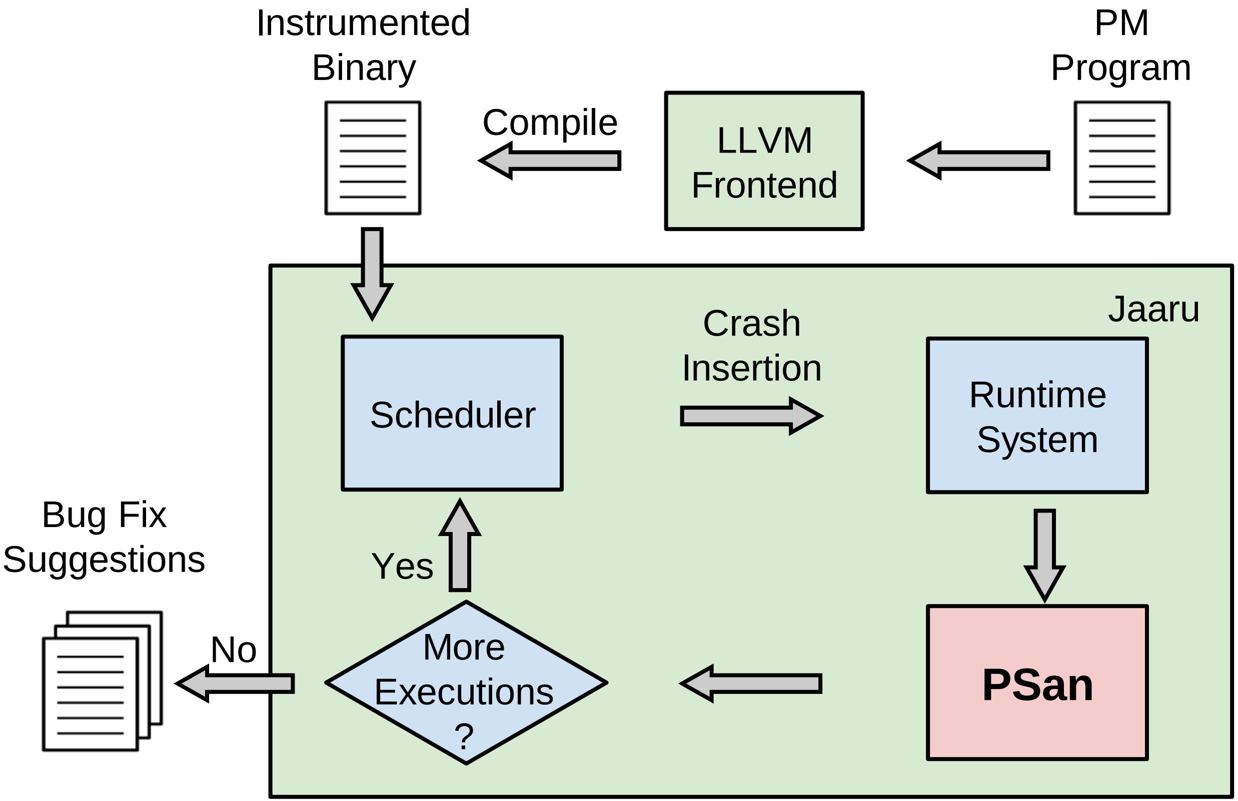
# Our Solution: PSan

**P**ersistent Memory **San**itizer (**PSan**):

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

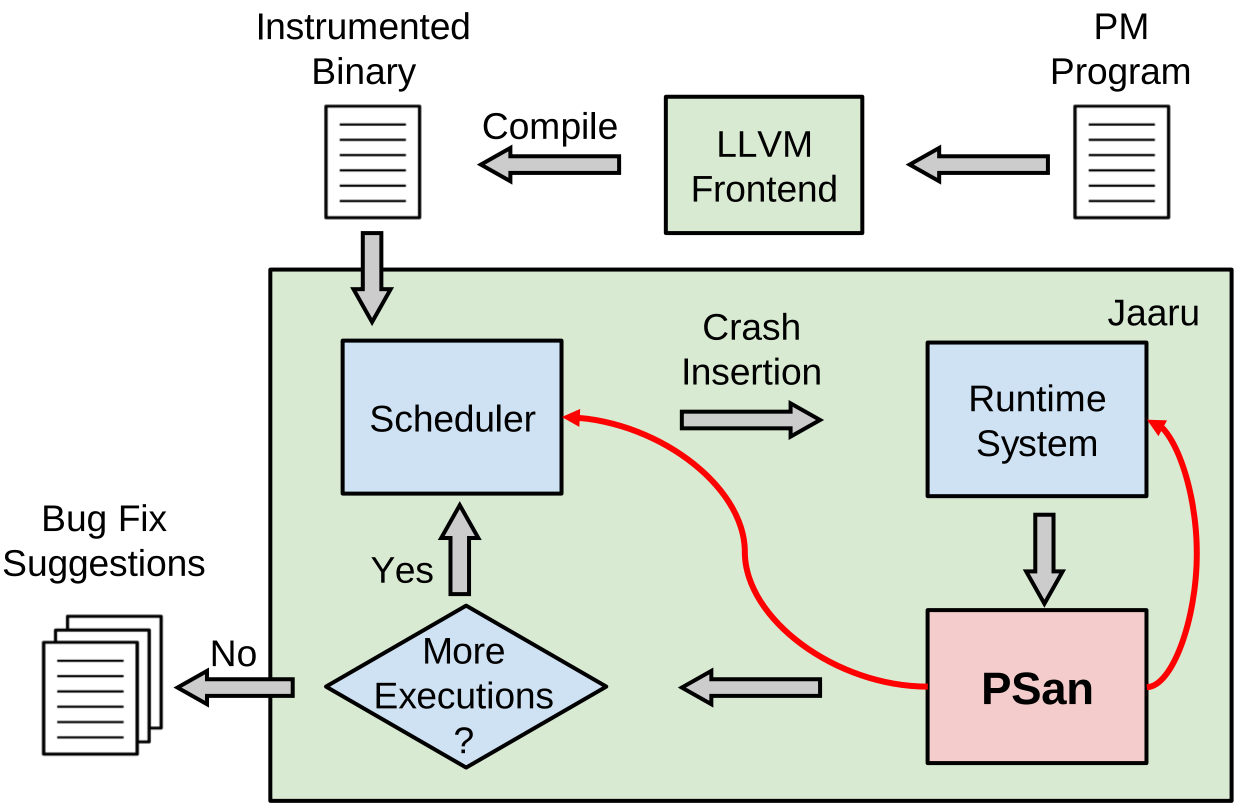
# PSan Overview

- Built on top of Jaaru



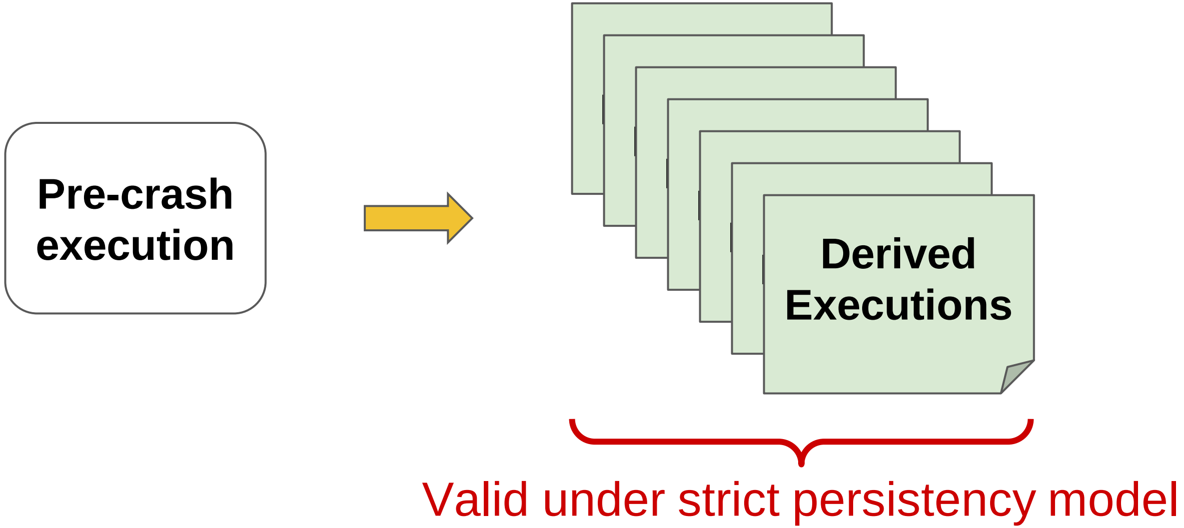
# PSan Overview

- 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

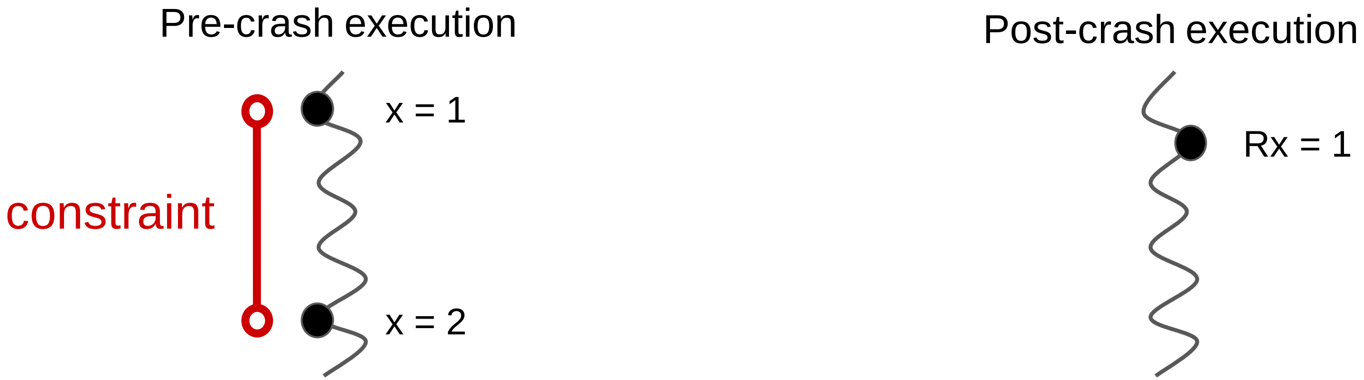


# PSan Key Idea

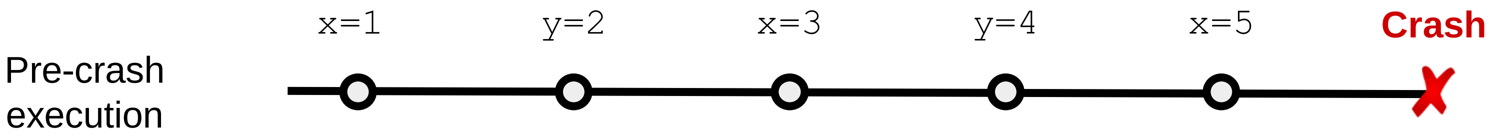
- Reason about the potential crash interval
  - Using constraints

# PSan Key Idea

- Reason about the potential crash interval
    - Using 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 = ???
```

# PSan Key Idea

Pre-crash execution


**Pre-crash Code**

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

```
>>> Crash
```

```
x = 1
>>> Crash
```

```
x = 1
y = 2
>>> Crash
```

```
x = 1
y = 2
x = 3
>>> Crash
```

```
x = 1
y = 2
x = 3
y = 4
>>> Crash
```

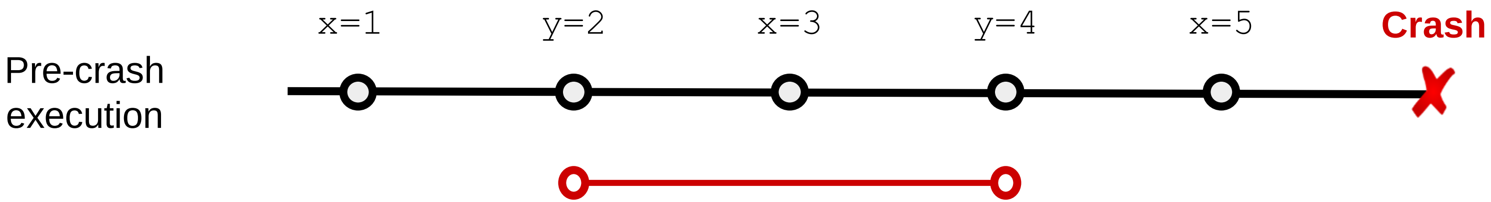
```
x = 1
y = 2
x = 3
y = 4
x = 5
>>> Crash
```

**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
```

```
x = 1
y = 2
>>> Crash
```

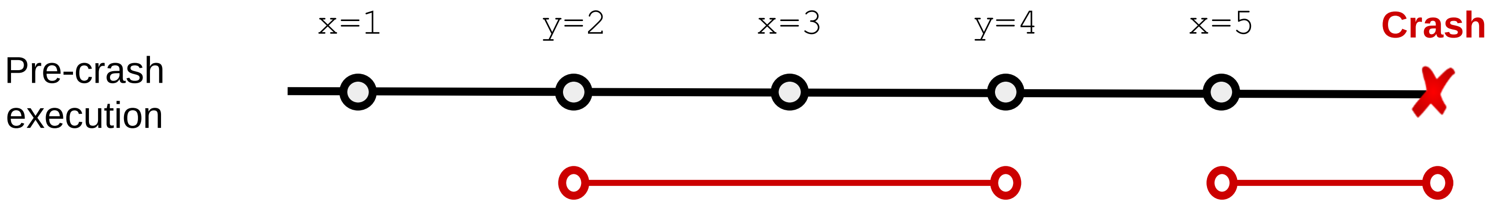
```
x = 1
y = 2
x = 3
>>> Crash
```

**Post-crash Code**

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

strictly persistent executions represented by the constraints

# PSan Key Idea




**Pre-crash Code**

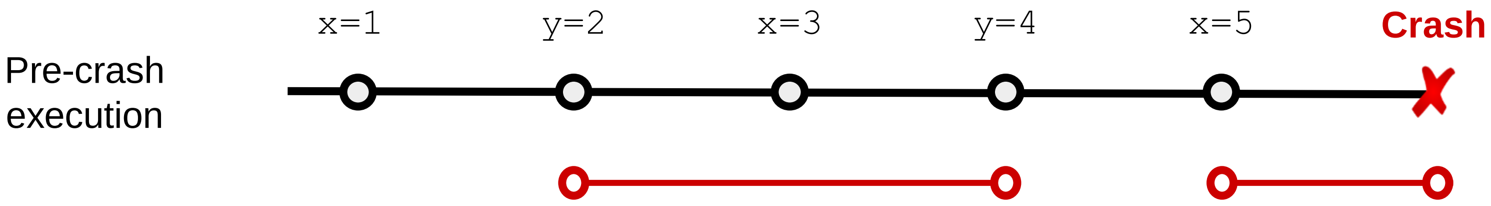
```
1. x = 1
2. y = 2
3. x = 3
4. y = 4
5. x = 5
```

**Post-crash Code**

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

strictly persistent executions represented by the constraints

# PSan Key Idea




**Pre-crash Code**

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

**Robustness Violation**

**Post-crash Code**

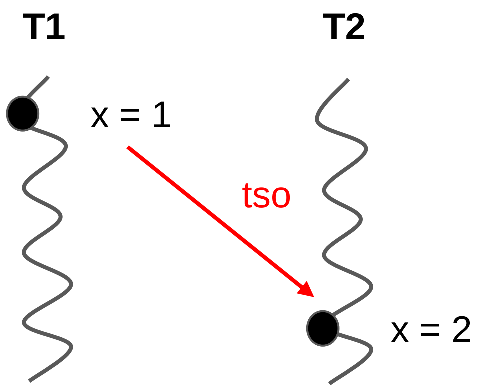
```
 1. r1 = y
// r1 = ???
 2. r2 = x
// r2 = ???
```

strictly persistent executions represented by the constraints

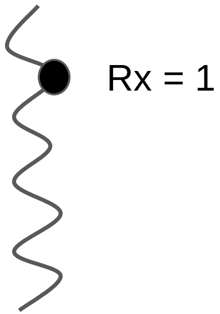
# Supporting Multi-threaded Programs

- 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

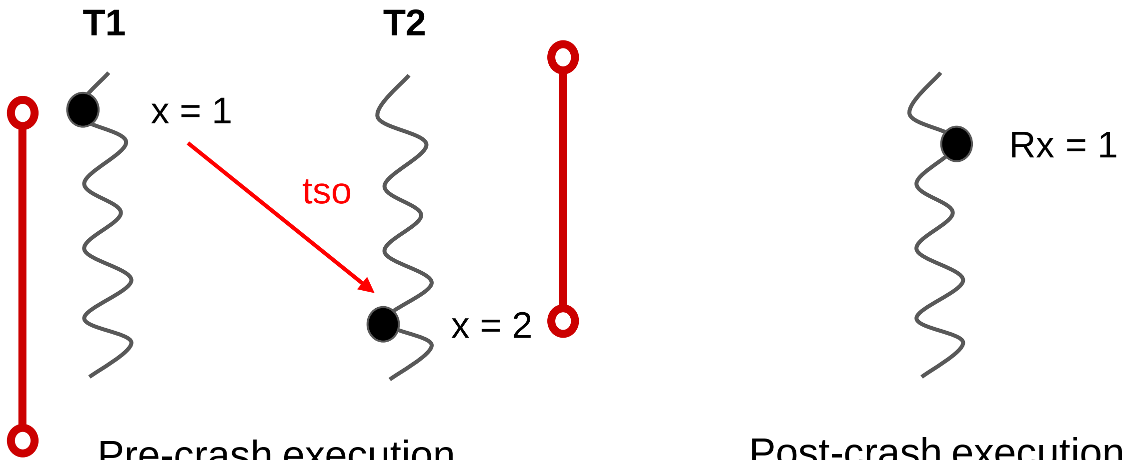



Pre-crash execution




Post-crash execution

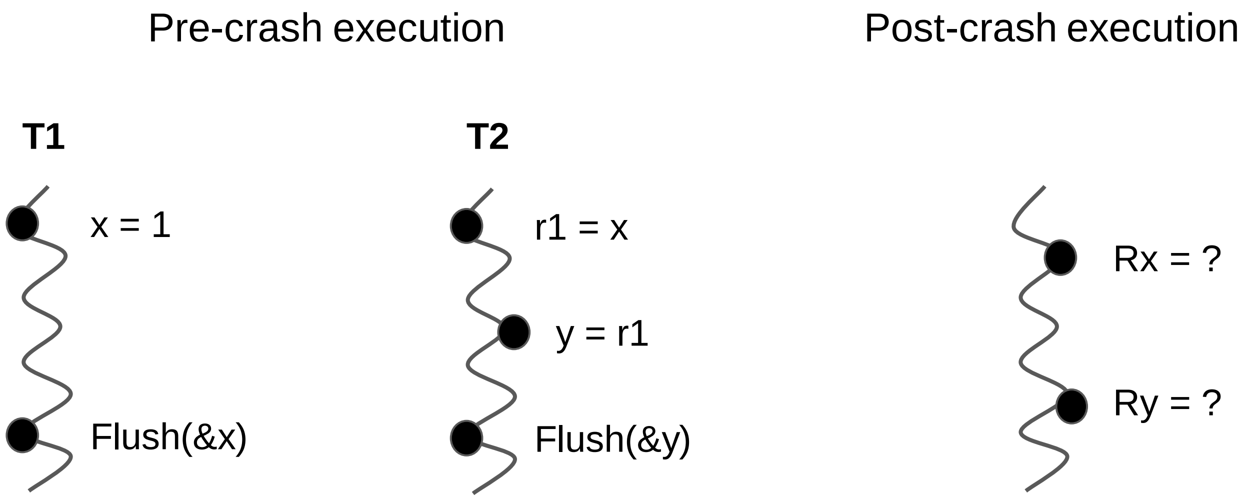
# Supporting Multi-threaded Programs




Pre-crash execution

Post-crash execution

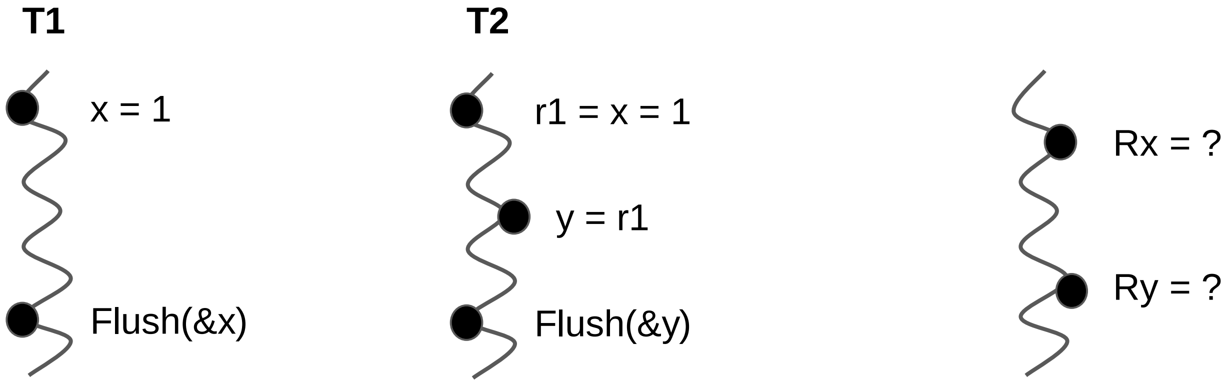
# Supporting Multi-threaded Programs



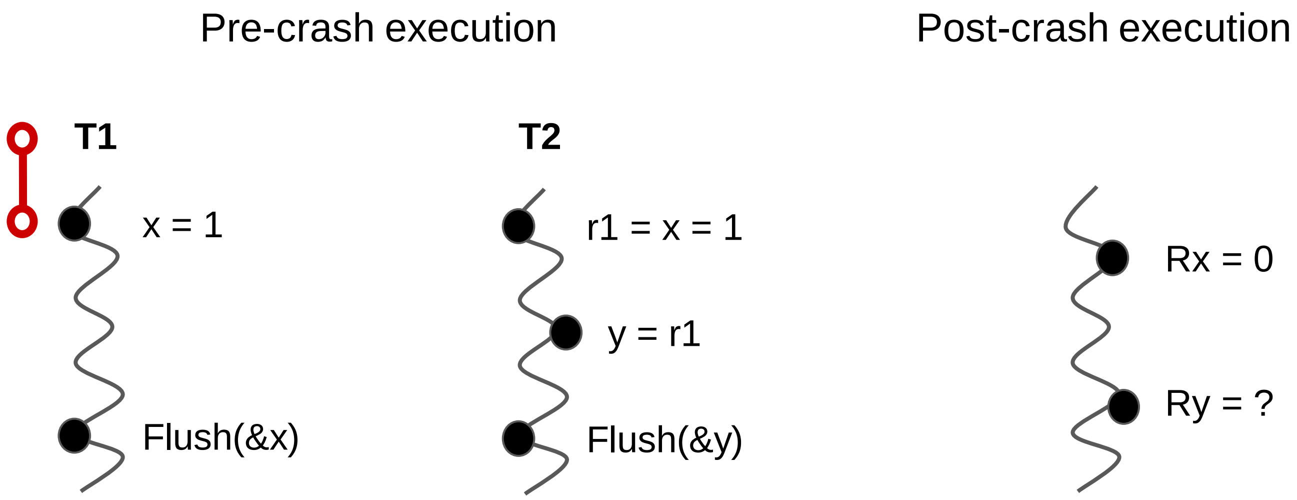
# Supporting Multi-threaded Programs

Pre-crash execution

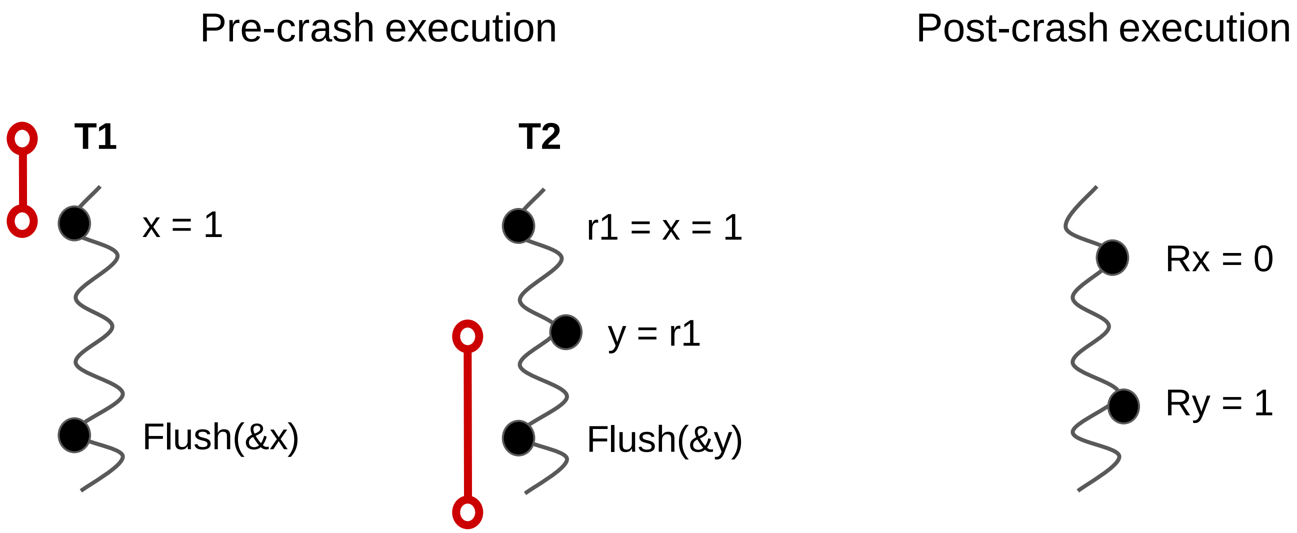
Post-crash execution



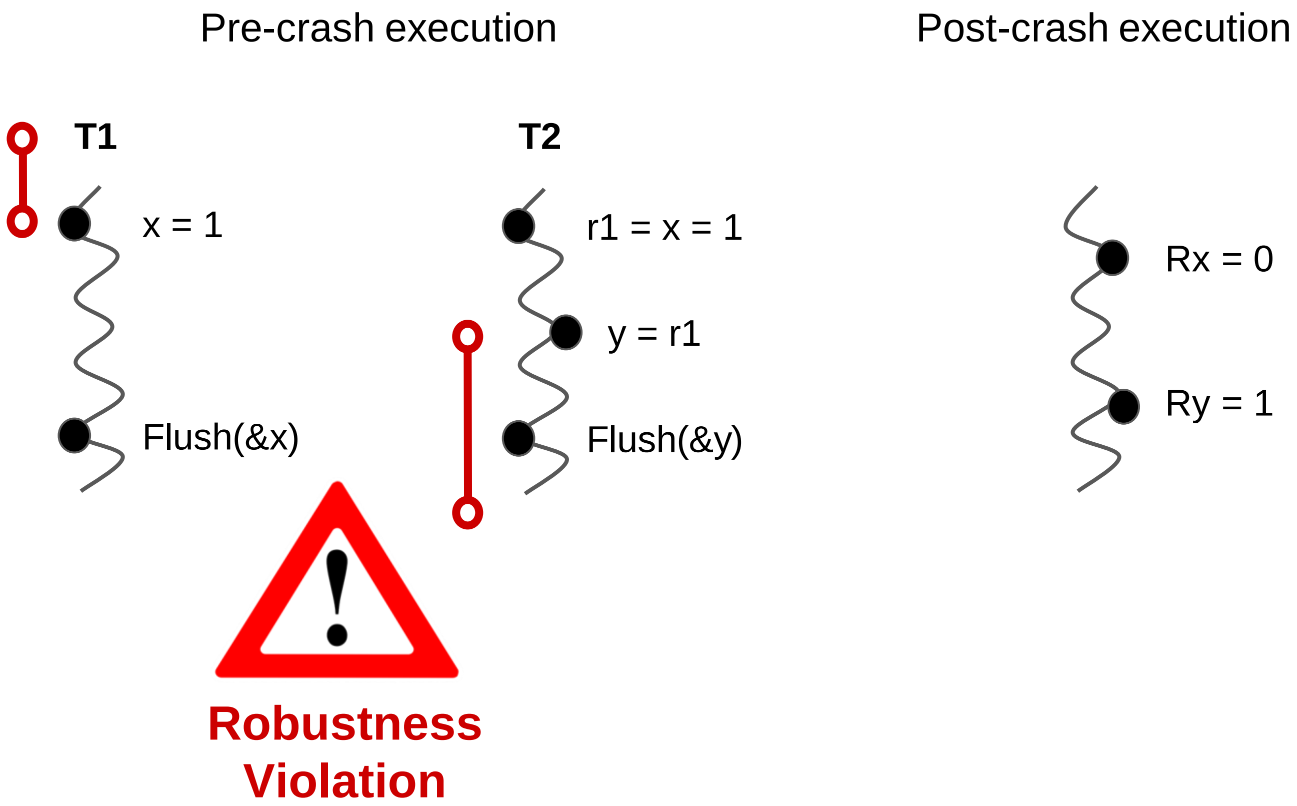
# Supporting Multi-threaded Programs



# Supporting Multi-threaded Programs



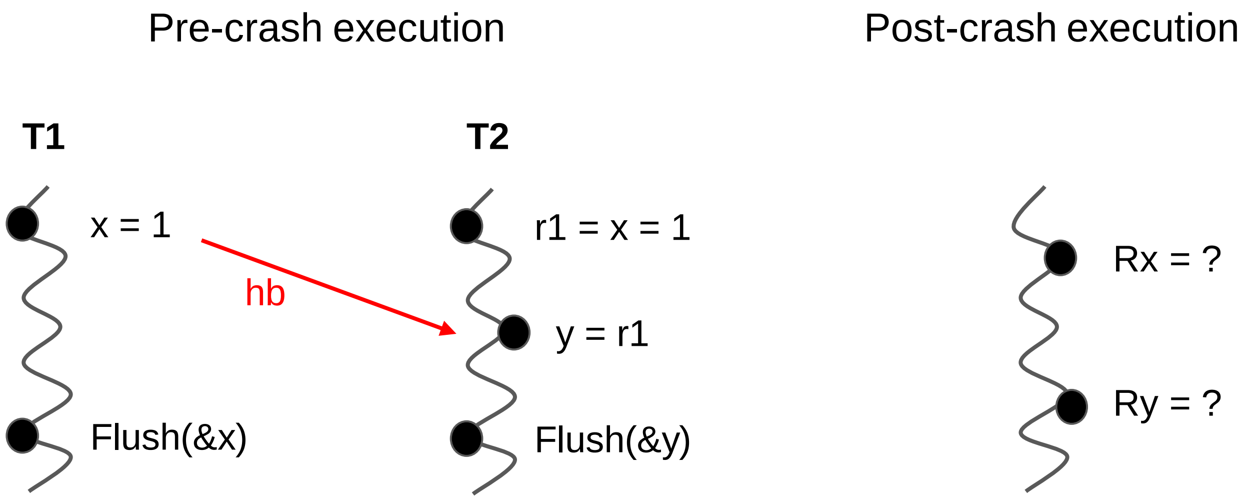
# Supporting Multi-threaded Programs



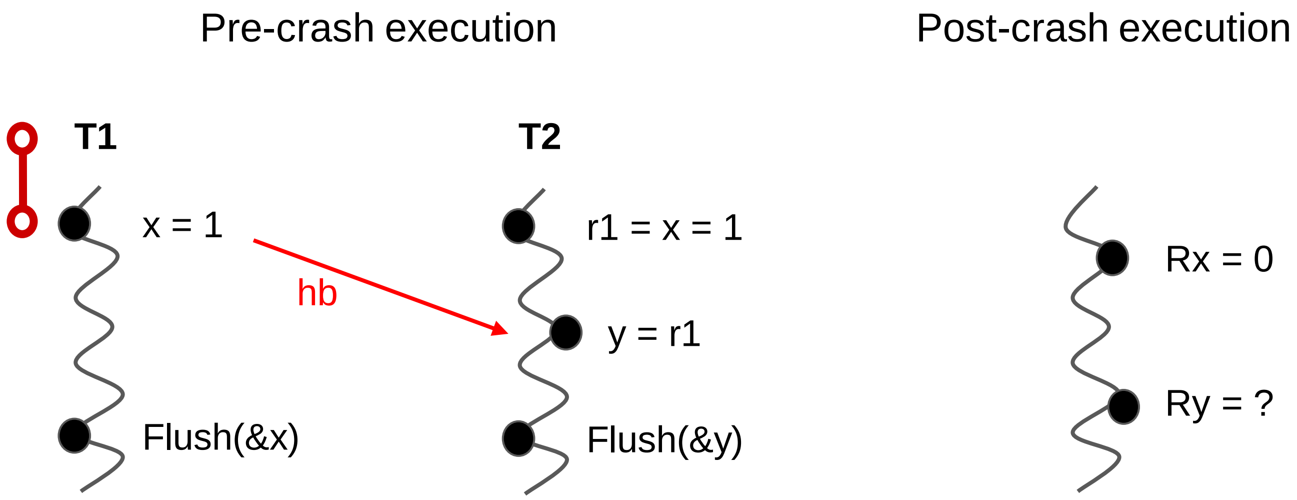
# Supporting Multi-threaded Programs

- 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

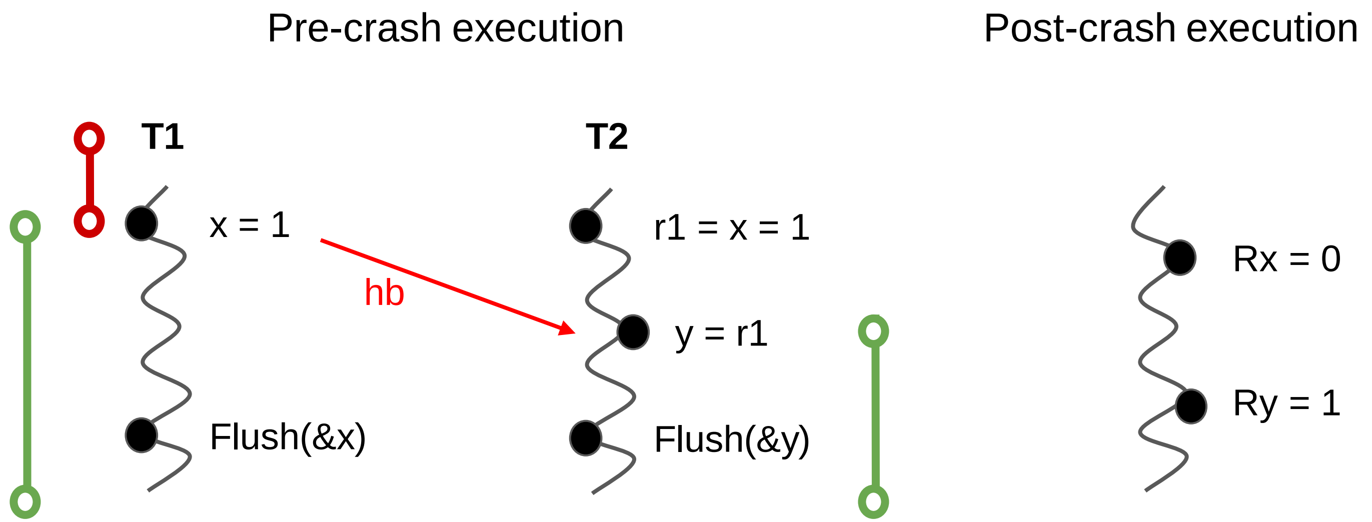
# Supporting Multi-threaded Programs



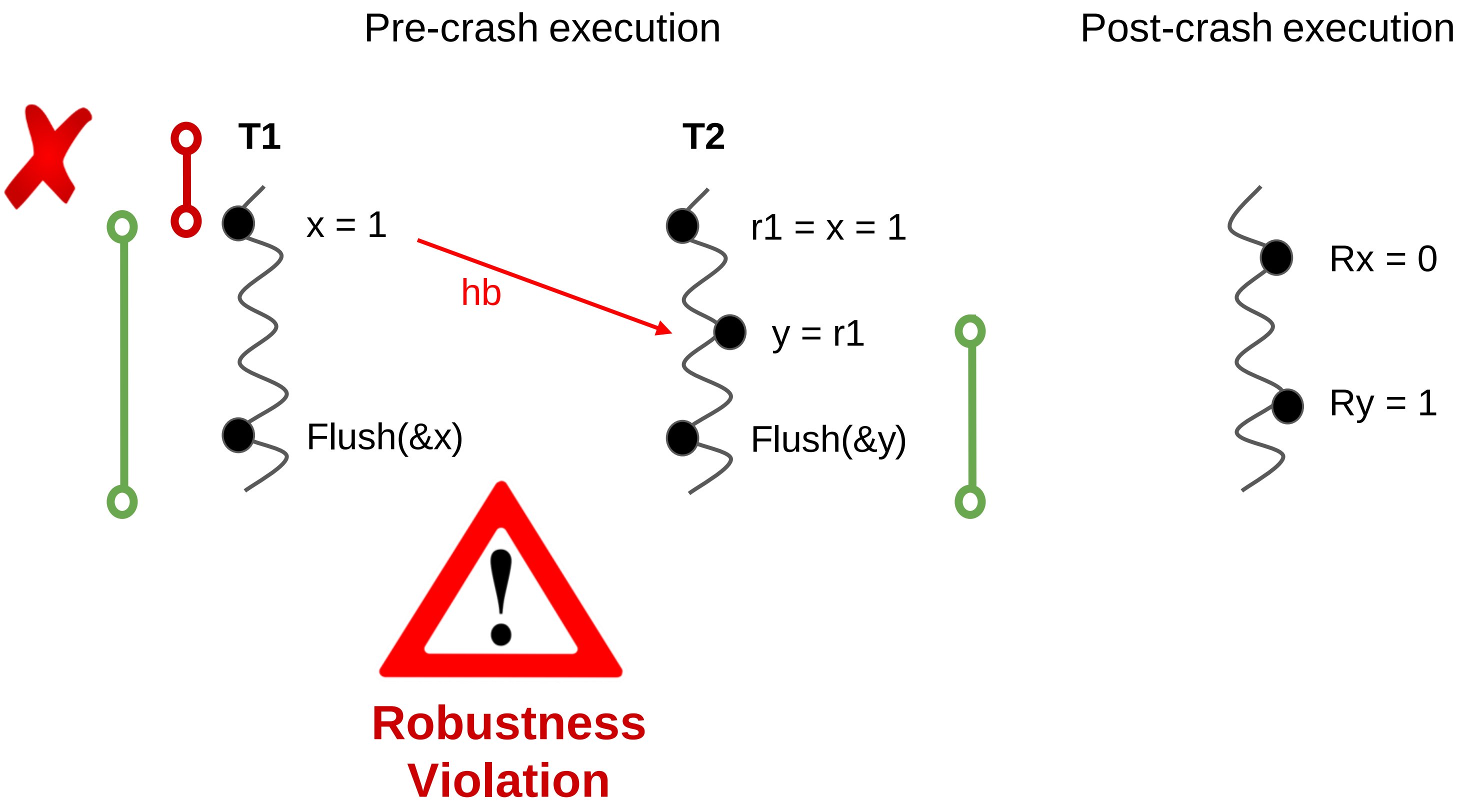
# Supporting Multi-threaded Programs



# Supporting Multi-threaded Programs



# Supporting Multi-threaded Programs



# Suggesting Fixes for Robustness Violations

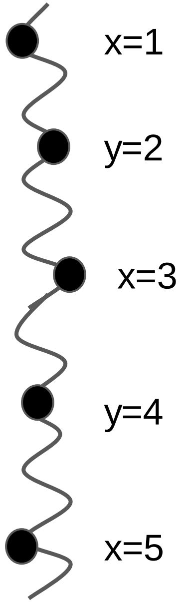
- Defining a fix as a set of flush intervals

Two cases for robustness violations:

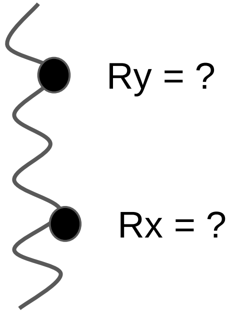
1. Reading from too old of store
2. Reading from too new of store

# Suggesting Fixes for Robustness Violations

Pre-crash execution



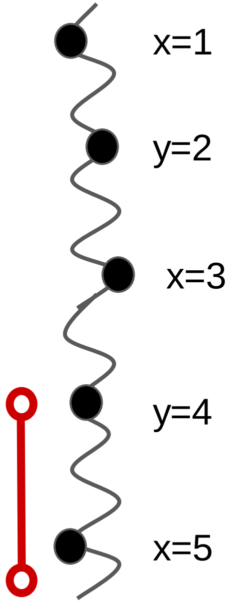

Post-crash execution



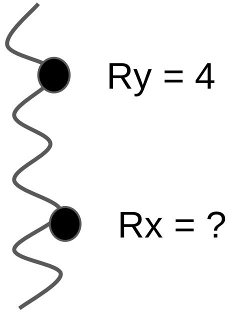

**Case 1: Reading from too old of store**

# Suggesting Fixes for Robustness Violations

Pre-crash execution




Post-crash execution

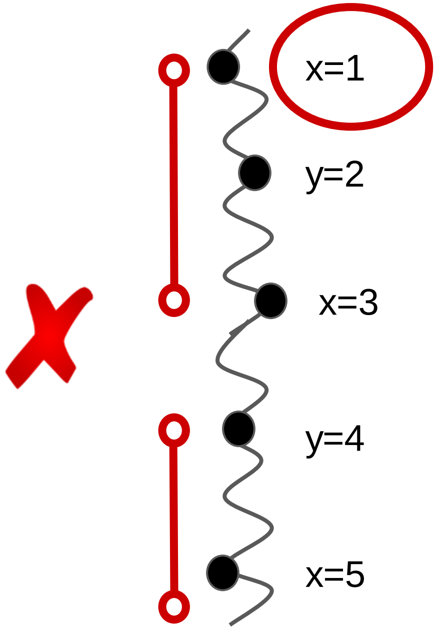



**Case 1: Reading from too old of store**

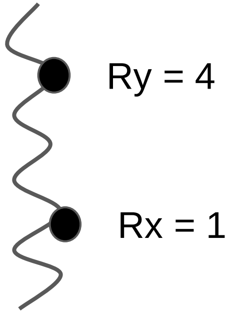
# Suggesting Fixes for Robustness Violations

Pre-crash execution

Post-crash execution



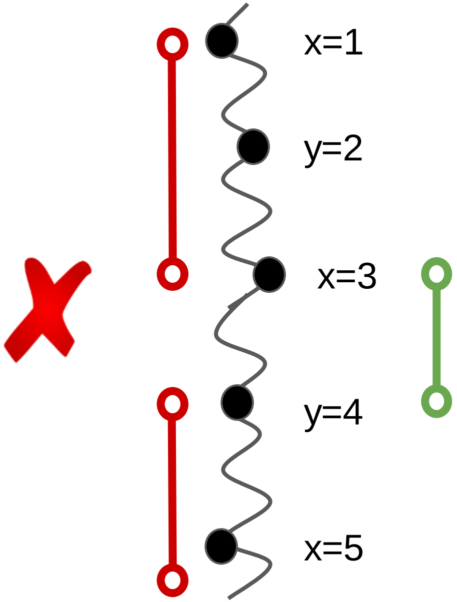

**Reads from a store that is too old**




**Case 1: Reading from too old of store**

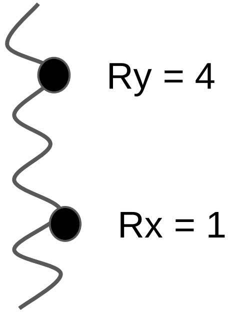
# Suggesting Fixes for Robustness Violations

Pre-crash execution




**Flush interval suggested by PSan for variable X**

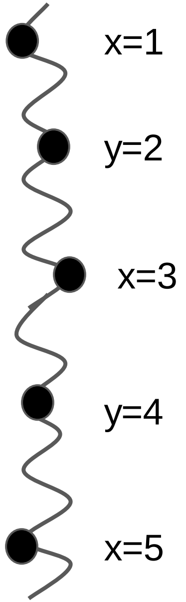
Post-crash execution



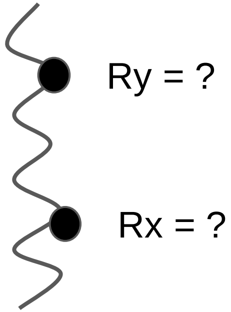

**Case 1: Reading from too old of store**

# Suggesting Fixes for Robustness Violations

Pre-crash execution



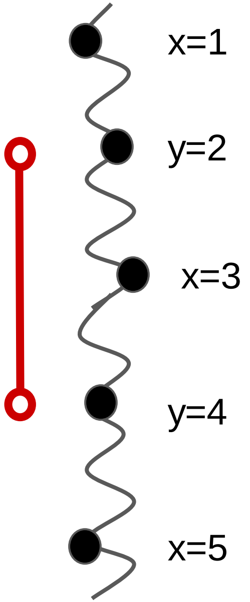

Post-crash execution



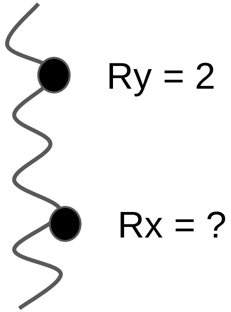

**Case 2: Reading from too new of store**

# Suggesting Fixes for Robustness Violations

Pre-crash execution



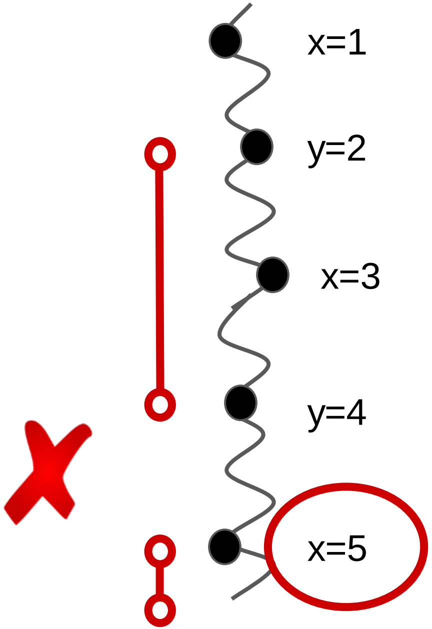

Post-crash execution



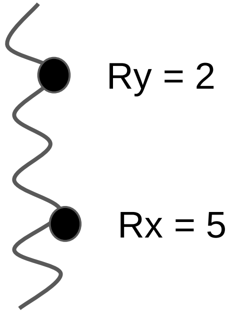

**Case 2: Reading from too new of store**

# Suggesting Fixes for Robustness Violations

Pre-crash execution




Post-crash execution

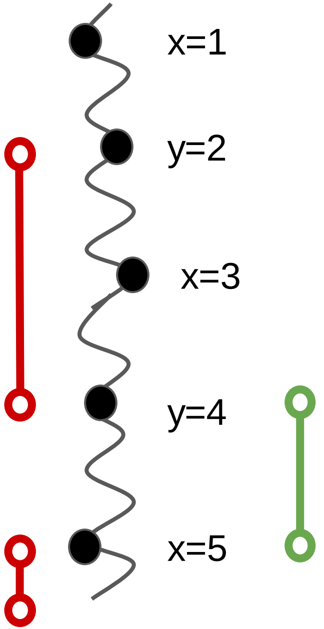



**Reads from a store that is too new**

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

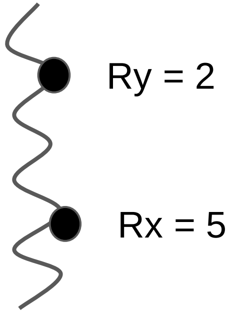
# Suggesting Fixes for Robustness Violations

Pre-crash execution



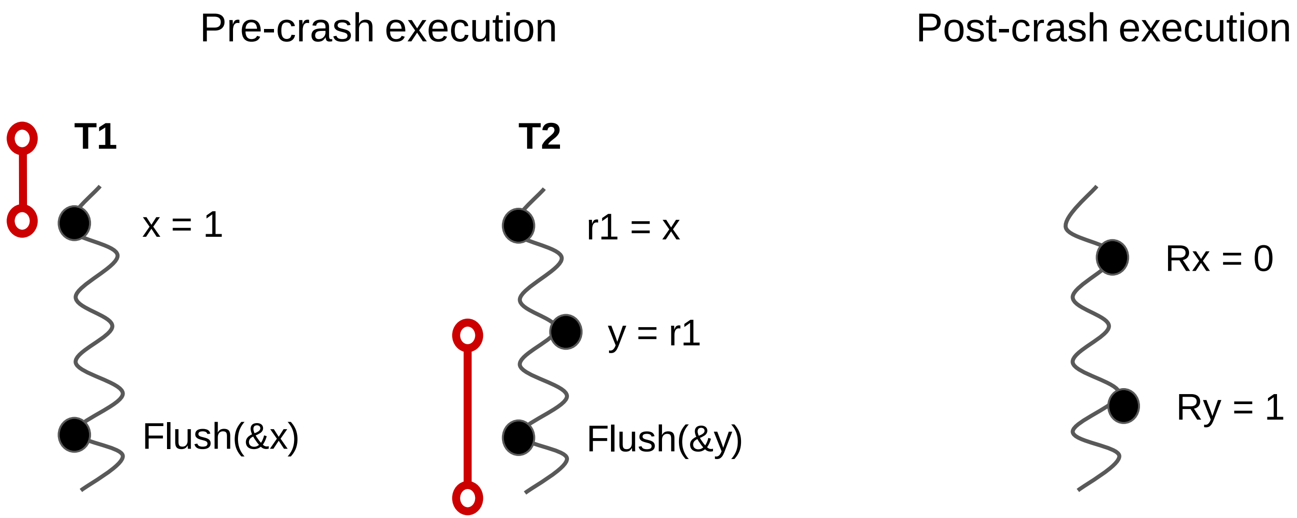

**Flush interval suggested by PSan for variable Y**

Post-crash execution




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

# Suggesting Fixes for Robustness Violations

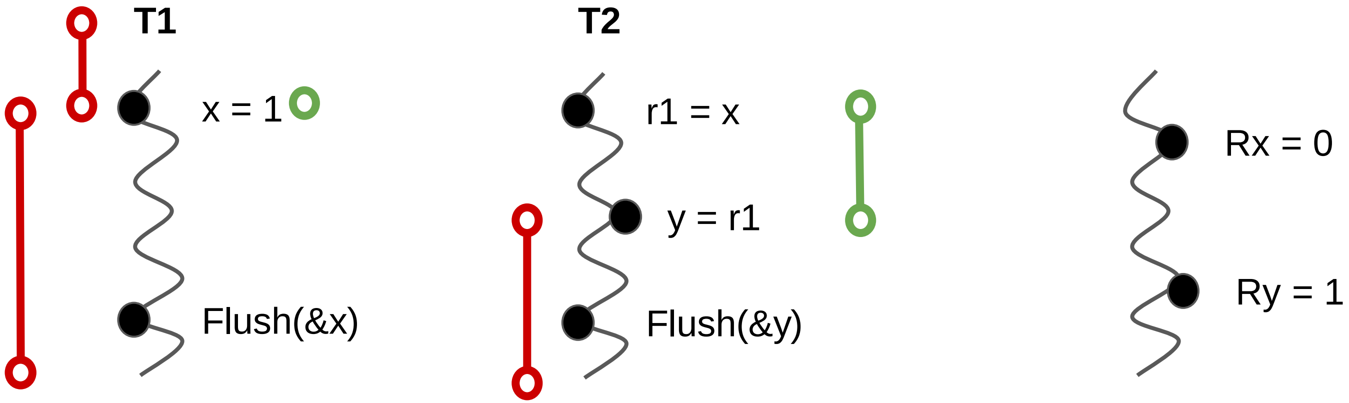



**Special case: Multi-threaded programs**

# Suggesting Fixes for Robustness Violations

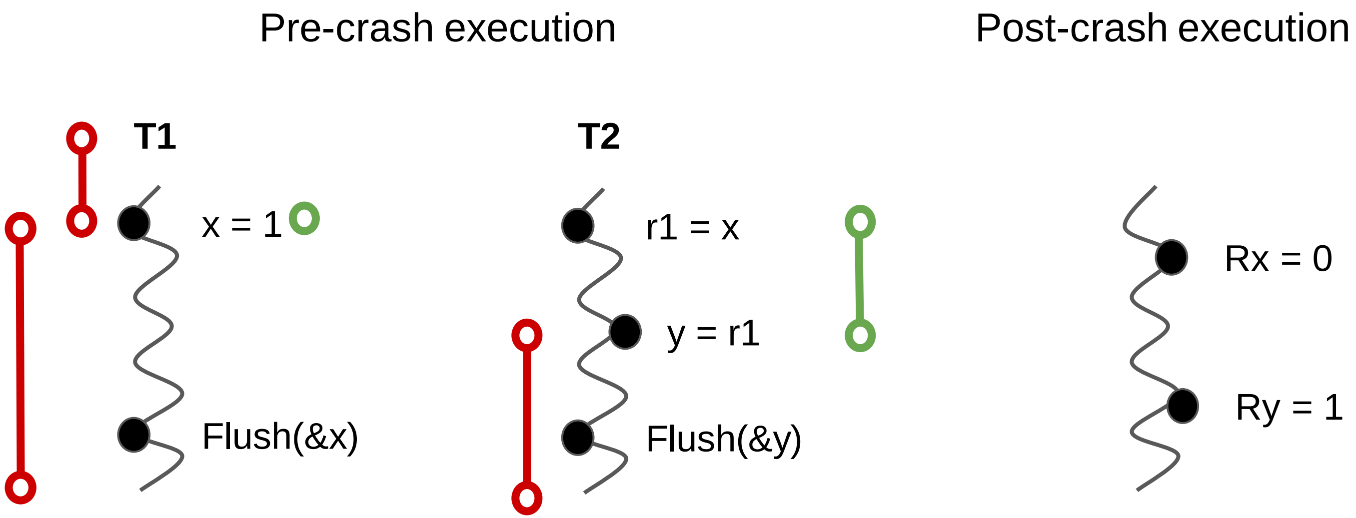
Pre-crash execution

Post-crash execution




**Special case: Multi-threaded programs**

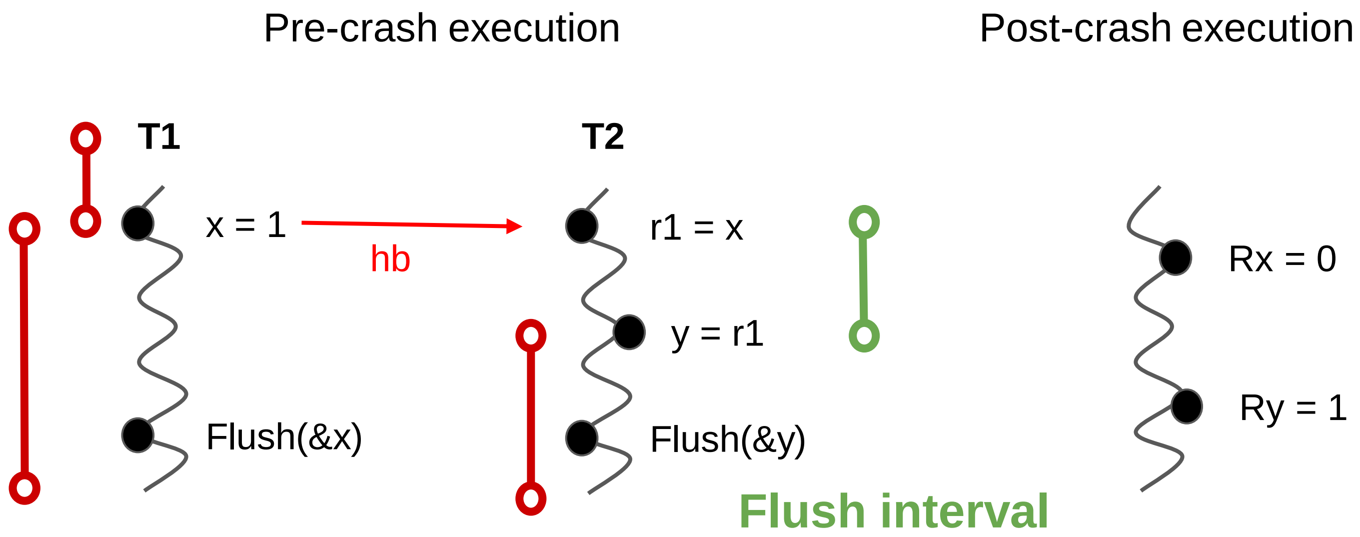
# Suggesting Fixes for Robustness Violations




**Flush interval is empty!!**

**Special case: Multi-threaded programs**

# Suggesting Fixes for Robustness Violations



# 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

```
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

```
class header{
        page* leftmost_ptr;
        page* sibling_ptr;
        uint32_t level;
        uint32_t switch_counter;
        std::mutex *mtx;
        union Key highest;
        uint8_t is_deleted;
        int16_t last_index;
        uint8_t dummy[5];
};
```

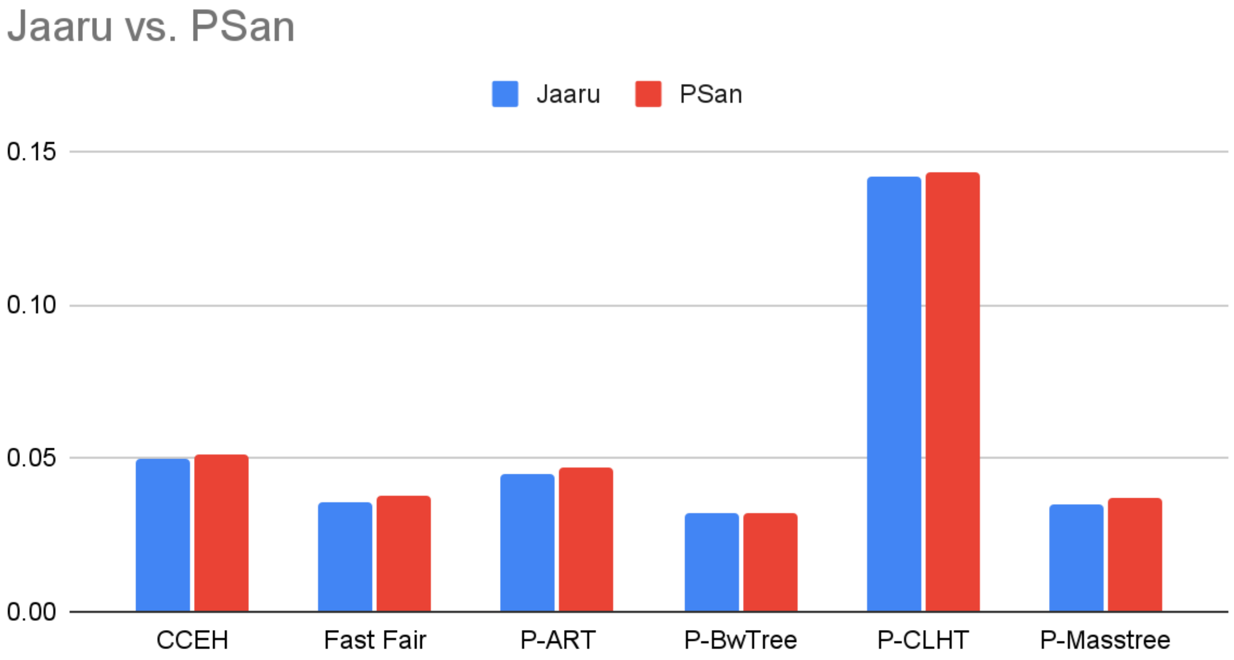
# Evaluation

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