USENIX ATC 2025

Inferring Likely Counting-related Atomicity Program Properties for Persistent Memory

Yunmo Zhang[1], Junqiao Qiu[1], Hong Xu[2], Chun Jason Xue[3]

[1]City University of Hong Kong [2]Chinese University of Hong Kong [3]MBZUAI

香港城市大學
City University of Hong Kong

香港中文大學
The Chinese University of Hong Kong

MOHAMED BIN ZAYED
UNIVERSITY OF
ARTIFICIAL INTELLIGENCE

Background - Persistent Memory

- **Advantages of PM**
  - Byte-addressable access like DRAM (e.g., Intel Optane, CXL-SSD).
  - Avoids storage stack overhead.

- **Crash Consistency Challenge**
  - Writes are buffered and then flushed to the PM in arbitrary order.
  - Programmers must use clflush/sfence or transaction interfaces (TXs) to ensure crash consistency, but this is error-prone.

PM Crash Consistency

- For a sequential program, typical types of **requirements** for achieving consistent PM program states after crash include:
  - Durability: A `Store` persists before the end of program.
  - Ordering: A `Store` to $addr_1$ persists *before* a `Store` to $addr_2$.
  - Atomicity: A set of `Store`s persist *together* (all or nothing).

Durability

```
ST(addr);
clflush(addr);
```

Ordering

```
ST & clflush(addr1);
sfence ();
ST & clflush(addr2);
```

Atomicity

```
Tx_begin ();
ST (addr1);
ST (addr2);
Tx_end();
```

PM testing tools are proposed…

From 2019 to 2025,

PMTest: A Fast and Flexible Testing Framework for Persistent Memory Programs

Sihang Liu, University of Virginia
Yizhou Wei, University of Virginia
Jishen Zhao, UC San Diego
Aasheesh Kolli, Penn S… VMw…
Samira Kh…

Fast, Flexible, and Comprehensive Bug Detection for Persistent Memory Programs

Bang Di
Jiawen Liu

Jaaru: Efficiently Model Checking Persistent Memory Programs

…msky
…ifornia, Irvine
…uci.edu

AGAMOTTO: How Persistent is your Persistent Memory Application?

Ben Reeves, *University of Michigan*
Ben Stoler, *University of Michigan*

Mumak: Efficient and Black-Box Bug Detection for Persistent Memory

João Gonçalves, Instituto Superior Técnico
Miguel Matos, Instituto Superior Técnico
Rodrigo Ro…, Instituto Superi…

Checking Robustness to Weak Persistency Models

…orjiara …Luo
Guoqing Harry Xu, University of California, Los Angeles, U.S.A.
Brian Demsky, University of California, Irvine, U.S.A.

WITCHER: Systematic Crash Consistency Testing for Non-Volatile Memory Key-Value Stores

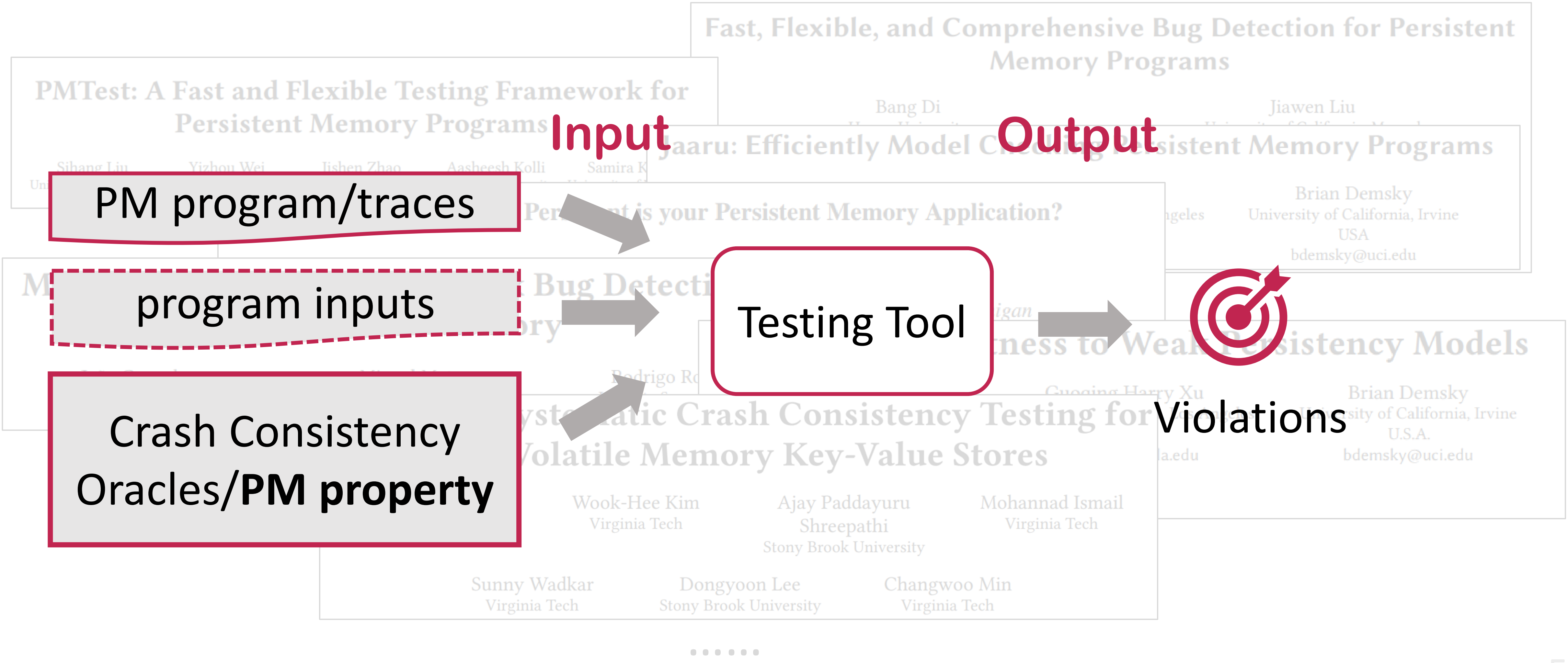
Xinwei Fu, Virginia Tech
Wook-Hee Kim, Virginia Tech
Ajay Paddayuru Shreepathi, Stony Brook University
Mohannad I…, Virginia Te…
Sunny Wadkar, Virginia Tech
Dongyoon Lee, Stony Brook University
Changwoo Min, Virginia Tech

Robustness Verification for Checking Crash Consistency of Non-volatile Memory

Zhilei Han, School of Software, Tsinghua University
Fei He, School of Software, Tsinghua University

……

PM testing tools are proposed…



Specifying PM Property

- Method 1: User Annotation - Time-consuming and still error-prone.
  - PMTest[ASPLOS ’19], XFDetector[ASPLOS ’20], Agamotto[OSDI’20], PMDebugger[ASPLOS ’21].

CityU HONG KONG

Specifying PM Property

- Method 1: User Annotation - Time-consuming and still error-prone.
  - PMTest[ASPLOS ’19], XFDetector[ASPLOS ’20], Agamotto[OSDI’20], PMDebugger[ASPLOS ’21].
- Method 2: Persistency Model - Only provide ordering properties.
  - Strict Persistency
  - Robustness: reducing persistency to memory consistency model.
    - Strict (PSan [PLDI ’22]), TSO (PMVerify [ASPLOS ’25]).

Specifying PM Property

- Method 1: User Annotation - Time-consuming and still error-prone.
  - PMTest[ASPLOS ’19], XFDetector[ASPLOS ’20], Agamotto[OSDI’20], PMDebugger[ASPLOS ’21].
- Method 2: Persistency Model - Only provide ordering properties.
  - Strict Persistency
  - Robustness: reducing persistency to memory consistency model.
    - Strict (PSan [PLDI ’22]), TSO (PMVerify [ASPLOS ’25]).
- **Method 3: Inferring PM property**
  - Based on dependency patterns
  - Witcher [SOSP ’21], Huang et al. [ASE ’24].

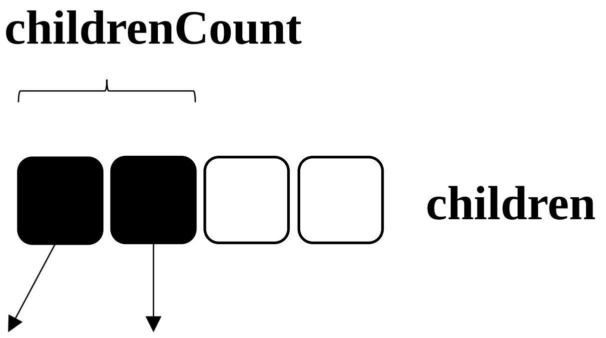
+ Covering part of Atomicity Properties

- Fail to infer critical atomicity properties *without explicit dependency*.

Counting-correlated Variables

- An example:
  - **childrenCount** tracks the number of valid pairs of key and child pointer in a **children** node in persistent adaptive radix tree.

```
N4::getChildren (…) {
        ...
        children[childrenCount] = std::make_tuple(key, child);
        childrenCount++;
        ...
}
```

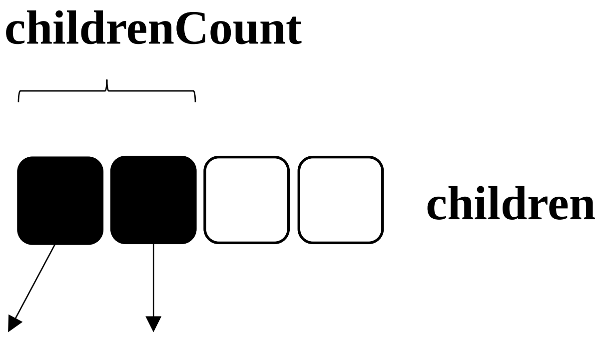


Counting-correlated Variables

- An example:
  - **childrenCount** tracks the number of valid pairs of key and child pointer in a **children** node in persistent adaptive radix tree.

```
N4::getChildren (…) {
        Tx_begin();
        children[childrenCount] = std::make_tuple(key, child);
        childrenCount++;
        Tx_end();
}
```

They should be in the same TX

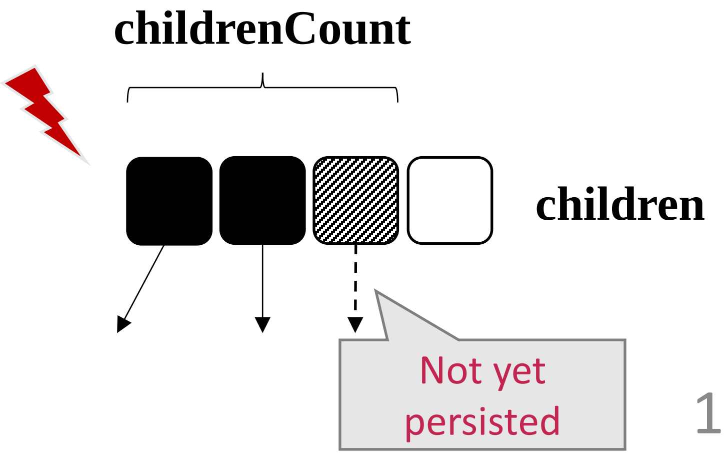


Counting-correlated Variables

- An example:
  - **childrenCount** tracks the number of valid pairs of key and child pointer in a **children** node in persistent adaptive radix tree.
  - Without **atomic** persistence, upon a crash,
    - partially persisting **childrenCount** leads to the return of dangling pointer

```
N4::getChildren (…) {
        Tx_begin();
        children[childrenCount] = std::make_tuple(key, child);
        childrenCount++;
        Tx_end();
}
```

They should be in the same TX

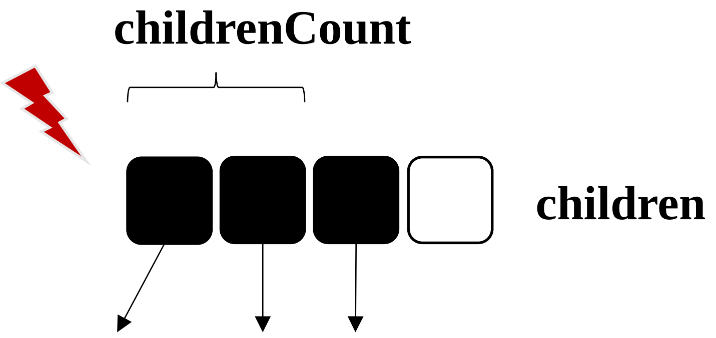


Counting-correlated Variables

- An example:
  - **childrenCount** tracks the number of valid pairs of key and child pointer in a **children** node in persistent adaptive radix tree.
  - Without **atomic** persistence, upon a crash,
    - partially persisting **childrenCount** leads to the return of dangling pointer
    - partially persisting **children** leads to data loss

```
N4::getChildren (…) {
        Tx_begin();
        children[childrenCount] = std::make_tuple(key, child);
        childrenCount++;
        Tx_end();
}
```

They should be in the same TX



Counting-related Atomicity Property

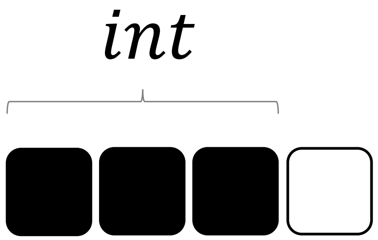
- An atomic persistence requirement for variables with relationship between:

container-like array(s)

integer variable(s) that tracks a numerical value about the logical size(s)

Counting-related Atomicity Property

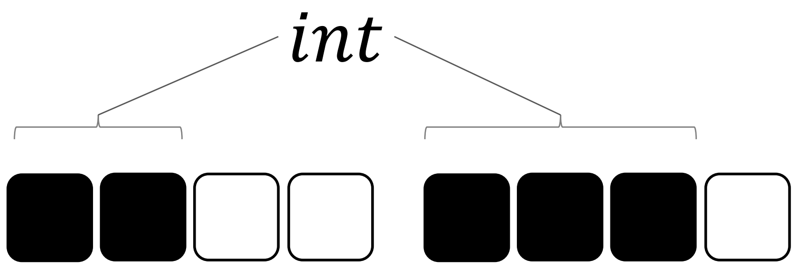
- An atomic persistence requirement for variables with relationship between:
  - a) the container-like array(s)
  - b) integer variable(s) that tracks a numerical value about the logical size(s) of array(s)



- Under three scenarios:
  1) the logical size of an array
  2) the sum of the logical sizes of multiple arrays
  3) the complementary size of an array to a constant

Counting-related Atomicity Property

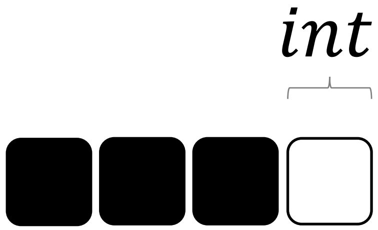
- The atomic persistence requirement for variables with relationship between:
  - a) the container-like array(s)
  - b) integer variable(s) that tracks a numerical value about the logical size(s) of array(s)



- Under three scenarios:
  1) the logical size of an array
  2) the sum of the logical sizes of multiple arrays
  3) the complementary size of an array to a constant

Counting-related Atomicity Property

- The atomic persistence requirement for variables with relationship between:
  - a) the container-like array(s)
  - b) integer variable(s) that tracks a numerical value about the logical size(s) of array(s)



- Under three scenarios:
  1) the logical size of an array
  2) the sum of the logical sizes of multiple arrays
  3) the complementary size of an array to a constant

Prevalence of Counting Correlation

- Exist in many PM Data Structures, e.g.,
  - Trees: child pointers array and its length/size of valid elements
  - Ring buffers: buffer and its head/tail offsets
  - Hash tables: table and its capacity

- Ever found bugs in other storage stacks
  - btrfs: i_size mismatched with actual file size after fsync [1]
  - ext4: i_disksize inconsistent with actual data size after crash [2]

[1] patchwork.kernel.org/project/linux-btrfs/patch/1434541763-23753-1-git-send-email-fdmanana@kernel.org/, 2015.
[2] https://marc.info/?l=linux-ext4&m=151669669030547&w=2, 2018.

Why existing methods are limited

- Existing PM atomicity property inference efforts:
  - Witcher [3], Huang et al. [4].
  - Infer properties from control dependency patterns.

	Multi-control Dependency [3]	Inter-control Dependency [4]
Dependency Pattern	`if (x) then m···` $(m \xrightarrow[\text{dep}]{\text{ctrl}} x)$ `if (y) then n···` $(n \xrightarrow[\text{dep}]{\text{ctrl}} y)$	`if (x) then y···` $(y \xrightarrow[\text{dep}]{\text{ctrl}} x)$ `if (y) then x···` $(x \xrightarrow[\text{dep}]{\text{ctrl}} y)$
Inferred Likely Atomicity Property	ATOMICITY(x, y)	ATOMICITY(x, y)

[3] Fu et al. Witcher: Systematic crash consistency testing for non-volatile memory key-value stores. SOSP (2021).
[4] Huang et al. Discovering likely program invariants for persistent memory. ASE (2024).

Why existing methods are limited

- Existing PM atomicity property inference efforts:
  - Witcher [3], Huang et al. [4].
  - Infer properties from control dependency patterns.

```
N4::getChildren (...) {
    ...
    children[childrenCount] = std::make_tuple(key, child);
    childrenCount++;
    ...
}
```

```
Tree::lookupRange (...) {
    ...
    for (uint32_t i = 0; i < childrenCount; ++i) {
        const N *n = std::get<1>(children[i]);
    ...}
}
```

children $\xrightarrow[dep]{ctrl}$ childrenCount

? $\xrightarrow[dep]{ctrl}$ children

Multi-control Dependency [3]	Inter-control Dependency [4]
if(x) then m··· $(m \xrightarrow[dep]{ctrl} x)$	if(x) then y··· $(y \xrightarrow[dep]{ctrl} x)$
$\xrightarrow[dep]{ctrl} y)$	if(y) then x··· $(x \xrightarrow[dep]{ctrl} y)$

Dependency analysis hardly captures the behaviors of container-like variables, since they rarely act as "guardians" in conditionals.

Main Idea

Infer from dependency patterns?

Infer by dynamic statistics? (see paper)

Infer by directly capturing the semantics!

Main Idea

- Problem:
  - Expressing the semantics of counting-related variables is not straightforward.
  - As the value of *int* is not always equal to the logical size it is intended to represent throughout the program.

Infer by directly capturing the semantics!

```
// inserting to an array with N elements p
for(i = size - 1; i >= p; i--){
    array[i + 1] = array[i];
}
array[p] = 20;
size += 1;
```

array's logical size: N+1
size's value: N

Our approach

CityU HONG KONG

Q1. How to capture the semantics of counting correlation?

- We observe the **predictable access range behaviors** of the counting-correlated variables, named as Access Range Invariants (predicates)

Q2. How to infer the counting-related PM properties?

- Use symbolic analysis to extract access range behaviors.
- Validate the counting-correlated variables by SMT constraint solving.

Our approach

Q1. How to capture the semantics of counting correlation?

- Observation: All reads to the container ARR have address within the area restricted by the value of its logical size variable int.

necessary condition

- Because read behaviors encode the programmer's intent for acquiring the valid elements in a container.

```
// inserting to an array with N elements
for(i = size - 1; i >= p; i--){
    array[i + 1] = array[i];
}
array[p] = 20;
size += 1;
```

i: [p, size)

Our approach

Q1. How to capture the semantics of counting correlation?

- The read access range invariant (predicate) for scenario 1 is:

$$\forall \rho \in P, Read_{\rho}(ARR, idx) \Rightarrow idx < int_{\rho}$$

three scenarios:

1) the logical size of an array
2) the sum of the logical sizes of multiple arrays
3) the complementary size of an array to a constant

Our approach

Q1. How to capture the semantics of counting correlation?

- The read access range invariant (predicate) for scenario 1 is:

$$\forall \rho \in P, Read_\rho(ARR, idx) \Rightarrow idx < int_\rho$$

- Similarly, for scenario 2 (N ARRs),

$$\forall \rho \in P, \sum_{i \in [1,N] \wedge Read\rho(ARR_i, idx_i)} idx_i < int_\rho$$

three scenarios:

1) the logical size of an array
2) the sum of the logical sizes of multiple arrays
3) the complementary size of an array to a constant

Our approach

Q1. How to capture the semantics of counting correlation?

- The read access range invariant (predicate) for scenario 1 is:

$$\forall \rho \in P, Read_{\rho}(ARR, idx) \Rightarrow idx < int_{\rho}$$

- Similarly, for scenario 2 (N ARRs),

$$\forall \rho \in P, \sum_{i \in [1,N] \wedge Read\rho(ARR_i, idx_i)} idx_i < int_{\rho}$$

- For scenario 3,

$$\forall \rho \in P, Read_{\rho}(ARR, idx) \Rightarrow idx < C - int_{\rho}$$

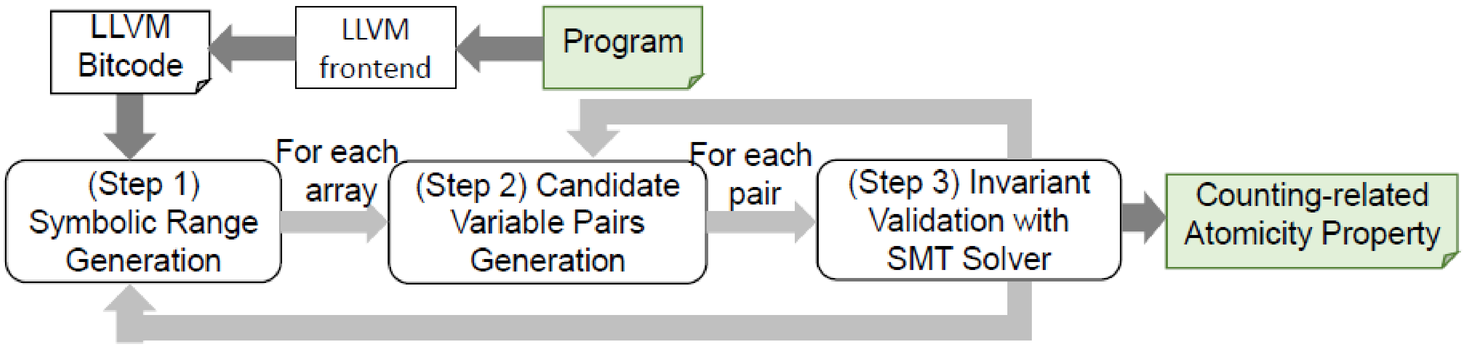
three scenarios:

1) the logical size of an array
2) the sum of the logical sizes of multiple arrays
3) the complementary size of an array to a constant

Inference Approach

Q2. How to infer the counting-related PM properties?

- Step 1: Exploit **Symbolic Range Analysis** [5] to extract symbolic values of all array access indices.
- Step 2: Filter out potential variables pairs/groups.
- Step 3: For each variable pair/group, validate if it satisfies the access range invariant (predicate) through constraint solving.

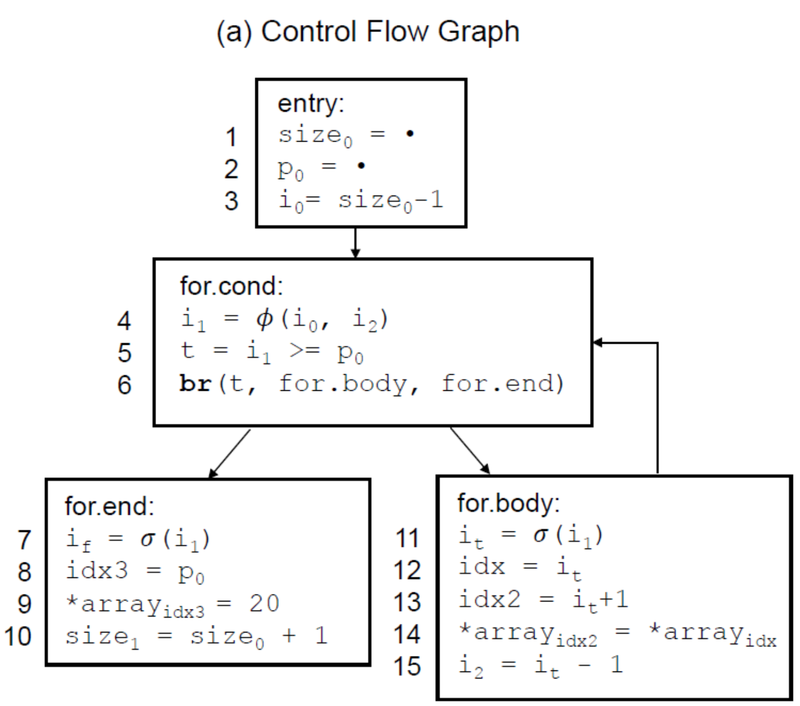


[5] Nazaré et al. Validation of memory accesses through symbolic analyses. OOPSLA, 2014.

Inference Approach

Q2. How to infer the counting-related PM properties?

- Step 1: Exploit **Symbolic Range Analysis** [5] to extract symbolic values of all array access indices.



(b) Symbolic Range

entry:

$[\![size_0]\!] = [\mathrm{size}_0, \mathrm{size}_0]$

$[\![p_0]\!] = [\mathrm{p}_0, \mathrm{p}_0]$

$[\![i_0]\!] = [\mathrm{size}_0 - 1, \mathrm{size}_0 - 1]$

for.cond:

$[\![i_1]\!] = [Min(\mathrm{p}_0 - 1, \mathrm{size}_0 - 1), \mathrm{size}_0 - 1]$

for.body:

$[\![i_t]\!] = [\mathrm{p}_0, \mathrm{size}_0 - 1]$

$[\![idx]\!] = [\mathrm{p}_0, \mathrm{size}_0 - 1]$

$[\![idx2]\!] = [p_0 + 1, \mathrm{size}_0]$

$[\![i_2]\!] = [\mathrm{p}_0 - 1, \mathrm{size}_0 - 2]$

for.end:

$[\![i_f]\!] = [Min(\mathrm{p}_0 - 1, \mathrm{size}_0 - 1),\ Min(p_0 - 1, size_0 - 1)]$

$[\![idx3]\!] = [Min(p_0, size_0), p_0]$

$[\![size_1]\!] = [size_1, size_1]$

[5] Nazaré et al. Validation of memory accesses through symbolic analyses. OOPSLA, 2014.

Inference Approach

Q2. How to infer the counting-related PM properties?

- Step 2: For each potential array, filter potential integer variables.

(b) Symbolic Range

entry:
$[\![size_0]\!] = [\mathrm{size}_0, \mathrm{size}_0]$
$[\![p_0]\!] = [\mathrm{p}_0, \mathrm{p}_0]$
$[\![i_0]\!] = [\mathrm{size}_0 - 1, \mathrm{size}_0 - 1]$

for.cond:
$[\![i_1]\!] = [Min(\mathrm{p}_0 - 1, \mathrm{size}_0 - 1), \mathrm{size}_0 - 1]$

for.body:
$[\![i_t]\!] = [\mathrm{p}_0, \mathrm{size}_0 - 1]$
$[\![idx]\!] = [\mathrm{p}_0, \mathrm{size}_0 - 1]$
$[\![idx2]\!] = [p_0 + 1, \mathrm{size}_0]$
$[\![i_2]\!] = [\mathrm{p}_0 - 1, \mathrm{size}_0 - 2]$

for.end:
$[\![i_f]\!] = [Min(\mathrm{p}_0 - 1, \mathrm{size}_0 - 1),\ Min(p_0 - 1, size_0 - 1)]$
$[\![idx3]\!] = [Min(p_0, size_0), p_0]$
$[\![size_1]\!] = [size_1, size_1]$

{*array, p_0}
{*array, size_0}

Inference Approach

Q2. How to infer the counting-related PM properties?

- Step 3: For each variable pair, validate if it satisfies the access range invariant(s) through constraints solving.
- Invariant (INV) constraint:

$$\wedge_{idx \in R(ARR)} [\![idx]\!]_{\uparrow} < int$$

- To check the satisfaction across all possible values of int,

$$\frac{\neg INV}{}$$

Z3 →

Satisfiable: Invariant unsatisfied

Unsatisfiable: Invariant satisfied for all values

→ likely PM atomicity property

Evaluation

- Using inferred PM atomicity property to discover bugs in PM data structures.
  - Persistent Adaptive Radix Tree (P-ART)
  - Persistent BwTree (P-BwTree)
  - Dynamic Hashing for PM (CCEH)
  - Hash Indexing for PM (Level-hashing)

- Compared inference methods
  - Dependency-based approaches: Witcher [SOSP ’21], Huang et al. [ASE ’24].
  - Atomicity property inference for concurrent programs: MUVI [SOSP ’07].

Evaluation

- Using inferred PM atomicity property to discover bugs in PM data structures.
  - No bug detected by Huang el at.

PM Program	ID	New	Code	Description	Impact	MUVI	Witcher
P-ART	1	✓	N4.cpp:117	Creating an array of valid nodes	Fault or data loss		
	2		N4.cpp:22	Inserting a node to an array of children nodes	Fault or data loss		✓
	3	✓	N16.cpp:124	Creating an array of valid nodes	Fault or data loss		
	4	✓	N48.cpp:120	Creating an array of valid nodes	Fault or data loss		
	5	✓	N256.cpp:81	Creating an array of valid nodes	Fault or data loss		
	6		N16.cpp:13	Inserting a node to an array of children nodes	Fault or data loss		✓
	7	✓	Epoch.cpp:57	Adding to an array of fixed size arrays	Fault or data loss	✓	
P-BwTree	8	✓	bloom_filter.h:143	Inserting an element to a "ValueType" array	Stale read or data loss	✓	
CCEH	9	✓	CCEH_LSB.cpp:220	Resizing an array before insertions.	Fault or data loss	✓	
	10	✓	linear_probing.cpp:151	Resize a hash table	Memory corruption		
	11	✓	extendible_hash.cpp:329	Resizing an array before insertions.	Fault or data loss	✓	
	12	✓	cuckoo_hash.cpp:295	Resizing a "table" array	Memory corruption		
Level-Hashing	13		level_hashing.c:112	Expanding a level hash table	Memory corruption		✓
	14	✓	level_hashing.c:226	Shrinking a level hash table	Corruption or data		

  - Analysis Time: < 1 second for each program

Summary

CityU HONG KONG

- Observe a counting-related atomicity requirements for the crash consistency of PM programs.
- Propose to use predictable read access range to encode the semantics of counting-correlated variables.
- Design an inference approach based on symbolic analysis and SMT constraint solving.
- Found 14 atomicity bugs (11 new) from PM programs using the inferred properties.

yunmo.zhang@my.cityu.edu.hk