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

Xinwei Fu[*], Wook-Hee Kim[*], Ajay Paddayuru[+], Mohannad Ismail[*], Sunny Wadkar[*], Dongyoon Lee[+], Changwoo Min[*]

[*] VIRGINIA TECH™ [+] Stony Brook University

Summary

- NVM enables writing crash consistent programs without paying storage overhead
- Writing crash consistent programs is error-prone

- Existing NVM bug detectors
  - Exhaustive Searching
  - User-provided Oracles

- Witcher
  - NVM-backed Key-Value Stores
  - Inference of Likely-Correctness Conditions
  - Validation with Output Equivalence Checking
  - Detected 205 (149 new) correctness/performance bugs

Outline

1. ***Background and Motivation***
2. Witcher
3. Evaluation
4. Conclusion

Finally NVM is here to stay but ...

NVM Characteristics:

- Persistence
- Low access latency
- Byte-addressability
- High capacity

Crash Consistency

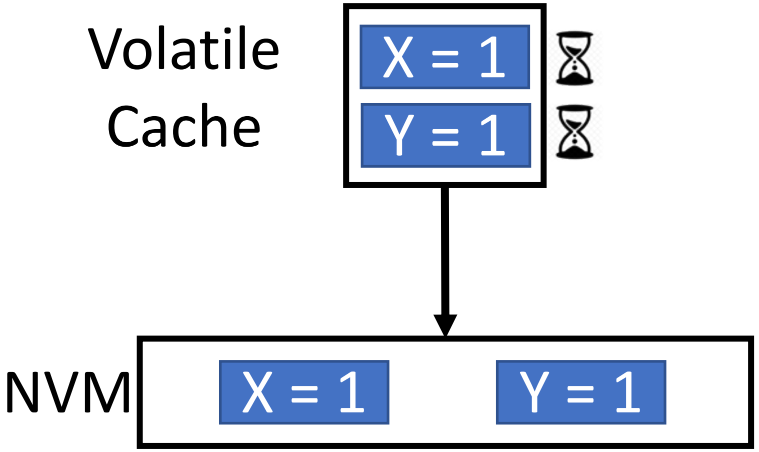
Applications can recover a consistent state from NVM in the event of a crash (e.g., power failure)

Challenges in NVM Programming

- “Volatile” cache states may be lost upon a crash
- Controlling the durability and the ordering for cachelines is the key

Controlling Durability and Ordering

- flush (x86: clwb): write back a cache line from cache to memory
- fence (x86: sfence): ordering guarantee between flushes



1. X = 1
2. flush (&X)
3. fence()
4. Y = 1
5. flush (&Y)
6. fence()

Durability and *Ordering*

Persistence Bugs

- Persistence Ordering Bug
  : Fail to enforce that "A" must be persisted before "B"

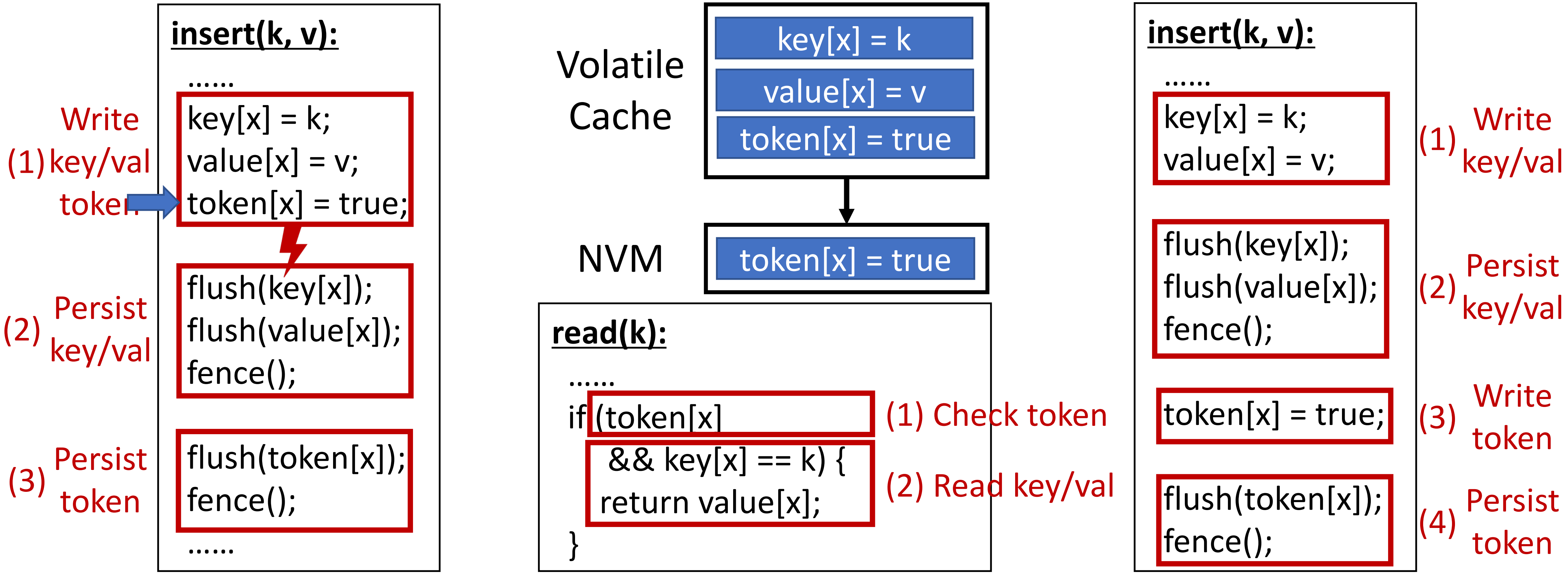
- Persistence Atomicity Bug
  : Fail to enforce that "A" and "B" must be persisted together

- Persistence Performance Bug
  : e.g., extra flush/fence

Persistence Ordering Bug

: Fail to enforce that “A” must be persisted before “B”.

LevalHash [OSDI’18]: Each bucket has arrays of Keys, Values, Tokens (valid flags)



Application-specific knowledge is required to detect this bug!

Persistence Atomicity Bug

: Fail to enforce that “A” and “B” must be persisted together

LevalHash [OSDI’18]: Each bucket has arrays of Keys, Values, Tokens (valid flags)

update(k, v):

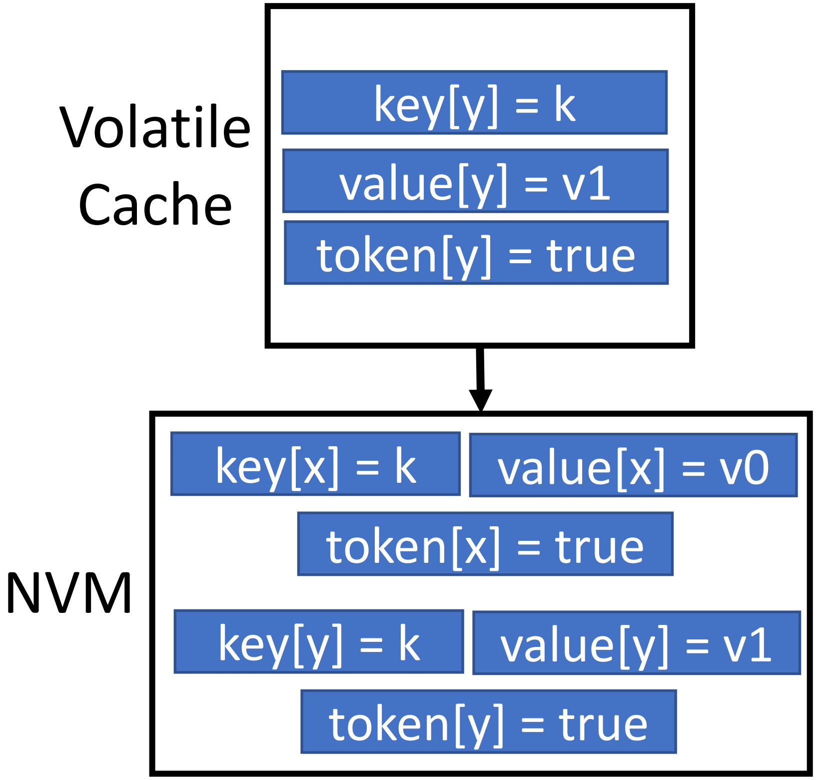
```
……
key[y] = k;
value[y] = v1;
token[y] = true;

token[x] = false;

flush(key[y]);
flush(value[y]);
fence();

flush(token[x]);
fence();
……
```

(1) Write new key/val token

(2) Invalidate old token

(3) Persist new key/val

(4) Persist tokens



update(k, v):

```
……
key[y] = k;
value[y] = v1;

flush(key[y]);
flush(value[y]);
fence();

token = ……;

flush(token);
fence();

……
```

(1) Write new key/val

(2) Persist new key/val

(3) Update 2 tokens in an 8-byte writes

(4) Persist tokens

Existing Works

- Exhaustive searching:
  - Yat[ATC'14]
  - Enumerate all possible crash states
  - [Pros] no false negative
  - [Cons] not scalable

- User-provided oracles:
  - PMTest[ASPLOS'19], XFDetector[ASPLOS'20], Agamotto[OSDI'20]
  - Rely on users' guidance to validate a crash state
  - [Pros] make bug validation process simple
  - [Cons] manual efforts, error-prone, especially for application-specific oracles

<u>Witcher uses neither exhaustive searching nor user-provided oracles.</u>

Outline

1. Background and Motivation
2. ***Witcher***
3. Evaluation
4. Conclusion

Key Idea 1: Likely-Correctness Conditions

- Challenge: How can we prune the test space without user-provided oracles?

Infer likely-correctness conditions from code

```
read(k):
  ……
  if (token[x]
      && key[x] == k) {
      return value[x];
  }
```

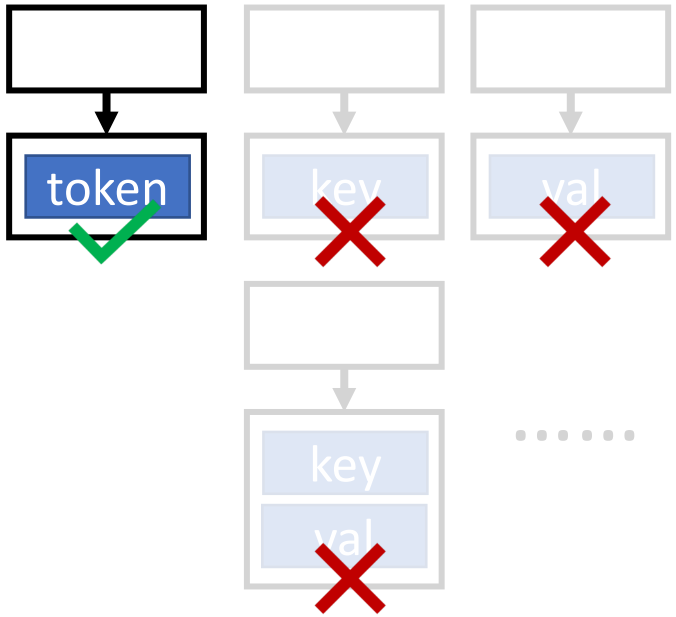
Hint: Check token first then read key-val
(Control dependence: token & key-val)

Likely-correctness condition:
Persist key-val before updating token

Only test crash states violating inferred conditions

```
insert(k, v):
  ……
  key[x] = k;
  value[x] = v;
  token[x] = true;

  flush(key[x]);
  flush(value[x]);
  fence();

  flush(token[x]);
  fence();
```

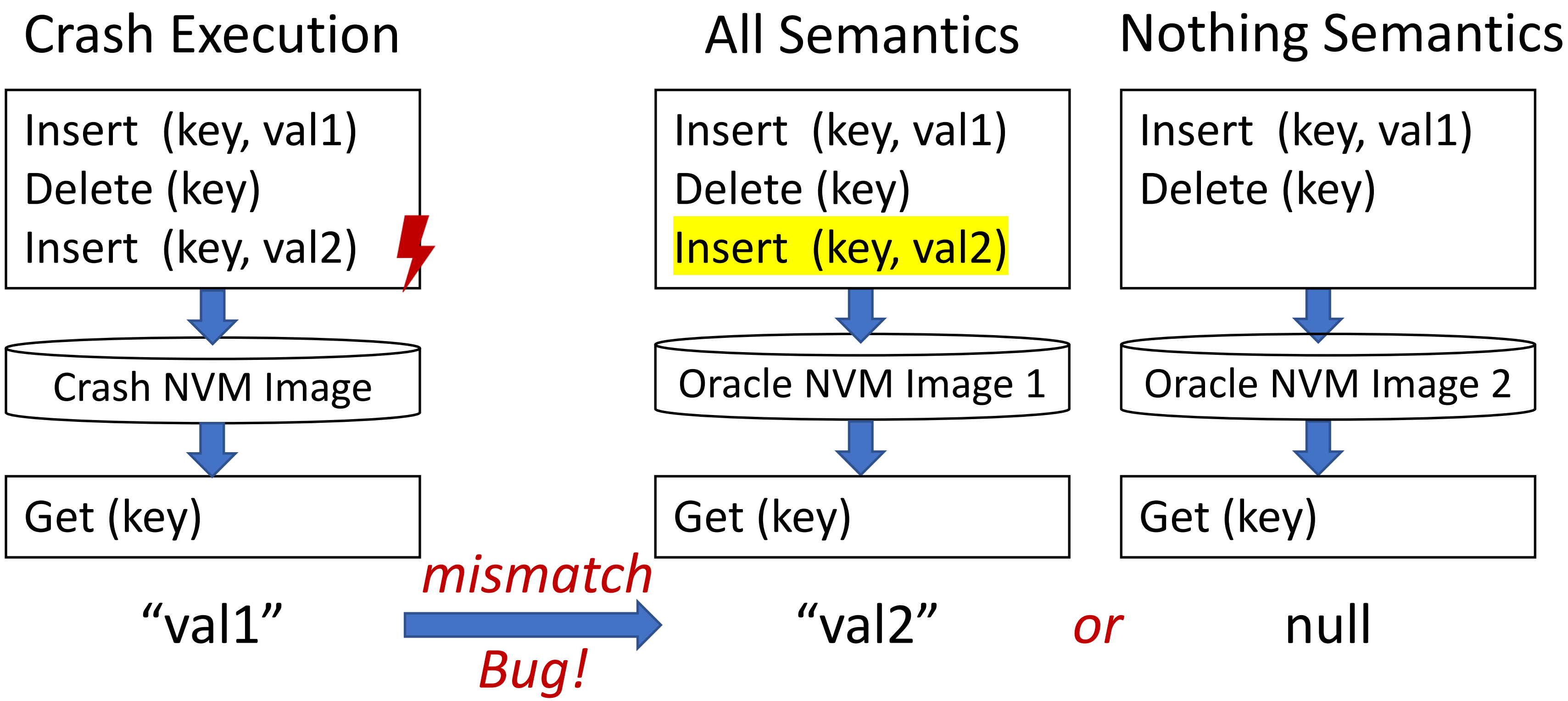
Testing a crash state where
only token is persisted



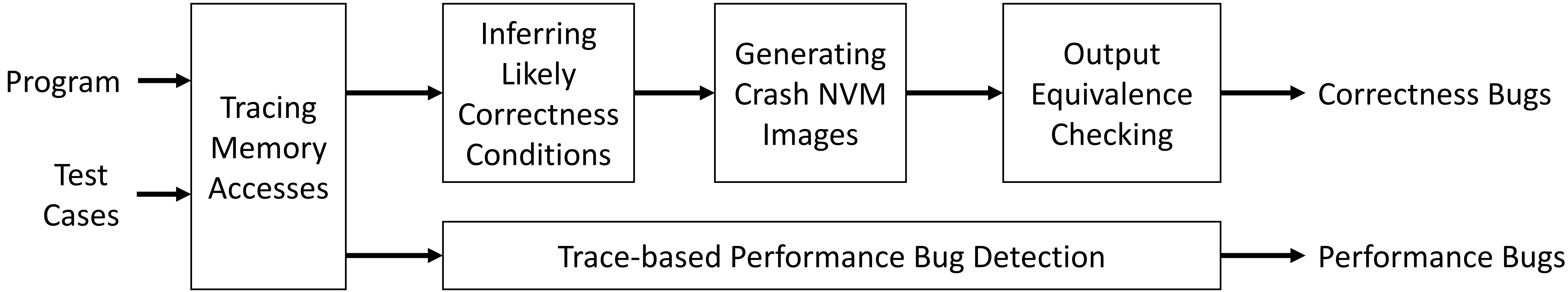
Key Idea 2: Output Equivalence Checking

- Challenge: How to automatically validate a crashed state?

Test durable linearizability (all or nothing semantics) *by comparing outputs*



Witcher: Systematic Crash Consistency Testing for NVM KV Stores



Tracing Memory Accesses

- LLVM compiler pass
- Execute the instrumented binary with a test case to collect trace

- Load
- Store with updated value
- Branch
- Call/return
- Flush
- Fence

Inferring Likely-Correctness Conditions

Correlate program dependence to NVM correctness conditions.

Y = X + 1; // both X and Y are on NVM

W(Y) is data-dependent on R(X)

Implicit correctness condition: X should be persistent before writing Y

Inferred likely-correctness condition: P(X) happens before W(Y)

Only test a crash state violating the condition: Y is persistent but X is not persistent

Inferring Likely-Correctness Conditions

- PO1: A data dependency implies a persistence ordering
- PO2: A control dependency implies a persistence ordering
- PO3: A guarded read implies a persistence ordering
- PA1: Guardian implies persistence atomicity

#	Hint		Likely-correctness Cond		NVM Image	
	Example	Rule	Example	Rule	$\mathbb{P}$	$\mathbb{U}$
PO1	Y=X+3;	W(Y) $\xrightarrow{dd}$ R(X)	X=...;Y=...;	P(X) $\xrightarrow{hb}$ W(Y)	Y	X
PO2	if(X){Y=3;}	W(Y) $\xrightarrow{cd}$ R(X)	X=...;Y=...;	P(X) $\xrightarrow{hb}$ W(Y)	Y	X
PO3	if(X){Z=Y+3;}	R(Y) $\xrightarrow{cd}$ R(X)	Y=...;X=...;	P(Y) $\xrightarrow{hb}$ W(X)	X	Y
PA1	if(X){M=N+3;}	R(N) $\xrightarrow{cd}$ R(X)	X=...;Y=...;	AP(X,Y)	X	Y
	if(Y){K=J+3;}	R(J) $\xrightarrow{cd}$ R(Y)			Y	X

- Program Analysis for Inference
  - Static analysis: register-level data and control dependency
  - Dynamic trace analysis: memory-level data dependency

Generating Crash Images

- How to guarantee each crash NVM state is valid?

Cache and NVM simulation

- Starting from the empty cache and NVM states
- Simulates the effects of store, flush and fence along the trace

- How to detect condition violations?

Check before simulating each fence instruction

Outline

1. Background and Motivation
2. Witcher
3. ***<u>Evaluation</u>***
4. Conclusion

Evaluation

Evaluation Questions:

- Can Witcher detect new bugs?
- Is Witcher scalable?

Tested Applications:

- 20 NVM programs
    - highly optimized persistent key-value indexes
    - concurrent persistent indexes converted by RECIPE [SOSP’19]
    - Intel PMDK applications
    - Persistent server applications
- Low-level persistence primitives and High-level persistence transactions
- Single-thread, lock-based, lock-free
- 2000 randomly generated key-value operations

Detected Correctness Bugs

- 47 (36 new) correctness bugs from 18 apps
- 25 persistence ordering bugs and 22 persistence atomicity bugs
- All confirmed by the developers

Diverse impact
- Lost, unexpected, duplicated key-val pairs
- Unexpected operation failure
- Inconsistent structure

Fixing strategies
- Adding required persistence primitives
- Reordering persistence primitives
- Merge multiple writes into one word-size write
- Crash-inconsistency-tolerable
- Crash-inconsistency-recoverable
- Logging/transaction

All those bug fixes are complicated and require deep understanding of the applications.

Detected bug in PMDK memory allocator


```
diff --git a/src/libpmemobj/heap.c b/src/libpmemobj/heap.c
index 4cbb52c42..a45e5742d 100644
--- a/src/libpmemobj/heap.c
+++ b/src/libpmemobj/heap.c
@@ -953,11 +953,11 @@ heap_split_block(struct palloc_heap *heap, struct bucket *b,
        uint32_t new_chunk_id = m->chunk_id + units;
        uint32_t new_size_idx = m->size_idx - units;

-       *m = memblock_huge_init(heap, m->chunk_id, m->zone_id, units);
-
        struct memory_block n = memblock_huge_init(heap,
                                                   new_chunk_id,
                                                   m->zone_id,
                                                   new_size_idx);

+       *m = memblock_huge_init(heap, m->chunk_id, m->zone_id, units);
+
```

Change the old header

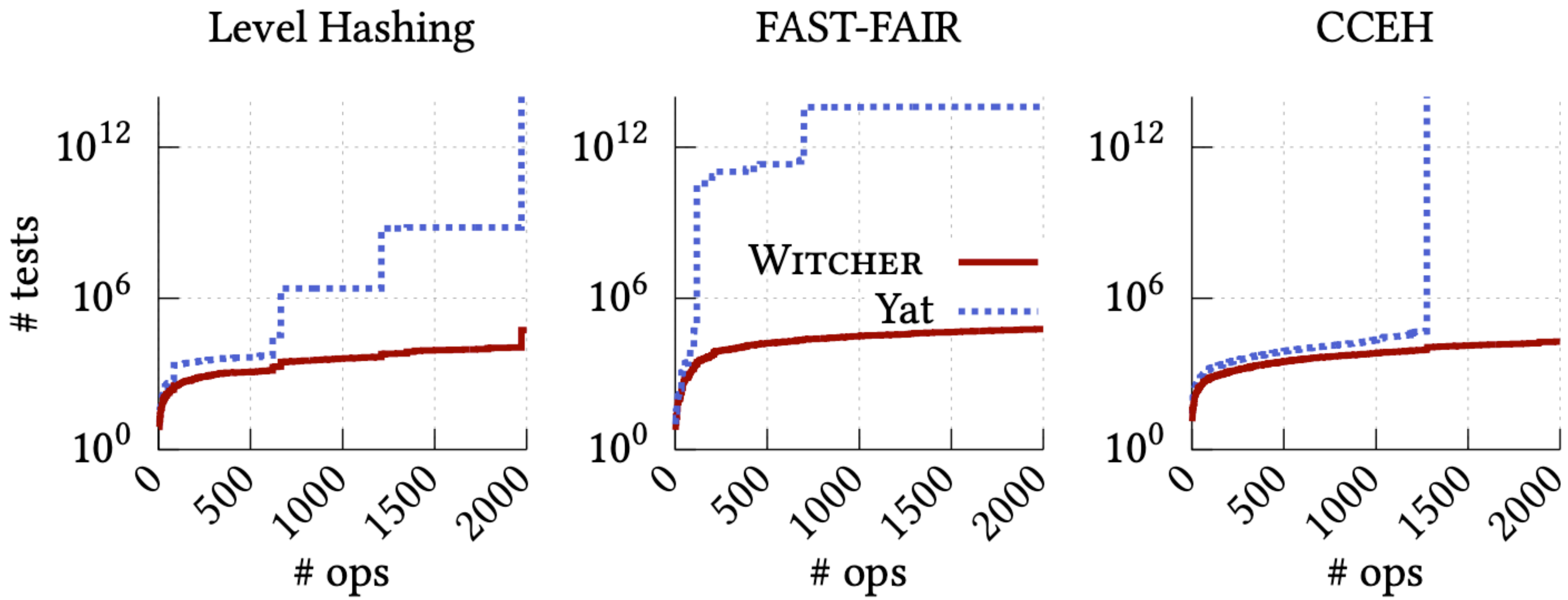
Write a new header

- Detecting this bug requires application-specific knowledge
- Witcher is able to detect this bug by using
  - Likely-correctness condition inference
  - Output equivalence checking

https://github.com/pmem/pmdk/issues/4945

Scalability

Comparison with Exhaustive Searching approach (Yat [ATC'14])



Comparison with Random Searching

- 100 Million (1 week)
- Detect one or two of the bugs

Outline

1. Background and Motivation
2. Witcher
3. Evaluation
4. ***<u>Conclusion</u>***

Conclusion

- Developing a correct NVM-backed crash consistent program is hard

- Witcher
  - Infers Likely-Correctness Conditions to prune test space
  - Performs Output Equivalence Checking to test inconsistencies automatically

- Detected 205 (149 new) correctness/performance bugs in NVM-backed key-value stores and PMDK library.

Witcher can effectively detect NVM bugs

- *without a user-provided checker*
- *without a test space explosion problem.*