Robustness Verification for Checking Crash Consistency of Non-volatile Memory

Zhilei Han*, Fei He

ASPLOS'25, March 28, 2025

Tsinghua University

So...what is Non-volatile Memory?

1 Introduction

Non-volatile Memory are persistent storage devices that allow byte-level access.

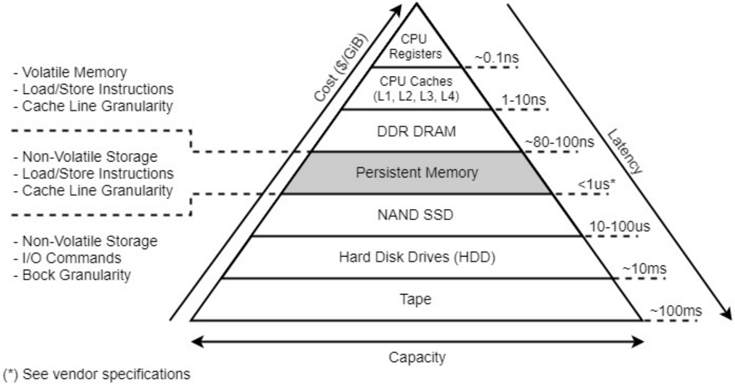


Image Source: Steve Scargall, *Programming Persistent Memory: A Comprehensive Guide for Developers*

Challenges

1 Introduction

Modern architecture employs transparent **cache**.

This leads to unintuitive and **non-standard semantics of programs running on NVMs.**

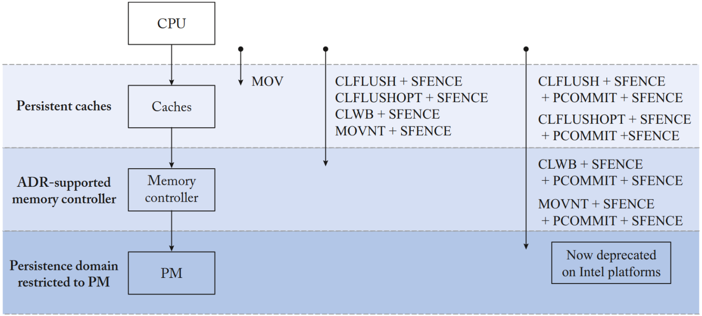
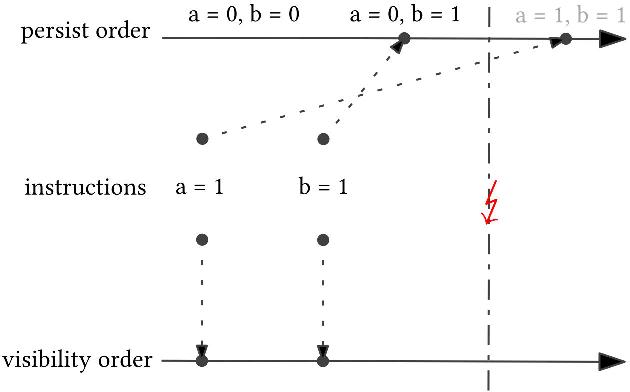


Image Source: V. Gogte, A. Kolli, and T. F. Wenisch, A Primer on Memory Persistency.

Challenges (Example)

1 Introduction

The example shows potential execution of $a = 1; b = 1;$



Challenges (Cont.)

Instructions like `clflush` and `sfence` on x86 are provided to manipulate the persistence order.

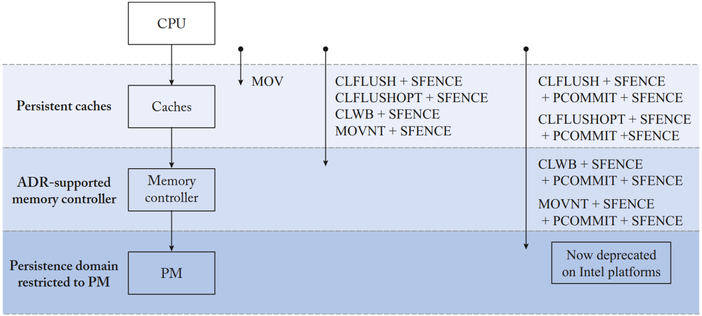


Image Source: V. Gogte, A. Kolli, and T. F. Wenisch, A Primer on Memory Persistency.

Our Goal

Persistent programming is extremely error-prone!

Our Goal

Persistent programming is extremely error-prone!

We aim to mitigate this by **developing an automatic verification method for NVM**.

Our Goal

Persistent programming is extremely error-prone!

We aim to mitigate this by **developing an automatic verification method for NVM**.

Crash Consistency: the running program could always recover from a crash correctly.

The Definition

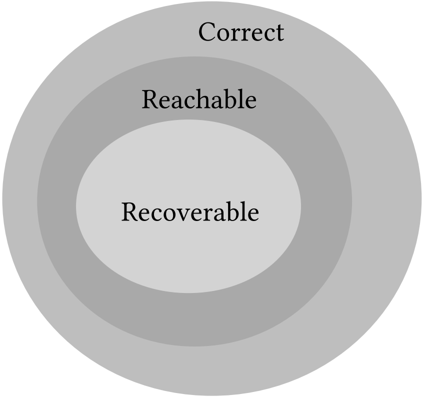
We rely on **robustness** to reduce crash consistency to memory consistency checking:

Robustness

A program running on NVM is **robust**, if any recovered memory state after system failure is guaranteed to be reachable (per the underlying memory consistency model).

Implication of Robustness

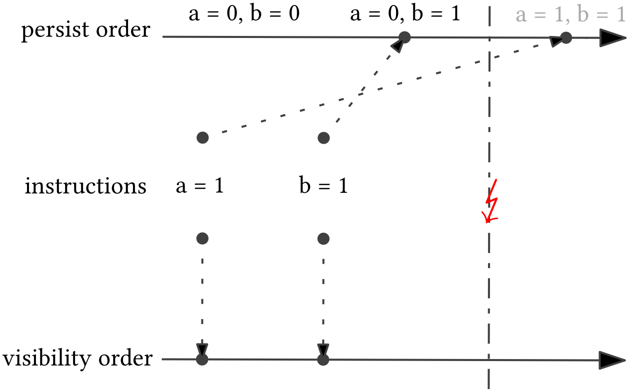
Assuming the volatile behaviour of programs is correct, robustness is a **sufficient condition** for crash consistency.



Refutation

2 Robustness Verification

Clearly, the previous example is not robust.



Proving Robustness: A Naive Algorithm

Consider enumeration of all possible program states for $a = 1; \parallel b = 1;$

Program State	Is it possible to recover from NVM?	Is it reachable?
(a = 0, b = 0)	Y	Y
(a = 0, b = 1)	Y	Y
(a = 1, b = 0)	Y	Y
(a = 1, b = 1)	Y	Y

Since all NVM states are reachable, the program is robust.

Reachability can be checked by existing approaches, but a **core problem** remains:

Core Problem

Given a program state s, how do we check s is recoverable, i.e. is it a valid NVM state?

Recovery Observer: An Example

Firstly, we use *recovery observer* to instrument the program with a virtual thread

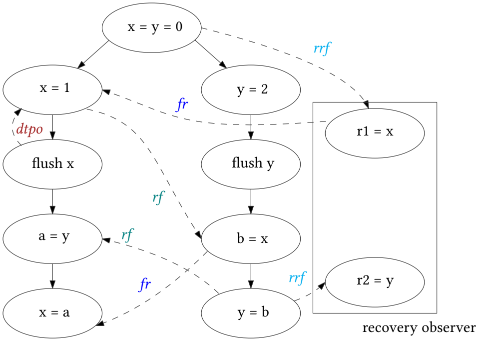
```
x = 1;          y = 2;
flush x;   ||   flush y;   ||   r1 = x;
a = y;          b = x;          r2 = y;
x = a;          y = b;
```

The recovery observer represents a recovered state from NVM.

Event Order Graph

2 Robustness Verification

- A concurrent execution could be modeled as a labeled directed graph.

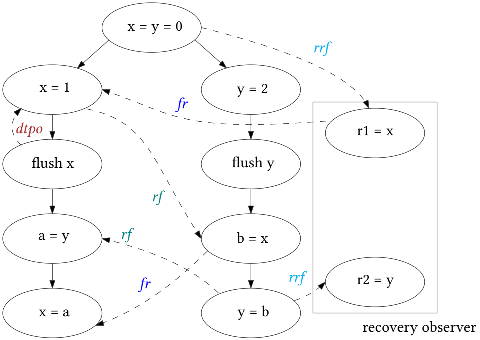


[1] F. He, Z. Sun, and H. Fan, Satisfiability modulo ordering consistency theory for multi-threaded program verification

[2] J. Alglave, D. Kroening, and M. Tautschnig, Partial Orders for Efficient Bounded Model Checking of Concurrent Software

Event Order Graph

- A concurrent execution could be modeled as a labeled directed graph.
- However, the standard constraints (`po`, `rf` etc.) are **not sufficient to model programs running on NVM**.



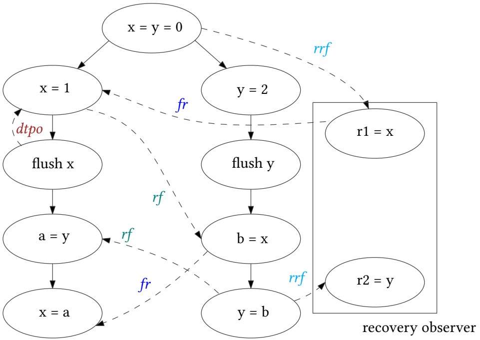
[1] F. He, Z. Sun, and H. Fan, Satisfiability modulo ordering consistency theory for multi-threaded program verification
[2] J. Alglave, D. Kroening, and M. Tautschnig, Partial Orders for Efficient Bounded Model Checking of Concurrent Software

Modeling Persistency

To model persistency, we introduce additional constraints `dtpo`:

`dtpo`

`dtpo` orders any flush on the shared variable x before any store w to x that are `co`-ordered after the store w' to x read by recovery observer.



Core Problem

Given a program state s, how do we check s is recoverable, i.e. is it a valid NVM state?

Now we just need to:

- add recovery observer to the program, representing the state s,
- construct the event order graph, and
- **check if the graph is acyclic**.

Basically, the problem is reduced to validity of a concurrent execution with additional ordering constraints.

The Exploration Algorithm

Consider the naive algorithm again

Program State	Is it possible to recover from NVM?	Is it reachable?
(a = 0, b = 0)	Y	Y
(a = 0, b = 1)	Y	Y
(a = 1, b = 0)	Y	Y
(a = 1, b = 1)	Y	Y

The brute-force search is inefficient!

Instead, we implement our algorithm in an SMT solver leveraging the DPLL(T) exploration.

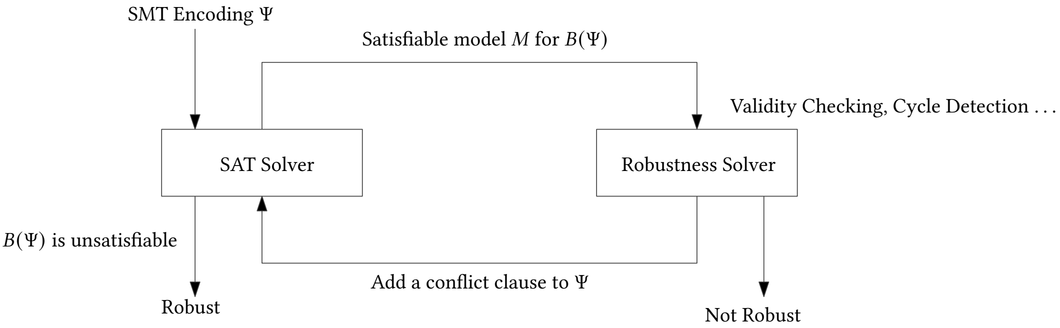
$$\rho = x_0 = 0 \wedge y_0 = 0 \qquad \text{(initial value)}$$
$$\wedge\, x_1 = 1 \wedge a = y_1 \wedge x_2 = a \qquad \text{(first thread)}$$
$$\wedge\, y_2 = 2 \wedge b = x_3 \wedge y_3 = b \qquad \text{(second thread)}$$
$$\wedge\, r_1 = x_4 \wedge r_2 = y_4 \qquad \text{(recovery observer)}$$

`x = 1;`
`flush x;`
`a = y;`
`x = a;`
	‖	`y = 2;`
		`flush y;`
		`b = x;`
		`y = b;`
			‖	`r1 = x;`
				`r2 = y;`

Implementation (Cont.)

2 Robustness Verification

A dedicated theory solver is implemented for robustness checking.



Experimentation (Overview)

Benchmark: 26 programs from PMDK `pmemobj` (YES = Robsut).

	PMVerify	PSan	PSan*
YES	1	0	0
No	12	6	0
UNKNOWN	13	20	26
Unique No.	7	0	0
Average Time	2768.42s	16.7s	5.7s
Standard Deviation	1045.26s	9.98s	2.8s

Baseline: PSan random/model checking mode

Takeaway

To summarize:

- To prove programs running on NVM is crash safe, we propose to prove **robustness** of the program.

Takeaway

To summarize:

- To prove programs running on NVM is crash safe, we propose to prove **robustness** of the program.
- To solve the core problem of checking NVM state validity, we show that it can be **reduced to a concurrent execution** with additional constraints.

Takeaway

To summarize:

- To prove programs running on NVM is crash safe, we propose to prove **robustness** of the program.
- To solve the core problem of checking NVM state validity, we show that it can be **reduced to a concurrent execution** with additional constraints.
- The algorithm is implemented in an SMT solver for **efficient exploration** of search space.

Thank you for listening!

Any questions?