PMVerify: Robustness Verification for Checking Crash Consistency of Non-volatile Memory

Abstract

The emerging non-volatile memory (NVM) technologies provide competitive performance with DRAM and ensure data persistence in the event of system failure. However, it exhibits weak behaviour on hardware architectures in terms of the order in which stores are committed to NVMs, and therefore requires developers to manually flush pending writes. To ensure correctness of this error-prone task, it is crucial to develop a rigid method to check crash consistency of programs running on NVM devices. Most existing solutions are based on testing and rely on user input to dynamically detect such deficiencies. In this paper, we present a fully automated method to statically verify robustness, a newly established property for ensuring crash consistency of such programs. The method is based on the observation that, reachability of a post-crash non-volatile state under a given pre-crash execution can be reduced to validity of the pre-crash execution with additional ordering constraints under memory consistency model. Our robustness verification algorithm employs a search-based framework to explore all partial executions and states, and checks if any non-volatile state is reachable under certain pre-crash execution. Once a reachable non-volatile state is obtained, we further check its reachability under memory consistency model. The algorithm is implemented in a prototype tool PMVerify that leverages symbolic encoding of the program and utilizes an SMT solver to efficiently explore all executions and states. A dedicated theory solver is integrated into the DPLL(T) framework to optimize the robustness checking algorithm. Experiments on the PMDK example benchmark show that PMVerify is the first tool to establish robustness, and is competitive with the state-of-the-art dynamic tool, PSan, in terms of robustness violation detection.

Keywords: persistent memory, non-volatile memory, robustness, program verification, crash consistency

1 Introduction

Non-volatile memory (a.k.a. NVM, or persistent memory) is a kind of non-conventional, byte-addressable storage device that preserves its content after a power failure [35, 36]. It enables direct access to persistent data using standard load and store instructions, and thus avoids the overhead of OS system calls. Due to its competitive performance with DRAM and guarantee of data persistence, it has been widely used in persistency-critical systems such as databases [4, 47, 49] and file systems [8, 13, 33, 42, 58–61, 65].

However, modern processors have write-back caches that induce non-determinism in the order stores are written to memory. Since cache systems are volatile, it may lead to data loss if some stores have not been committed to NVM when a crash happens. The exact order in which stores are written back to NVM, referred to as *persist order*, is constrained by the cache coherence protocol. Similar to memory consistency models which specify visibility order of memory operations, in recent works the Intel-x86 [10, 37, 51–53] and ARMv8 [10, 54] persistency models have been formalized which prescribes the persist order. Both architectures exhibit weak behaviours in terms of persist order.

As a simple example on Intel-x86, assume crash happens after executing the two instructions `a = 1; b = 1`. Upon recovery, it is possible to observe the non-volatile state `a = 0; b = 1` (we assume 0 is the initial value of `a`). In general, persist order might differ from the order memory operations are made visible. Figure 1 shows a possible execution of these two instructions and relevant orders. Here the store `a = 1` is issued and becomes visible first per program order, but remains in caches. On the contrary, the store `b = 1` is issued later but leaves the cache before the system fails. The store `a = 1` in the volatile cache is thus lost due to the crash.

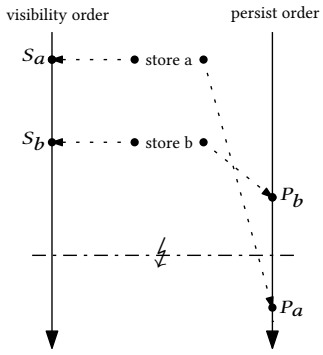

Figure 1. A possible visibility and persist order of two stores `a = 1; b = 1;` in a single thread. S_a, S_b are the points the stores are made visible to all threads, and P_a, P_b are when they are committed to NVM. $\lightning$ signifies system failure.

Overall, persistent programming is an error-prone task. It is the responsibility of the developers to avoid corruption of data residing on NVMs, since any inconsistency would persist across reboots. This necessitates a clear understanding of the persistency semantics. Although instructions (e.g. `clflush` and `clflushopt` on Intel-x86) have been introduced to constrain persist order of memory operations, the fact that stores are committed to NVMs in an out-of-order

manner can be counter-intuitive. The matter becomes even more intricate for a multi-threaded program.

To assist developers in correctly programming NVMs, researchers primarily pursue two approaches. On the one hand, high-level mechanisms such as *transactions* [6, 11, 18–20, 27, 54] and *locks* [5, 7, 26, 29, 44] have been developed to facilitate the development of NVM programs. However, these mechanisms often introduce significant overhead. On the other hand, another line of research focuses on enhancing the reliability of NVM programs [9, 12, 16, 23, 34, 43, 45, 46, 48, 55].

An important property, known as *crash consistency*, has been proposed to characterize the reliability of NVM programs. It ensures that the program state recovered from NVM after a system failure is always consistent, thereby enabling seamless resumption of program execution [55]. However, a considerable amount of existing tools require specifications provided by users for accurate bug detection. For instance, XFDetector [45] requires user annotation of commit variables to avoid false alarms, while PMTest [46] and PMDebugger [12] requires explicit annotation of ordering constraints in the program. [48] is able to prove correctness of NVM programs in terms of persistency invariant, a predicate that always holds on recovered state, but it is also restricted to the specifications provided by the user. The model checkers Jaaru [23] and Yat [43] do not require user input, but they only detect observable bugs, i.e. segmentation fault or assertion violation.

To circumvent the aforementioned difficulties in crash consistency checking, Gorjiara et al. [22] proposed a novel correctness criterion called *robustness*. Intuitively, a program is *robust* if the state recovered from NVM after system failure is guaranteed to be reachable under memory consistency model [1]. For example, consider the program in Figure 1 and the observed post-crash state `a = 0; b = 1`. This state is not reachable if we ignore the existence of NVM devices and possible crashes, i.e. we only consider its normal executions. Therefore, the program is non-robust.

An advantage of robustness is that user annotation is no longer necessary for verification. In this setting, crash consistency checking can be separated into two steps: (1) prove the program is robust, and (2) verify program correctness under memory consistency model. The latter problem is well-studied and numerous methods for checking weak memory consistency exist in the literature [3, 15, 24, 28, 39, 57, 62], which could be reused on a robust program.

Since robustness acts as a bridge that reduces crash consistency to memory consistency, in this paper, we focus on developing a method for checking robustness of NVM programs. The tool PSan developed in [22] employs a dynamic algorithm to sample execution traces from the program, and checks these traces for robustness violation. While able to find robustness violation, it relies on test input generation and sampling and thus is inherently incapable of proving robustness. In contrast with PSan, we propose a static method aimed at formally proving robustness.

Our method is based upon an observation that, the reachability of a post-crash non-volatile state under certain pre-crash execution can be reduced to validity of the execution with some additional ordering constraints. This enables us to efficiently identify reachable non-volatile states given a pre-crash execution, and check if they are also reachable under memory consistency model. The latter reachability checks are further optimized by shrinking its search space using the pre-crash execution.

Furthermore, we leverage a search method to explore all possible executions and (non-volatile) states and check their reachability. Apart from general-purpose search algorithms, some methods have been designed for efficient exploration of the vast search space in concurrent programs. These include stateless model checking algorithms with dynamic partial order reduction [1, 39–41] and SMT-based methods that encode the program and rely on constraint solving [3, 15, 17, 24, 57, 63, 64]. Our implementation opts for the latter method for exploration, which depends on a symbolic encoding of the input program, and a dedicated theory solver for robustness checking. The solver utilizes the emerging ordering consistency theory [24] for optimized validity checking used in the dual reachability checks and is incorporated into the DPLL(T) framework. Robustness violation is reported whenever we find a non-volatile state that is unreachable under memory consistency model, and we confirm robustness of the program if the exploration is exhaustive.

The proposed method has been implemented in a prototype tool called PMVerify. On 26 programs collected from the PMDK [27] `pmemobj` libraries, PMVerify is able to report 12 robustness violations and successfully proves robustness of one case. Compared to the dynamic model checking tool PSan [22], our method finds 6 more violations while PSan fails to prove robustness. Besides, on a set of 12 manually crafted robust programs, PMVerify is able to prove 6 of them, while PSan is unable to prove any case.

In summary, our main contributions are:

1. We show that the reachability checking problem of a post-crash non-volatile state under a given pre-crash execution can be reduced to the well-studied validity checking problem of a concurrent execution (Section 3).
2. We propose a novel and efficient algorithm for checking robustness of all possible executions within a non-volatile memory program (Section 4). This algorithm is encapsulated as a dedicated theory solver and incorporated into the DPLL(T) framework (Section 5).

[1] The definition of robustness in this paper is formulated differently from the original definition in [22], where it is defined using the notion of strong persistency model instead of reachability of recovered state.

3. The approach is implemented in a prototype tool, and we conduct experiments on PMDK benchmarks and a set of manually crafted robust programs. Evaluation results show our method is competitive with dynamic tool PSan on robustness violation detection and outperforms on robustness verification (Section 6).

2 Preliminaries

2.1 x86 Persistency Model

In this paper, we focus on checking robustness of non-volatile memories on Intel-x86 platforms. The visibility order of memory operations is characterized by the standard `x86-TSO` model [56], while persist order is prescribed by Px86, a persistency model formalized in [53]. In this setting, a system typically employs a three-layer memory hierarchy per Px86 operational semantics: instructions are issued to thread-local store buffers first, then propagated to a global persistent buffer (write-back caches), from where stores are committed to NVM.

Cache line write-back instructions can be used to constrain persist order. The Intel-x86 architecture provides three such instructions: (1) cache line flush instruction `clflush`, (2) cache line optimized flush instruction `clflushopt`, and (3) cache line write back instruction `clwb`. All three of these instructions write back the content of a single cache line, but differ in how they could be reordered with other instructions. `clflush` instruction has stronger constraints and can only be reordered with loads, while `clflushopt` can be reordered with store, `clflush` and `clflushopt` instructions to other cache lines. `clwb` has the same semantics as `clflushopt` but does not invalidate the cache line, providing better performance. To further constrain the order, the memory barriers `mfence` and `sfence` can be used. `mfence` can not be reordered while `sfence` allows reordering with loads.

Table 1 summarizes the order between relevant instructions based on the standard `x86-TSO` model and Px86 semantics. Note only visibility order is characterized in the table, which roughly corresponds to the order in which instructions propagate from store buffers to the persistent buffer on Px86.

We can now define the persist order and reachability of non-volatile states:

Definition 1 (Persist Order)**.** Given a fixed visibility order (defined later in Definition 5 as hb), the persist order, written nvo, is defined as a total order on all stores and flushes that satisfies the following two axioms [53]:

1. The visibility order and persist order coincide between stores to the same variable.
2. If a store is (visibility-)ordered before a flush to the same variable, then it must persist before any stores (visibility-)ordered after the flush.

At any point during program execution, only stores in a *prefix* of the events in nvo have persisted. In the case of system failure, these persisted stores in the prefix are safe and recoverable, which induce a *non-volatile state* s where for each location x, $s(x)$ equals the last store to x in the prefix. If a state s is induced by a prefix of some persist order nvo of the program that contains all flushes, s is said to be a *reachable* non-volatile state.

2.2 Program and Execution

2.2.1 Programs. We formulate a simple concurrent language for demonstration of our approach. It assumes a set of thread-local variables $\mathcal{V}_l$ (written *a, b, c* etc.) and shared variables $\mathcal{V}_p$ (written *x, y, z* etc.). All shared variables reside on non-volatile memory. For simplicity, flush operations work at the granularity of variables instead of cache lines. Let e represent an expression built from local variables, integers and arithmetic operators, an instruction i is then defined by the following grammar:

$$i ::= a = x \mid x = e \mid \texttt{fence} \mid \texttt{flush}\ x$$

Following [22] we ignore the differences between the flush instructions and assume a single `flush` operation. Likewise, we only consider a memory barrier `fence`. `flush` and `fence` exhibit semantics of `clflush` and `mfence` respectively as in Table 1. We note that our implementation supports all variants of flush operations and barriers (Section 6).

A thread consists of a sequence of instructions, and a (concurrent) program is the parallel composition of one or more threads. We use the symbol $\|$ for parallel composition, and for each thread, we designate a thread identifier $\tau \in \texttt{Tid}$. Likewise, each instruction in a thread is associated with an event identifier $i \in \mathbb{N}$, i.e. the index in the sequence of instructions. For a given program, a state s is defined as a valuation of all shared variables, i.e. $s \in \mathcal{V}_p \to \mathbb{Z}$.

2.2.2 Event Order Graph. Similar to the standard declarative methods in the literature [3, 17, 24, 40, 64], we represent execution of a concurrent program by an *event order graph* (EOG). However, executions on non-volatile memory are slightly different from those on traditional DRAM, in that system failure might happen before a program execution finishes. Therefore, we distinguish between partial and total executions, and adjust the definition of EOGs accordingly. We first define a memory event:

Definition 2 (Event)**.** An event e is a triple (τ, i, l) where $\tau \in \texttt{Tid}$ is a thread identifier, $i \in \mathbb{N}$ is an event identifier, and l is an event label that can be one of the following:

- $\texttt{R}(x, v)$, marking the event as a read event, where v is the value read from the shared variable x,
- $\texttt{W}(x, v)$, marking the event as a write event, where v is the value stored to shared variable x,

Table 1. The preserved program order of Intel-x86 instructions relevant to persistency. ✗ means two instructions can be reordered, while ✓ means they are always ordered. CL means the pair of instructions is only ordered when on the same cache line.

Earlier in Program Order \ Later in Program Order	**read**	**write**	**mfence**	**sfence**	**clflushopt**	**clflush**
read	✓	✓	✓	✓	✓	✓
write	✗	✓	✓	✓	CL	✓
mfence	✓	✓	✓	✓	✓	✓
sfence	✗	✓	✓	✓	✓	✓
clflushopt	✗	✗	✓	✓	✗	CL
clflush	✗	✓	✓	✓	CL	✓

- $\mathtt{FL}(x)$, marking the event as a flush event, where all pending stores to x are forced to persist in the order they are issued
- F, marking the event as a memory fence event, which prevents reordering of events before and after it.

Remark 1 (Notation)**.** Given an event label l, the functions type, loc, and val returns the type (R, W, FL, F), location (x), value read or written (v) of l if applicable. Given an event e, the functions Tid, # and lab return the thread identifier, event identifier and event label respectively. The functions on event labels (type, loc etc.) are also lifted to events. For a program P, we write E_P for the set of memory events in P. The method to generate E_P is straightforward [37] by simulation of program execution, and we omit the details here. We abuse the symbols R, W, FL, F for the set of events with the corresponding label in E_P.

Given a relation r, we write r^+ for the transitive closure of r, and r^{-1} for its inverse. Given a relation A, $r|_A$ is r restricted to A. We write $r_1; r_2$ as the relation composition of the two relations r_1 and r_2. For a set of events E, E_x is the subset of E restricted to events on variable x, i.e. $E_x = \{e \in E \mid \mathtt{loc}(e) = x\}$. For any ordering relation r over E, we also write $e_1 \prec_r e_2$ for $(e_1, e_2) \in r$.

An EOG is then defined with respect to ordering relations over memory events:

Definition 3 (Event Order Graph)**.** An event order graph $G = (E, E_0, \mathsf{po}, \mathtt{rf})$ consists of a set of events E and a subset of initialization events $E_0 \subseteq E$ containing a single write event to each shared variable. po, rf are relations over E where,

- $\mathsf{po} \subseteq E \times E$ is the program order, a total order of events in each thread. Moreover, initialization events in E_0 are ordered before the other events in E. po can be derived syntactically from the program, i.e. $\mathsf{po} = \{(e_1, e_2) \mid \mathtt{Tid}(e_1) = \mathtt{Tid}(e_2) \wedge \#e_1 < \#e_2\} \cup (E_0 \times E \setminus E_0)$.
- $\mathtt{rf} \subseteq (E \cap \mathtt{W}) \times (E \cap \mathtt{R})$ is the read-from relation between write and read events on the same variable. Intuitively, $(e_w, e_r) \in \mathtt{rf}$ if e_r reads the value written by e_w. It is obvious that each read event should read from at most one write event, i.e. for any events $e_w^1, e_w^2 \in (E \cap \mathtt{W})$ and $e_r \in (E \cap \mathtt{R})$, $(e_w^1, e_r) \in \mathtt{rf} \wedge (e_w^2, e_r) \in \mathtt{rf} \longrightarrow e_w^1 = e_w^2$.

For convenience, we use $G.x$ to refer to the element of G, where x can be E, E_0, po or rf. When the context is clear, we write E, po etc. directly.

An event order graph G represents a (total) *execution* of a concurrent program P, if $G.E$ equals E_P, and $G.\mathtt{rf}$ assigns a write event to each read event. In this case, the execution finishes without being interrupted by a potential crash. However, not all executions are *valid*, or *consistent*, per the underlying memory consistency model that prescribes allowed visibility order of memory operations.

Each memory consistency model M essentially defines a predicate over executions, denoted $\mathtt{cons}_M(\cdot)$, for the set of valid executions under M. The memory consistency model adopted in this paper is an extension of standard x86-TSO [56] for Intel-x86 platform which identifies a global *happens-before* order hb over all events. Unlike Sequential Consistency (SC), only *preserved program order* (ppo), a subset po that can not be reordered according to architecture specification, is included in hb.

Table 1 summarizes the ordering constraints of relevant instructions. It shows that writes and flushes might be reordered with later reads. The preserved program order ppo is then formally defined as:

$$\mathsf{ppo} \triangleq \{(e_1, e_2) \in \mathsf{po} \mid e_1 \in \mathtt{W} \cup \mathtt{FL} \longrightarrow e_2 \notin \mathtt{R}\}$$

To define happens-before order hb, we introduce coherence order (co):

Definition 4 (Coherence Order)**.** Given an execution G, a coherence order $\mathtt{co} \subseteq (G.E \cap \mathtt{W}) \times (G.E \cap \mathtt{W})$ is the disjoint union of relations $\mathtt{co}_x$ for each shared variable $x \in \mathcal{V}_p$, where $\mathtt{co}_x$ is a strict total order on write events to x.

Orders within the same thread (called *internal* orders) are distinguished from those across different threads (called *external* orders), and we denote them with suffix `i` and `e` respectively. For instance, $\mathtt{rf_i}$ is the relation $\mathtt{rf} \cap (\mathtt{po} \cup \mathtt{po}^{-1})$. Additionally, given a coherence order $\mathtt{co}$, the *from-read* relation $\mathtt{fr}$ between a write event and a read event is derived as $\mathtt{fr} \triangleq \mathtt{rf}^{-1}; \mathtt{co}$. Intuitively, if we have $(e_w, e_r) \in \mathtt{rf}$ and $(e_w, e'_w) \in \mathtt{co}$, e_r must happen before e'_w since e_r would read from e'_w otherwise.

Definition 5 (x86-TSO)**.** An execution G is valid under x86-TSO, written $\mathtt{cons}_{\mathtt{TSO}}(G)$, if there is a coherence order $\mathtt{co}$ such that

1. $\mathtt{hb} = (\mathtt{ppo} \cup \mathtt{rf_e} \cup \mathtt{co} \cup \mathtt{fr})^+$ is irreflexive,
2. $\mathtt{fr}; \mathtt{po}$ is irreflexive (per-location coherence)

Each valid execution G induces a (volatile) state s, where for each $x \in \mathcal{V}_p$, $s(x)$ equals the value written by the last write event to x in $\mathtt{hb}$. If s is induced by some execution of the program P, it is said to be a *reachable* state of P under x86-TSO.

2.3 Crash Consistency and Robustness

Crash consistency is an essential property of programs running on non-volatile memories. Given that the system may crash at any time, it specifies that program execution can be correctly resumed from the recovered non-volatile state, as defined in Section 2.1. This essentially requires that the post-crash execution starting from the state does not terminate unexpectedly (e.g. segmentation faults or assertion violation) or cause data corruption. However, most tools in the literature rely on user annotation for crash consistency bug detection, and the few automatic tools only detect observable bugs. To tackle the problem, *robustness* is proposed [22] as a sufficient condition for crash consistency of lock-free programs:

Definition 6 (Robustness)**.** A program P is *robust* iff all reachable non-volatile states of P, as defined in Section 2.1, are reachable under x86-TSO, as defined in Section 2.2.

In other words, the set of reachable non-volatile states is subsumed by the set of reachable states under x86-TSO. Crash consistency requires safe execution from any post-crash state. In this case, to prove crash consistency of a robust program, we only need to apply existing methods for ensuring correctness of a concurrent program under some weak memory consistency model, which is x86-TSO in our case. For a robust program, the problem in question is essentially reduced to the classical safety verification problem of concurrent programs. Furthermore, since consistency checking and proving robustness are decoupled from each other, this method is fully automated, and user annotation is not needed.

3 Checking Reachability of a Non-volatile State

In this paper, we focus on proving robustness. Since robustness is a universal property over non-volatile states, it is necessary to explore all non-volatile states and check if all states are reachable per definitions in Section 2.1 and Section 2.2. In this section, we focus on how to observe a potential non-volatile state from the program and check reachability of the post-crash state given a fixed pre-crash execution.

3.1 Recovery Observer

To enumerate non-volatile states efficiently, we leverage recovery observer to instrument the program. Recovery observer is originally proposed in [50] as a hypothetical notion that atomically observes the entire content of the NVM. It is then adopted for verification of software performing file I/O [38]. Unsurprisingly, the semantics of I/O operations to storage devices are analogous to memory operations on NVM. In fact, it has been utilized later for persistent invariant checking [48].

Intuitively, recovery observer is a virtual thread that reads each shared variable. As the recovery observer acts as an additional thread, the reads in it interleave with other memory operations. By going through all possible interleaving of the threads, each read also iterates through all possible writes. It facilitates enumeration of states since we could rely on the `rf` relation of these reads for a proper post-crash state. Figure 2 shows an example program with recovery observer. The third thread is the recovery observer with a read to each shared variable in the program.

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

Figure 2. An example of recovery observer. `r1 = x;` and `r2 = y` are not ordered.

To adopt recovery observer to our setting, we instrument the program with the virtual thread and introduce a dedicated reads-from order for reads in this thread. Formally, given a program P, its instrumented version is $P' = P \parallel P_r$, where P_r represents the recovery observer that contains an instruction $a_x = x$ for each $x \in \mathcal{V}_p$. Let REC be the set of events in the recovery observer P_r, i.e. $\mathtt{REC} = \{e \in E_{P'} \mid \mathtt{Tid}(e) = P_r\}$. To ensure they could observe all states of the program, these read events are not ordered with any other events, and in particular, they are not ordered with each

other. We thus have the following definition of instrumented execution:

Definition 7 (Instrumented Execution)**.** An instrumented execution G^i of P is an execution of the instrumented program $P \parallel P_r$. In particular, it satisfies $G^i.\mathsf{po} \cap (\mathsf{REC} \times G^i.E) = \emptyset$ and $G^i.\mathsf{po} \cap (G^i.E \times \mathsf{REC}) = \emptyset$.

We define the *recovery read-from* relation rrf of G^i as the projection of the read-from relation to REC, i.e. $\mathsf{rrf} \triangleq \mathsf{rf}|_{(\mathsf{W} \times \mathsf{REC})}$. The relation induces an observed non-volatile state s_o such that for each shared variable $x \in \mathcal{V}_p$, $s_o(x)$ equals the store read by REC_x.

Once a non-volatile state s_o is observed by the recovery observer, the next step is to check reachability of s_o under G^i, which is elaborated in Section 3.2. If s_o is indeed a reachable non-volatile state, we then need to check if s_o is reachable under x86-TSO. While this is a well-studied problem and not the topic of this paper, we note that recovery observer can be tweaked for this purpose as well. Briefly speaking, we retain the instrumented execution G^i, but group the read events in REC together as an *atomic block*, i.e. we only allow it to read the whole memory simultaneously. In this way, the recovery observer now signifies an equivalent volatile state instead. The details are given in Section 4.

3.2 Reduction to Validity Checking

In this section, we check reachability of the observed non-volatile state s_o under a given execution. Reachability under Px86, which is a more general problem than ours, has been previously proved to be decidable [2], but no algorithmic method is given. To solve the problem, the key is to reduce it to an equivalent validity checking problem of a pre-crash execution that is augmented with additional ordering constraints.

To accomplish this, we leverage the *derived TSO propagation order* (dtpo) from [37] as a bridge between memory consistency and persistency. Given an instrumented execution G^i of the program, we have:

$$\mathsf{dtpo} \triangleq \bigcup_{x \in \mathcal{V}_p} \mathsf{FL}_x \times \{w \in \mathsf{W}_x \mid \exists w' \in dom(\mathsf{rrf}), (w', w) \in \mathsf{co}\}$$

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 rrf. Note that although it is derived from the persistency-related relation rrf, it characterizes visibility order between flushes and certain stores. The correctness of this derived order can be seen by the following argument: if the flush event is instead ordered after w, then all pending writes to x, including w, should be committed to the NVM. Since w happens after w', it would overwrite w', which contradicts the fact that w' is the last write to x that has persisted. Fig. 3 demonstrates a possible instrumented execution of the program in Fig. 2 where the state $x = 0; y = 1;$ is observed. Since the observer reads the initial value of x, which is ordered before the two stores $x = 1$ and $x = a$, two dtpo ordering constraints are induced.

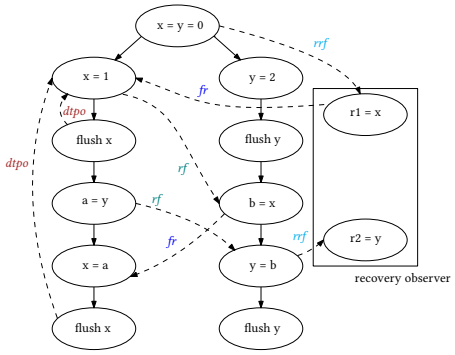

Figure 3. Visualization of an event order graph representing a possible instrumented execution of the program in Fig. 2, where the state $x = 0; y = 1;$ is observed by the recovery observer. The execution is not valid under DPTSO per Definition 8 due to cycles induced by the dtpo order.

Our goal is to check if the observed non-volatile state s_o, as induced by rrf, is reachable under G^i. Note the execution G^i only characterizes visibility order, and we do not know the exact persist order nvo. A naive approach to check reachability is to enumerate all possible persist order, and check whether the last persisted store to each shared variable conforms with s_o. Nevertheless, it is redundant to consider all possible orders. Even if we prune the search by leveraging the two axioms for nvo (Definition 1), the exhaustive method is still inefficient.

What matters in reachability checking is the last persisted stores and the additional ordering constraints they generate (dtpo). Therefore, the crux is to check if G^i is a valid execution under a memory model augmented with dtpo, called DPTSO in [37]:

Definition 8 (DPTSO)**.** An instrumented execution G^i is valid under DPTSO, written $\mathsf{cons}_{\mathsf{DPTSO}}(G^i)$, if there is a coherence order co such that

1. $\mathsf{hb} = (\mathsf{ppo} \cup \mathsf{rf}_e \cup \mathsf{co} \cup \mathsf{fr} \cup \mathsf{dtpo})^+$ is irreflexive,
2. $\mathsf{fr}; \mathsf{po}$ is irreflexive (per-location coherence)

The predicate $\mathsf{cons}_{\mathsf{DPTSO}}(\cdot)$ can be checked in an analogous way to checking $\mathsf{cons}_{\mathsf{TSO}}(\cdot)$. It is basically a cycle detection algorithm on a directed graph where the orders rf, co, dtpo etc. are regarded as edges. Since DPTSO only adds dtpo as a component in the happens-before relation, existing methods for checking validity of a concurrent execution could be easily adopted. For example, the execution depicted in Fig. 3 is not valid under DPTSO due to cycles introduced by the extra dtpo edges.

Theorem 1. Given an instrumented execution G^i valid under x86-TSO, the non-volatile state s_o induced by rrf is reachable under G^i iff $\mathsf{cons}_{\mathsf{DPTSO}}(G^i)$ holds.

Proof. The proof is elaborated in Appendix A. □

4 Robustness Checking Algorithm

In this section, we discuss the general search-based framework for robustness checking. More specifically, we extend the algorithm to partial executions and introduce the overall exploration algorithm based on it.

Previously, reachability of states is defined with an assumption that all memory events have been propagated from the store buffer and made visible to all threads before the program terminates. This is in line with the Px86 and DPTSO models. However, to verify programs running on NVM, it is necessary to reason about crashes, and in particular when a system failure would occur. Therefore, the verification algorithm must take partial executions into account.

Partial Executions. In Section 2.2, we defined an execution of a program P as an EOG that contains all memory events of a program, E_P, and assigns a value to each read event in rf. Likewise, a *partial execution* is an EOG that assigns a value to each read in rf, but it contains only a subset of all memory events E_P. However, the events must be *prefix-closed*. In other words, any event in the porf−prefix of an event in the partial execution should also be contained in the event set. This requirement corresponds to the fact that when a system fails, only a prefix of the total execution has been propagated. A partial instrumented execution is defined similarly.

Take the total execution of the program in Fig. 2 as an example, as depicted in Fig. 3. It is shown previously that this execution is not valid under DPTSO, thus the state s_o is not reachable. Now consider the realistic scenario where the instructions `flush x; a = y; x = a; flush x;` are not propagated before the crash. In this case, we obtain a partial execution as visualized below:

It can be easily checked that this partial execution is valid under DPTSO, and thus the state $x = 0; y = 1;$ is a reachable non-volatile state.

Exploration Algorithm. The reachability checking algorithm in Section 3 can be naturally lifted to partial executions since dtpo is defined analogously on partial executions. An exploration algorithm can then search through all partial executions and states, and utilize the aforementioned reduction to check reachability of a state with a given partial execution. Whenever a reachable non-volatile state is found, robustness is checked locally first by checking reachability of this state under x86-TSO. Robustness of the whole program is proved if no violation is found when the exploration ends.

The presence of recovery observer has embedded a non-volatile state in an instrumented execution, thus the exploration method only needs to search through all instrumented partial executions. Besides, it allows that some events in REC are not contained in the partial execution, i.e. it induces a

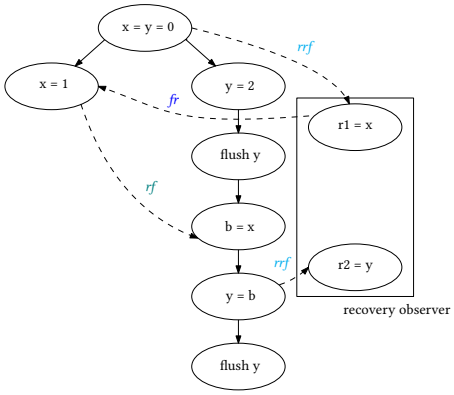

Figure 4. A partial instrumented execution of the program in Fig. 2, where the state $x = 0; y = 1;$ is observed by the recovery observer. No dtpo orders are derived since flush events to x are not in this partial execution. This execution is valid under DPTSO.

partial state. This boosts performance and allows our robustness checking algorithm to have the flexibility of leveraging different search methods, from brute force searching to more advanced stateless model checking with dynamic partial order reduction, or simply relying on program encoding and constraint solving. Our robustness checking algorithm could be incorporated into any exploration method capable of search tree pruning. Therefore, we abstract away the details and assume the exploration method provides the `next` and `hasNext` interface for exploration, and `block` interface to block a subset of partial instrumented executions. Section 5 will elaborate on this topic.

Remark 2 (Notation)**.** We say the partial instrumented execution G' is an *expansion* of partial instrumented execution G, written $G \prec G'$, if $G.E \subset G'.E$ and $G.\mathtt{rf} \subset G'.\mathtt{rf}$.

G' is an *alternation* of G, written $G' \backsimeq G$, if $G.E = G'.E$ and $G.\mathtt{rrf} = G'.\mathtt{rrf}$ (other orders in $G.\mathtt{rf}$ and $G'.\mathtt{rf}$ might differ).

The overall algorithm framework is shown in Algorithm 1. The input program is first instrumented with recovery observer, then the exploration method takes over the search. Each time a partial execution G is yielded, a partial state is also generated. We first check if it is reachable under the current execution, i.e. if $\mathsf{cons}_{\mathtt{DPTSO}}(G)$ holds. If not, we make sure not to extend G and further explore its expansion, since an invalid execution with additional ordering constraints is still invalid under DPTSO (Line 10). This optimization can be implemented in most search methods. In depth-first searching, for instance, the search immediately backtracks to avoid further redundant exploration.

If G exhibits a reachable non-volatile state s_o, the next step is to check if s_o is reachable under the x86-TSO model,

Algorithm 1: Robustness Checking Algorithm Framework for Non-volatile Memories

input : A program P running on non-volatile memory.
output : If P is robust.

```
1  P' ← P || P_r
2  while hasNext(P') do
3      G ← next(P')
4      if cons_DPTSO(G) holds then
5          foreach G' ⋍ G do
6              if cons_TSO(atomic(G')) holds then
7                  goto 2;
8          return false;
9      else
10         block({G' | G ≺ G'});
11 return true
```

which typically involves another search over all total executions (Line 5). There are abundant algorithms for this task in the literature. In our algorithm, we take advantage of the recovery observer by regarding the read events in them as an atomic block and keeping the rrf orders. The formal definition of atomic is given in Appendix B due to space limit, but intuitively this allows the recovery observer to signify an equivalent volatile state. As an example, Fig. 5 shows how to check validity of an execution under x86-TSO with the help of recovery observer.

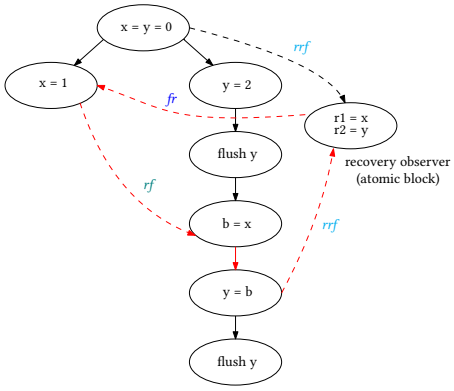

Figure 5. Visualization of validity checking of the execution in Fig. 4, where the recovery observer is regarded as an atomic block. It is invalid under x86-TSO due to the cycle in red.

While searching through total executions, we only alter other rf orders, keeping $G.E$ the same. Essentially, enumerating alternations of G is enough for checking reachability under x86-TSO, The soundness of this method is shown in the following lemma:

Lemma 1. If for every alternation G' of G, $\mathsf{cons}_{\mathsf{TSO}}(G')$ does not hold, then the observed state s_o must be unreachable under x86-TSO.

Proof. The proof is given in Appendix A. □

If the state s_o is proved unreachable under x86-TSO in this way, we report violation of robustness (Line 8). Note that s_o might be a partial state, but it is obvious that any total state that conforms with s_o is still invalid under x86-TSO. Therefore, we could always add rrf orders that read from the last store per co. This way validity under DPTSO is not affected. On the other hand, if s_o is reachable, once its reachability is proven we proceed with the exploration of executions (Line 7). If all executions have been explored and no violation is found, P has been proved robust (Line 11).

5 Integration with DPLL(T)

Algorithm 1 is parameterized by an exploration method. In this section, we instantiate our algorithm to leverage program encoding and constraint solving for this task.

5.1 Encoding

A complete encoding of a concurrent program should cover both functional program behaviours and possible interleaving of the threads, i.e. the orders between memory events. Following the standard encoding [3], the input program is firstly transformed into SSA form and the program event set E_P. The instructions in each thread, including the recovery observer in our case, are then naturally translated to atoms in first-order logic[2]. As a simple example, the program in Fig. 2 is encoded as:

$$\begin{aligned}
\rho_{ssa} = {} & x_0 = 0 \wedge y_0 = 0 && \text{(initial value)}\\
& \wedge x_1 = 1 \wedge a = y_1 \wedge x_2 = a && \text{(first thread)}\\
& \wedge y_2 = 2 \wedge b = x_3 \wedge y_3 = b && \text{(second thread)}\\
& \wedge r_1 = x_4 \wedge r_2 = y_4 && \text{(recovery observer)}
\end{aligned}$$

Note that it does not encode ordering relations. The read y_1, for instance, could potentially read from y_0, y_2 and y_3. To add ordering constraints to the encoding, we first model a partial execution by a predicate enabled (implemented as a Boolean variable) defined for every event, where enabled(e) signifies e is included in the partial execution. Furthermore, each order relations used in x86-TSO or DPTSO are represented by Boolean variables explicitly. For instance, since the coherence order co is total, it is encoded by adding a Boolean variable $\mathtt{ws}^x_{i,j}$ for each pair of stores x_i and x_j to the shared variable x. We have $(x_i, x_j) \in \mathtt{co}$ iff $\mathtt{ws}^x_{i,j}$ is assigned true. Additional *axioms* are included in the encoding that constrain its assignment:

[2]flush and fence operations are not included in the functional encoding of a program since they are irrelevant. However, they are still numbered and contribute to ordering constraints.

$$\rho^{\mathtt{co}}_{i,j} = \mathtt{ws}^x_{i,j} \longrightarrow \mathtt{enabled}(x_i) \wedge \mathtt{enabled}(x_j) \quad \text{(ws-cond)}$$
$$\wedge\, \mathtt{ws}^x_{i,j} \longrightarrow x_i \prec_{\mathtt{co}} x_j \quad \text{(ws-order)}$$
$$\wedge\, (\mathtt{enabled}(x_i) \wedge \mathtt{enabled}(x_j)) \longrightarrow \mathtt{ws}^x_{i,j} \vee \mathtt{ws}^x_{j,i} \quad \text{(ws-some)}$$

Similarly, we introduce for each variable x a Boolean variables $\mathtt{rf}^x_{j,i}$ for any read x_i and store x_j and axioms for rf as follows:

$$\rho^{\mathtt{rf}}_{j,i} = \mathtt{rf}^x_{j,i} \longrightarrow \mathtt{enabled}(x_i) \wedge \mathtt{enabled}(x_j) \wedge x_j = x_i \quad \text{(rf-val)}$$
$$\wedge\, \mathtt{rf}^x_{j,i} \longrightarrow x_j \prec_{\mathtt{rf}} x_i \quad \text{(rf-ord)}$$
$$\wedge\, \mathtt{enabled}(x_i) \longrightarrow \bigvee_{x_j \in \mathsf{W}_x} \mathtt{rf}^x_{j,i} \quad \text{(rf-some)}$$

Since fr can be derived from rf and co as discussed in Section 2, for each variable x, we introduce the following axiom for any two stores x_j, x_k and read x_i:

$$\rho^{\mathtt{fr}}_{j,i,k} = \mathtt{rf}^x_{j,i} \wedge \mathtt{ws}^x_{j,k} \longrightarrow r_{x_i} \prec_{\mathtt{fr}} w_{x_k}$$

For dtpo, we need the flush event to be enabled. Thus, for each variable x, we introduce the following axiom for any two stores x_j, x_k, read event x_i in the recovery observer, and flush $\mathtt{FL}^x_q$:

$$\rho^{\mathtt{dtpo}}_{q,j,i,k} = \mathtt{enabled}(\mathtt{FL}^x_q) \wedge \mathtt{rf}^x_{j,i} \wedge \mathtt{ws}^x_{j,k} \longrightarrow \mathtt{FL}^x_q \prec_{\mathtt{dtpo}} w_{x_k}$$

To ensure the prefix-closed property of a partial execution, encoding for each ordering relation requires the pair of events to be both enabled (e.g. the rule ws-cond and rf-val above), and an extra axiom is added to the encoding: for any two events e_1 and e_2 that are ordered by ppo, we have $\mathtt{enabled}(e_2) \longrightarrow \mathtt{enabled}(e_1)$. The encoding of the program Ψ is then the conjunction of ρ_{ssa} and all axioms ($\rho^{\mathtt{co}}_{i,j}$, $\rho^{\mathtt{rf}}_{j,i}$ etc.) related to ordering constraints.

5.2 DPLL(T) and Exploration

The encoded formula of the program Ψ is solved by an SMT solver. While it searches for a model of the formula, variables in it are assigned values. In particular, the assignment of Boolean variables representing various order relations corresponds to a partial execution and state. Modern SMT solvers typically utilize the DPLL(T) framework. In the framework, formulas are in a combination of certain first-order background theories. Each background theory $\mathcal{T}$ has a *theory solver* which decides $\mathcal{T}$-satisfiability of a conjunction of literals in $\mathcal{T}$. An overview of this framework is shown in Fig. 6.

In this framework, each atom in the given formula Ψ is first replaced by a Boolean variable, and the satisfiability of the resulting propositional formula $B(\Psi)$ is checked by an SAT solver. If $B(\Psi)$ is unsatisfiable, so is Ψ. Otherwise, since $B(\Psi)$ is an over-approximation of Ψ, theory solvers are called to check if the model M returned by the SAT solver is compatible with the underlying background theories. The theory solver also returns a *conflict clause* to prevent the SAT solver from exploring the same assignment.

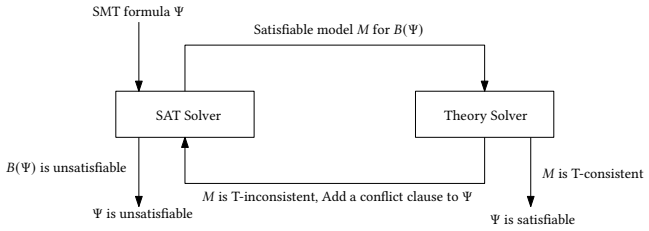

Figure 6. Overview of the DPLL(T) framework.

Following [24], while DPLL(T) controls the exploration, in our implementation each ordering constraints in the formula, such as $w_{x_i} \prec_{\mathtt{ws}} w_{x_j}$ from ws-order above, are passed to a dedicated theory solver for robustness checking. Based on the solver for ordering consistency theory, the backend employs an incremental cycle detection algorithm for efficient checking validity under DPTSO. If the current partial state is reachable, we use the solver in [15] to check its reachability under x86-TSO. Otherwise, a conflict clause is generated and returned to the DPLL(T) framework which blocks further exploration of this partial instrumented execution.

6 Implementation and Evaluation

We have implemented our method in a prototype tool called PMVerify, expanding on Deagle [25], a concurrent program verification tool that supports weak memory consistency on top of the bounded model checker CBMC. We extend Deagle's frontend to recognize NVM programs using a selected set of APIs in pmemobj library from PMDK [27]. A dedicated backend for robustness checking is implemented to complement the default solver of Deagle for weak memory consistency checking.

To evaluate the efficacy of PMVerify, we collect the example programs that accompany the pmemobj library in PMDK as the benchmark. It contains 26 small to medium-sized programs (548 LOC on average) that implement simple algorithms and basic data structures on non-volatile memory, such as binary search and persistent lists.

As a comparison, we also run PSan [22], the only robustness violation detection tool in the literature, on the same benchmark. PSan is implemented based on the dynamic model checking tool Jaaru [23] which observes the outcome of memory operations at runtime and checks for persistency bug on the observed trace. PSan offers a model checking mode that exhaustively enumerates program traces, as well as a random mode that relies on sampled traces.

Table 2. Evaluation of PMVerify and PSan on the PMDK `pmemobj` benchmarks (26 programs in total). The rows YES/No/UNKNOWN contain the total number of cases each tool proves to be robust, non-robust, or fails to give an answer. Unique No. is the number of cases only solved by this tool.

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

The experiments are conducted on an Intel® Core™ i5-10400 @ 2.90GHz CPU with 16GB memory. Timeout is set to two hours. For PSan running in random mode, the maximum amount of sampled traces is set to 100,000.

6.1 Experimental Results on PMDK benchmark

Table 2 shows the overall results of the experiment. Out of 26 programs, PMVerify is able to solve 13. It successfully proves robustness of one of the programs (`manpage.c`), and detects robustness violation of 12 test programs. The results are manually checked to ensure soundness. PMVerify fails on the remaining programs mainly due to usage of some PMDK primitives that are not modeled by the frontend, such as `pmemobj_tx_add_range_direct`, and timeouts on one program it supports.

Compared to PSan, our tool is able to find robustness violation of six more programs. Besides, since PSan employs an incomplete dynamic method, it is only able to refute robustness instead of verifying the property. On the contrary, PMVerify is able to prove robustness of a program after all executions have been explored.

In terms of performance, Fig. 7 shows the accumulated solving time of PMVerify. It takes around 45 minutes to complete verification on average. PSan, on the other hand, takes no more than a minute for the six cases it solves, which is, on average, 33 times shorter than PMVerify. We note that PSan is similar to a testing tool and could very efficiently find potential bugs in the program due to its dynamic nature. However, it cannot verify robustness. PMVerify has adopted some optimization methods to improve the performance of exploration, but still faces the common state explosion problem. The exhaustive exploration is necessary to ensure completeness, at the cost of performance. Therefore, our method and PSan can complement each other in robustness verification and bug detection.

Table 3. Evaluation results of PMVerify on 12 instrumented programs. PSan is unable to prove robustness of any program.

Program	LOC	PMVerify Time (s)
btree	493	5468.95
buffons_needle_problem	432	timeout
lists	551	timeout
pi	570	3565.19
examine_arttree	6379	7021.70
arttree	1793	timeout
fifo	207	4677.78
data_store	5512	timeout
mapcli	742	timeout
main	195	1765.79
reader	95	timeout
writer	67	5905.49
Average	1420	4734.15

6.2 Evaluation on robust programs

The robust case PMVerify solves, `manpage.c`, is a simple program that opens a persistency memory pool and does nothing. In this section, to further demonstrate the ability of PMVerify to prove robustness, we manage to instrument each of the 12 non-robust programs to manually produce a set of robust programs.

More specifically, we insert a cache line flush instruction after each memory operation. In this way, the instrumented program is guaranteed to be robust. We then run both PMVerify and PSan on this new benchmark.

Table 3 shows the results of PMVerify and PSan running on the set of instrumented programs. PMVerify is able to prove robustness of six programs with an average running time of 4734.15 seconds, including the medium-sized programs `examine_arttree` and `data_store`. This shows the ability to scale to larger robust programs. On the other hand, PSan is unable to prove robustness of any program.

Due to the increase in program size, the performance of PMVerify on this benchmark degrades by around half. We note that although adding a flush operation after every memory operation introduces considerable redundancy and increases the overall exploration space, the set of reachable non-volatile states is smaller because of stronger constraints. Since PMVerify checks for reachability under `DPTSO` model first, we can avoid later steps of checking `x86-TSO` consistency for some states. Therefore, the running time of PMVerify does not grow exponentially. In fact, all cases could be finished within three hours if we do not consider time limits,

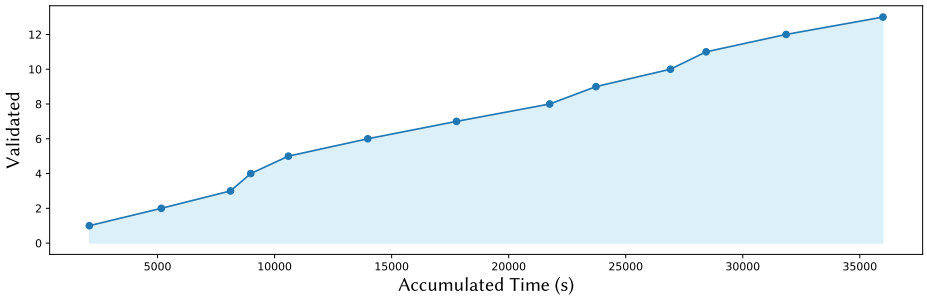

Figure 7. Accumulated solving time of PMVerify on the 13 solved case. 35989.46 seconds are spent in total.

with an average running time of 6712 seconds. This shows the efficacy of our tool PMVerify for verification of robustness.

7 Related Works

Persistent Models. The early studies on NVMs rely on certain persistency models, an extension of memory consistency models, to prescribe constraints on the persistence order. In [50], Pelly et al. classified these models into three categories: *strict persistency, epoch persistency* and *strand persistency*. The original definition of robustness in [22] is based on strict persistency, the strongest model where any recovered state is guaranteed to be an observable volatile state [14, 31].

Epoch persistency under sequential consistency is described in [50], while [32] proposes a persist barrier implementation that works on x86-TSO [56]. The first formal definition of epoch persistency is given by [30] under release consistency, and [52] formally describes operational and declarative semantics of epoch persistency under x86-TSO. StrandWeaver [21] implements strand persistency in hardware to minimally constrain persists to NVMs.

Recently, a line of work focused on formally defining the persistency model of hardware architecture. [54] develops PARMv8 model for ARMv8, followed by Px86 [53] for Intel-x86. Later, the PEx86 model [51] is proposed with formalized semantics of non-temporal stores. Alternative models such as DPTSO [37] and view-based models for Intel-x86 and Armv8 [10] are proposed to further develop these formalisms.

Memory consistency checking. The essential idea of multi-threaded program verification is to explore the possible executions caused by thread interleaving. [3] gives a framework for using partial order relations to model possible executions and encode program behaviors into a formula. Several works expand on this idea and rely on bounded model checking, such as lazy sequentialization [28] and a line of work that employs the scheduling constraints-based abstraction refinement method (SCAR) [62–64]. [17] proposes to solve the difference logic-based ordering constraints more efficiently with DPLL(T) framework. [24] proposes an ordering consistency theory and integrates a dedicated theory solver into the DPLL(T) routine, which is extended to weak memory consistency in [15]. On the other hand, stateless model checking (SMC) methods enumerate all interleavings with respect to an equivalence class, i.e. a Mazurkiewicz trace. Several algorithms have been proposed to further weaken the ordering requirement and efficiently explore the search space [1, 39, 40].

Persistency Bug Detection. Several tools have been developed to assist persistent programming, including testing applications such as XFDetector [45], PMTest [46] and PMDebugger [12]. Yat [43] is a model checker that exhaustively explores all persistence orders and crash points. The model checker Jaaru [23] reduces search space by focusing on the last writes to each location. To our knowledge, the only automated verification tool for persistent memories is introduced in [48], which utilizes an SMT-based method to formally verify persistent invariants.

8 Conclusion

In this paper, we propose a novel approach to check robustness, a sufficient condition for crash consistency of lock-free programs running on non-volatile memories. Our algorithm employs a search method to explore all partial executions and non-volatile states, and check reachability of the state under the pre-crash execution. This is achieved by reducing the reachability checking problem to checking validity of an instrumented execution under an alternative model DPTSO. Our implementation is based on encoding the program into a SMT formula and constraint solving. It succeeds in robustness verification of a set of example programs in PMDK while the dynamic robustness violation tool PSan fails. In terms of robustness bug detection, our tool also outperforms PSan.

References

[1] Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. Stateless model

checking for TSO and PSO. *Acta Informatica*, 54(8):789–818, December 2017.

[2] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Ahmed Bouajjani, K. Narayan Kumar, and Prakash Saivasan. Deciding reachability under persistent x86-TSO. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–32, January 2021.

[3] Jade Alglave, Daniel Kroening, and Michael Tautschnig. Partial Orders for Efficient Bounded Model Checking of Concurrent Software. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, Lecture Notes in Computer Science, pages 141–157, Berlin, Heidelberg, 2013. Springer.

[4] Joy Arulraj and Andrew Pavlo. How to Build a Non-Volatile Memory Database Management System. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 1753–1758, New York, NY, USA, May 2017. Association for Computing Machinery.

[5] Hans-J. Boehm and Dhruva R. Chakrabarti. Persistence programming models for non-volatile memory. *ACM SIGPLAN Notices*, 51(11):55–67, June 2016.

[6] Daniel Castro, Paolo Romano, and João Barreto. Hardware Transactional Memory meets memory persistency. *Journal of Parallel and Distributed Computing*, 130:63–79, August 2019.

[7] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. *ACM SIGPLAN Notices*, 49(10):433–452, October 2014.

[8] Youmin Chen, Jiwu Shu, Jiaxin Ou, and Youyou Lu. HiNFS: A Persistent Memory File System with Both Buffering and Direct-Access. *ACM Transactions on Storage*, 14(1):4:1–4:30, April 2018.

[9] Zhangyu Chen, Yu Hua, Yongle Zhang, and Luochangqi Ding. Efficiently detecting concurrency bugs in persistent memory programs. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, pages 873–887, New York, NY, USA, February 2022. Association for Computing Machinery.

[10] Kyeongmin Cho, Sung-Hwan Lee, Azalea Raad, and Jeehoon Kang. Revamping hardware persistency models: View-based and axiomatic persistency models for Intel-x86 and Armv8. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, pages 16–31, New York, NY, USA, June 2021. Association for Computing Machinery.

[11] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. *ACM SIGARCH Computer Architecture News*, 39(1):105–118, March 2011.

[12] Bang Di, Jiawen Liu, Hao Chen, and Dong Li. Fast, flexible, and comprehensive bug detection for persistent memory programs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, pages 503–516, New York, NY, USA, April 2021. Association for Computing Machinery.

[13] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 1–15, New York, NY, USA, April 2014. Association for Computing Machinery.

[14] Per Ekemark, Yuan Yao, Alberto Ros, Konstantinos Sagonas, and Stefanos Kaxiras. TSOPER: Efficient Coherence-Based Strict Persistency. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 125–138, February 2021.

[15] Hongyu Fan, Zhihang Sun, and Fei He. Satisfiability Modulo Ordering Consistency Theory for SC, TSO, and PSO Memory Models. *ACM Transactions on Programming Languages and Systems*, 45(1):6:1–6:37, March 2023.

[16] Xinwei Fu, Wook-Hee Kim, Ajay Paddayuru Shreepathi, Mohannad Ismail, Sunny Wadkar, Dongyoon Lee, and Changwoo Min. Witcher: Systematic Crash Consistency Testing for Non-Volatile Memory Key-Value Stores. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, pages 100–115, New York, NY, USA, October 2021. Association for Computing Machinery.

[17] Cunjing Ge, Feifei Ma, Jeff Huang, and Jian Zhang. SMT Solving for the Theory of Ordering Constraints. In Xipeng Shen, Frank Mueller, and James Tuck, editors, *Languages and Compilers for Parallel Computing*, pages 287–302, Cham, 2016. Springer International Publishing.

[18] Kaan Genç, Michael D. Bond, and Guoqing Harry Xu. Crafty: Efficient, HTM-compatible persistent transactions. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, pages 59–74, New York, NY, USA, June 2020. Association for Computing Machinery.

[19] Jerrin Shaji George, Mohit Verma, Rajesh Venkatasubramanian, and Pratap Subrahmanyam. Go-pmem: Native support for programming persistent memory in go. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 859–872. USENIX Association, July 2020.

[20] Ellis Giles, Kshitij Doshi, and Peter Varman. Continuous checkpointing of HTM transactions in NVM. *ACM SIGPLAN Notices*, 52(9):70–81, June 2017.

[21] Vaibhav Gogte, William Wang, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. Relaxed Persist Ordering Using Strand Persistency. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 652–665, May 2020.

[22] Hamed Gorjiara, Weiyu Luo, Alex Lee, Guoqing Harry Xu, and Brian Demsky. Checking robustness to weak persistency models. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, pages 490–505, New York, NY, USA, June 2022.

[23] Hamed Gorjiara, Guoqing Harry Xu, and Brian Demsky. Jaaru: Efficiently model checking persistent memory programs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, pages 415–428, New York, NY, USA, April 2021. Association for Computing Machinery.

[24] Fei He, Zhihang Sun, and Hongyu Fan. Satisfiability modulo ordering consistency theory for multi-threaded program verification. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, pages 1264–1279, New York, NY, USA, June 2021. Association for Computing Machinery.

[25] Fei He, Zhihang Sun, and Hongyu Fan. Deagle: An SMT-based Verifier for Multi-threaded Programs (Competition Contribution). In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 424–428, Cham, 2022. Springer International Publishing.

[26] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. NVthreads: Practical Persistence for Multi-threaded Applications. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 468–482, New York, NY, USA, April 2017. Association for Computing Machinery.

[27] Intel Corporation. Persistent Memory Development Kit. https://pmem.io/pmdk/, 2023.

[28] Omar Inverso, Ermenegildo Tomasco, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. Bounded Model Checking of Multi-threaded C Programs via Lazy Sequentialization. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification*, Lecture Notes in Computer Science, pages 585–602, Cham, 2014. Springer International Publishing.

[29] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-Atomic Persistent Memory Updates via JUSTDO Logging. *ACM SIGARCH*

Computer Architecture News, 44(2):427–442, March 2016.

[30] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model. In Cyril Gavoille and David Ilcinkas, editors, *Distributed Computing*, pages 313–327, Berlin, Heidelberg, 2016. Springer.

[31] Jungi Jeong and Changhee Jung. PMEM-spec: Persistent memory speculation (strict persistency can trump relaxed persistency). In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, pages 517–529, New York, NY, USA, April 2021. Association for Computing Machinery.

[32] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. Efficient persist barriers for multicores. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 660–671, New York, NY, USA, December 2015. Association for Computing Machinery.

[33] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. SplitFS: Reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 494–508, New York, NY, USA, October 2019. Association for Computing Machinery.

[34] Tomasz Kapela. An introduction to pmemcheck (part 1) - basics. https://pmem.io/blog/2015/07/an-introduction-to-pmemcheck-part-1-basics/, July 2015.

[35] T. Kawahara, K. Ito, R. Takemura, and H. Ohno. Spin-transfer torque RAM technology: Review and prospect. *Microelectronics Reliability*, 52(4):613–627, April 2012.

[36] Terence Kelly. Is persistent memory persistent? *Communications of the ACM*, 63(9):48–54, August 2020.

[37] Artem Khyzha and Ori Lahav. Taming x86-TSO persistency. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–29, January 2021.

[38] Michalis Kokologiannakis, Ilya Kaysin, Azalea Raad, and Viktor Vafeiadis. PerSeVerE: Persistency semantics for verification under ext4. *Proceedings of the ACM on Programming Languages*, 5(POPL):43:1–43:29, January 2021.

[39] Michalis Kokologiannakis, Iason Marmanis, Vladimir Gladstein, and Viktor Vafeiadis. Truly stateless, optimal dynamic partial order reduction. *Proceedings of the ACM on Programming Languages*, 6(POPL):49:1–49:28, January 2022.

[40] Michalis Kokologiannakis and Viktor Vafeiadis. HMC: Model Checking for Hardware Memory Models. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, pages 1157–1171, New York, NY, USA, March 2020. Association for Computing Machinery.

[41] Michalis Kokologiannakis and Viktor Vafeiadis. GenMC: A Model Checker for Weak Memory Models. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification*, pages 427–440, Cham, 2021. Springer International Publishing.

[42] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A Cross Media File System. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 460–477, New York, NY, USA, October 2017. Association for Computing Machinery.

[43] Philip Lantz, Subramanya Dulloor, Sanjay Kumar, Rajesh Sankaran, and Jeff Jackson. Yat: A validation framework for persistent memory software. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 433–438, Philadelphia, PA, June 2014. USENIX Association.

[44] Qingrui Liu, Joseph Izraelevitz, Se Kwon Lee, Michael L. Scott, Sam H. Noh, and Changhee Jung. iDO: Compiler-Directed Failure Atomicity for Nonvolatile Memory. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 258–270, October 2018.

[45] Sihang Liu, Korakit Seemakhupt, Yizhou Wei, Thomas Wenisch, Aasheesh Kolli, and Samira Khan. Cross-Failure Bug Detection in Persistent Memory Programs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, pages 1187–1202, New York, NY, USA, March 2020. Association for Computing Machinery.

[46] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan. PMTest: A Fast and Flexible Testing Framework for Persistent Memory Programs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 411–425, New York, NY, USA, April 2019. Association for Computing Machinery.

[47] Virendra J. Marathe, Margo Seltzer, Steve Byan, and Tim Harris. Persistent memcached: Bringing legacy code to Byte-Addressable persistent memory. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, Santa Clara, CA, July 2017. USENIX Association.

[48] Iason Marmanis and Viktor Vafeiadis. SMT-Based Verification of Persistency Invariants of Px86 Programs. In Akash Lal and Stefano Tonetta, editors, *Verified Software. Theories, Tools and Experiments.*, Lecture Notes in Computer Science, pages 92–110, Cham, 2023. Springer International Publishing.

[49] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast crash recovery in RAMCloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 29–41, New York, NY, USA, October 2011. Association for Computing Machinery.

[50] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. Memory persistency. *ACM SIGARCH Computer Architecture News*, 42(3):265–276, June 2014.

[51] Azalea Raad, Luc Maranget, and Viktor Vafeiadis. Extending Intel-x86 consistency and persistency: Formalising the semantics of Intel-x86 memory types and non-temporal stores. *Proceedings of the ACM on Programming Languages*, 6(POPL):1–31, January 2022.

[52] Azalea Raad and Viktor Vafeiadis. Persistence semantics for weak memory: Integrating epoch persistency with the TSO memory model. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):137:1–137:27, October 2018.

[53] Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis. Persistency semantics of the Intel-x86 architecture. *Proceedings of the ACM on Programming Languages*, 4(POPL):11:1–11:31, January 2020.

[54] Azalea Raad, John Wickerson, and Viktor Vafeiadis. Weak persistency semantics from the ground up: Formalising the persistency semantics of ARMv8 and transactional models. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):135:1–135:27, October 2019.

[55] Jinglei Ren, Jishen Zhao, Samira Khan, Jongmoo Choi, Yongwei Wu, and Onur Mutlu. ThyNVM: Enabling software-transparent crash consistency in persistent memory systems. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 672–685, New York, NY, USA, December 2015. Association for Computing Machinery.

[56] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. X86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 53(7):89–97, January 2010.

[57] Zhihang Sun, Hongyu Fan, and Fei He. Consistency-preserving propagation for SMT solving of concurrent program verification. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA2):158:929–158:956, October 2022.

[58] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M. Swift. Aerie: Flexible file-system interfaces to storage-class memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 1–14, New York, NY, USA, April 2014. Association

for Computing Machinery.

[59] Xiaojian Wu and A. L. Narasimha Reddy. SCMFS: A file system for storage class memory. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 1–11, New York, NY, USA, November 2011. Association for Computing Machinery.

[60] Jian Xu and Steven Swanson. NOVA: A log-structured file system for hybrid Volatile/Non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 323–338, Santa Clara, CA, February 2016. USENIX Association.

[61] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. NOVA-Fortis: A Fault-Tolerant Non-Volatile Main Memory File System. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 478–496, New York, NY, USA, October 2017. Association for Computing Machinery.

[62] Liangze Yin, Wei Dong, Wanwei Liu, Yunchou Li, and Ji Wang. YOGAR-CBMC: CBMC with Scheduling Constraint Based Abstraction Refinement. In Dirk Beyer and Marieke Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 422–426, Cham, 2018. Springer International Publishing.

[63] Liangze Yin, Wei Dong, Wanwei Liu, and Ji Wang. Scheduling constraint based abstraction refinement for weak memory models. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE '18, pages 645–655, New York, NY, USA, September 2018. Association for Computing Machinery.

[64] Liangze Yin, Wei Dong, Wanwei Liu, and Ji Wang. On Scheduling Constraint Abstraction for Multi-Threaded Program Verification. *IEEE Transactions on Software Engineering*, 46(5):549–565, May 2020.

[65] Bohong Zhu, Youmin Chen, Qing Wang, Youyou Lu, and Jiwu Shu. Octopus+: An RDMA-Enabled Distributed Persistent Memory File System. *ACM Transactions on Storage*, 17(3):19:1–19:25, August 2021.

A Proof

Theorem 2. Given an instrumented execution G^i valid under x86-TSO, the non-volatile state s_o induced by rrf is reachable under G^i iff $\mathsf{cons}_{\mathsf{DPTSO}}(G^i)$ holds.

Proof. To clarify, we use suffixes TSO and DPTSO to distinguish the respective co and hb order in Definition 5 and Definition 8. Assume that there are m shared variables $\mathcal{V}_p = \{x_1, \ldots, x_m\}$ and $|G^i.E \cap \mathsf{W}| + |G^i.E \cap \mathsf{FL}| = n$, i.e. there are n store and flush events in total.

($\leftarrow$): Suppose $\mathsf{cons}_{\mathsf{DPTSO}}(G^i)$ holds.

To prove s_o is reachable, let $\mathsf{hb}_{\mathsf{TSO}} = \mathsf{hb}_{\mathsf{DPTSO}}$ and $\mathsf{co}_{\mathsf{TSO}} = \mathsf{co}_{\mathsf{DPTSO}}$. We first arbitrarily construct a persist order nvo that satisfies the two axioms per Definition 1. The first axiom requires nvo to conform with the per-location store order of $\mathsf{hb}_{\mathsf{TSO}}$, which is exactly $\mathsf{co}_{\mathsf{TSO}}$ in this case. Note nvo is a total order over all stores and flushes. Let the sequence of events induced by nvo be $e_1, \ldots, e_n$, and for each variable x_i, let $s_o(x_i)$ equals the store e_{k_i} where $1 \le k_i \le n$.

Let e_f be the last flush event in the sequence. Consider $u = max(k_1, k_2, \ldots, k_m, f)$. Now we have the prefix $\bar{e} = e_1, \ldots, e_u$ and try to adjust nvo such that $\bar{e}$ induces s_o. In other words, it requires $\forall 1 \le i \le m, \forall k_i < j \le u.\mathtt{loc}(e_j) \neq x_i$ holds.

This is ensured by repeatedly reordering events in nvo while adhering to the two axioms. At each step, we pick a shared variable x_i for which the above condition does not hold and find the event e_p such that $\forall k_i < j \le u.\mathtt{loc}(e_j) = x_i \rightarrow j \le p$, i.e. e_p is the last store to x_i in the range $[e_{k_i}, e_u]$. We then rearrange nvo so that e_p succeeds e_u. Apparently, the new persist order does not infringe the first axiom, since e_p is the last store in the range and no $\mathsf{co}_{\mathsf{TSO}}$ order is violated by the reorder.

We now prove that the second axiom is not violated. From the above assumption, we know $(e_{k_i}, e_p) \in \mathsf{co}_{\mathsf{TSO}}$, so for any flush event e_q on x_i we also have $(e_q, e_p) \in \mathtt{dtpo}$. Since $\mathsf{cons}_{\mathsf{DPTSO}}(G^i)$ holds, the first requirement of Definition 8 gives $(e_p, e_q) \notin (\mathtt{ppo} \cup \mathtt{rf} \cup \mathtt{fr} \cup \mathsf{co}_{\mathsf{DPTSO}})^+$ which simplifies to $(e_p, e_q) \notin \mathsf{hb}_{\mathsf{TSO}}$. In this case, the premise of the second axiom does not hold, and thus e_p is not nvo-ordered with any flush events. e_p is therefore safe to be reordered.

Since the reorder of e_p in nvo does not violate the two axioms, after finite steps, the prefix $\bar{e}$ must satisfy the aforementioned condition and induces s_o, thus we have proved s_o is reachable under G^i.

($\rightarrow$): Suppose s_o is reachable, then for some $\mathsf{co}_{\mathsf{TSO}}$, $\mathsf{hb}_{\mathsf{TSO}}$, there is a persist order nvo and its prefix $\bar{e} = e_1, \ldots, e_u$ that induces s_o. Let $\mathsf{co}_{\mathsf{DPTSO}} = \mathsf{co}_{\mathsf{TSO}}$. For each variable x_i, let $s_o(x_i)$ equals the store e_{k_i} where $1 \le k_i \le u$.

Assume some store e_p to x_i happens after e_{k_i}, then $(e_{k_i}, e_p) \in \mathsf{co}_{\mathsf{TSO}}$. By definition of a reachable state, we have $p > u$. Note that the prefix contains all flush events. Therefore, all flushes e_q to x_i must be nvo-ordered before e_p. This entails that $(e_q, e_p) \in \mathsf{hb}_{\mathsf{TSO}}$, otherwise the first axiom of nvo is violated.

Now consider the validity of G^i under DPTSO. Requirement (2) of Definition 8 follows directly from Definition 5. Suppose that requirement (1) is violated. Since G^i is valid under x86-TSO, there must be a flush event e_q, store event e_{k_i} and e_p on x_i such that $(e_q, e_p) \in \mathtt{dtpo}$ and $(e_p, e_q) \in \mathsf{hb}_{\mathsf{DPTSO}}$. From the reasoning above, we have $(e_q, e_p) \in \mathsf{hb}_{\mathsf{TSO}}$.

We now consider the path from e_p to e_q on the directed graph. If no dtpo edge is on the path, then we have $(e_p, e_q) \in \mathsf{hb}_{\mathsf{TSO}}$. If there are one or more dtpo edge on the directed path, we note that any $(e'_p, e'_q) \in \mathtt{dtpo}$ entails $(e'_p, e'_q) \in \mathsf{hb}_{\mathsf{TSO}}$ by the argument above as well. Therefore we also have $(e_p, e_q) \in \mathsf{hb}_{\mathsf{TSO}}$ In either case, it contradicts the assumption that G^i is valid under x86-TSO. We hereby prove requirement (1) holds, i.e. $\mathsf{cons}_{\mathsf{DPTSO}}(G^i)$ holds. □

Lemma 2. If for every alternation G' of G, $\mathsf{cons}_{\mathsf{TSO}}(G')$ does not hold, then the observed state s_o must be unreachable under x86-TSO.

Proof. Suppose the state is reachable under x86-TSO and let G'' be an execution that induces this state, thus $\mathsf{cons}_{\mathsf{TSO}}(G'')$ holds. Since x86-TSO (Definition 5) only requires acyclicity of certain orders, by restricting $G''.E$ and the ordering constraints of G'' on G, we can always construct an execution G' such that $\mathsf{cons}_{\mathsf{TSO}}(G')$ also holds. G' is an alternation of G, thus contradicts with the assumption. □

B Atomic Block

Intuitively, regarding the recovery observer as atomic forces all read events in it to happen at the same time.

Definition 9. Given a partial instrumented execution G', regarding the recovery observer as atomic block yield the partial execution $\mathtt{atomic}(G') = (E', E'_0, \mathtt{po}', \mathtt{rf}')$ such that

- $E' = (G'.E \setminus \mathsf{REC}) \cup \{e_r\}$
- $E'_0 = E_0$
- $\mathtt{po}' = \mathtt{po}$
- $\mathtt{rf}' = \mathtt{rf} \setminus \mathtt{rrf} \cup \{(e_1, e_r) \mid \exists e_2 \in \mathsf{REC}.(e_1, e_2) \in \mathtt{rrf}\}$

where e_r is a fresh event with $\mathtt{Tid}(e_r) = P_r$.