SMT-based Verification of Persistency Invariants of Px86 Programs

Iason Marmanis and Viktor Vafeiadis

MPI-SWS, Germany
{imarmanis,viktor}@mpi-sws.org

Abstract. While non-volatile memory (NVM) promises to be both performant and durable, the semantics provided by the hardware architectures are rather subtle and significantly complicate reasoning about the possible observed state after a crash.
Starting from recent persistency extension of the x86 model, we present the first automated approach for proving invariants about the persistent state of bounded NVM programs. Our approach works by encoding the program's semantics along with its intended invariants into a compact logical formula and querying an SMT solver for its satisfiability. We propose two alternative encodings, which differ in the way the notion of a crash is encoded. For a collection of small to medium-size benchmarks, our implementation is able to detect or prove absence of persistency bugs in time ranging from a couple of seconds to some minutes.

1 Introduction

Non-volatile memory (NVM) technology can yield large performance improvements in applications that need to persist their data, since they can do so by issuing memory writes to addresses mapped to NVM. Achieving these performance improvements, however, is highly non-trivial because memory writes have rather complex persistency semantics. They are generally persisted neither synchronously nor in program order, unless programmers insert special fence and cache-line flush instructions at the appropriate program points.

Due to the high cost of these instructions, programmers often insert fewer fences than necessary, which can lead to data corruption upon a power failure, thereby negating the benefits of NVM. To date, there are a few tools that can help programmers with fence/flush placements, but sacrifice precision and/or soundness for scalability. Jaaru [10] is a stateless model checker that does not explore all program executions exhaustively and so cannot be used to prove absence of bugs. Static analysis approaches assume that all appropriately annotated data is to be flushed, thereby requiring many redundant flush instructions. Dynamic analysis (testing) approaches (e.g., [18, 20]) can achieve precision but do not provide any correctness guarantees beyond the executions actually explored, and so they can be used only to find bugs—not to show absence of bugs.

In this work, we develop an automated approach for proving invariants about the persistent state of concurrent *bounded* (loop-free) NVM programs, which explores the whole state-space induced by both concurrency and persistency. We employ symbolic model checking: we encode the program, its semantics, and its specification as a logical formula that is satisfiable if and only if the program is incorrect, and use an SMT (Satisfiability Modulo Theories) solver to check for its satisfiability.

The main challenge is to find a suitable encoding so that satisfiability of the constructed formula can be checked in a reasonable amount of time. This is by no means easy because straightforward encodings of the program's semantics generate large formulas (typically, cubic in the size of the program) and checking satisfiability is an NP-hard problem. It is therefore important to optimize the translation because even a small increase in the formula size can quickly lead to an intractable satisfiability problem.

To do so, we first have to choose an appropriate persistency semantics to base our verification upon. Existing semantic models are either operational in terms of a machine with multiple buffers [23, 24] or multiple views of the shared state [3], or declarative/axiomatic in terms of a set of constraints over a graph representing a single program execution (e.g., [14, 24, 25]). We choose the declarative $\text{DPTSO}_{\text{syn}}$ model [14] for the x86 architecture because it can easily be encoded into propositional logic and is much more suitable for automated verification. Still, however, there are three important challenges that we need to overcome.

First, a large part of the program's state space is typically irrelevant for the specifications we want to prove (invariants about the persisted state). For instance, we do not care whether a write has persisted unless the invariant depends upon the value written by that write. To overcome this challenge, we adopt the idea of a *recovery observer* of Kokologiannakis et al. [15] from stateless model checking setting and model the invariants as an additional thread that reads the relevant memory locations.

Second, $\text{DPTSO}_{\text{syn}}$, along with the other recent x86 persistency models, places constraints on *partial* program execution graphs (i.e., up until a program crash). Directly encoding these partial graphs (e.g., by introducing variables describing whether each event belongs to the executed prefix of the program) generates a huge state space, which may slow down satisfiability checking. As an alternative, we develop an equivalent reformulation of $\text{DPTSO}_{\text{syn}}$ on full execution graphs, which leads to a smaller state space without noticeably affecting verification time.

Third, there are a number of places in the $\text{DPTSO}_{\text{syn}}$ definition containing sequential compositions of relations or where the acyclicity of a relation is checked. A direct encoding of these in propositional logic leads to a formula cubic in the size of the program. While acyclicity can be more effectively encoded using theory of *integer difference logic* (IDL), this is not the case for general sequential compositions of relations. Our solution here is to employ *abstraction refinement* (SCAR) [27, 28], i.e., to avoid encoding the constraints related to the memory model in the initial formula and to add clauses on demand to rule out spurious counterexamples. As shown by He et al. [11], SCAR can nicely be integrated with the DPLL(T) framework of SMT solvers as a custom *theory solver*. He et al. [11] address the case of sequential consistency, whose definition contains only one instance of sequential composition. Here, we extend this approach to the weak consistency/persistency x86 model, which contains several instances of sequential composition, and which requires further care to handle TSO program order relaxations and the semantics of cache-line flush operations.

Putting all of this together, we have developed a prototype tool for verifying invariants about the persistent state of C programs against the DPTSO$_{\text{syn}}$ memory model. Our tool uses Z3 [19] for solving SMT queries and, following SCAR, implements parts of the encoding natively as a custom theory solver on top of Z3. We have used our tool on a collection of persistency benchmarks and were able to find or prove absence of persistency bugs.

Outline We start by reviewing the x86 model (§2). We then give an overview of our approach (§3) and present our adaptation of DPTSO$_{\text{syn}}$ over full execution graphs (§4). We then discuss the encoding of the program and its semantics as a logical formula (§5), and our implementation of the theory solver (§6). We evaluate our tool on a set of benchmarks (§7), discuss related work (§8), and conclude (§9).

2 Preliminaries

In this section, we review the terminology of axiomatic memory models and their extensions to capture the persistency semantics of x86.

2.1 Axiomatic Memory Consistency Models

Axiomatic memory models define the semantics of a program as a set of execution graphs that satisfy a certain consistency predicate. The nodes of these graphs are called *events* and represent the execution of a single memory access or a fence. Formally, an event $e \in \mathsf{Event}$ is a tuple of the form $\langle i, t, lab \rangle$, where $i \in \mathbb{N}$ is the unique *event identifier*, $t \in \mathsf{Tid}$ is the *thread identifier* of the executing thread, and *lab* is the *event label*. The event label can be one of the following types:

- a *read* label, $\mathtt{R}(l, v_R)$, accessing location $l \in \mathsf{Loc}$ and reading value $v_R \in \mathsf{Val}$;
- a *write* label $\mathtt{W}(l, v_W)$, writing value $v_W \in \mathsf{Val}$ to location $l \in \mathsf{Loc}$;
- a *read-modify-write* (RMW) label $\mathtt{U}(l, v_R, v_W)$, updating the value of location l from v_R to v_W;
- a *failed compare-and-swap* (CAS) label $\mathtt{Rex}(l, v_R)$, which reads the value v_R at location l, or
- a *memory fence* label $\mathtt{MF}$.

When applicable, the functions $\mathtt{tid}$, $\mathtt{loc}$, $\mathtt{val}_R$, and $\mathtt{val}_W$ project the thread identifier, the location (l), the value read (v_R), and the value written (v_W) of an event, respectively. For each type of label, we define the corresponding set of events; the set of read events ($\mathtt{R}$), all write events ($\mathtt{W}$), etc. Let $\mathtt{RU} \triangleq \mathtt{R} \cup \mathtt{U} \cup \mathtt{Rex}$ and $\mathtt{WU} \triangleq \mathtt{W} \cup \mathtt{U}$.

An execution graph G also comprises a number of relations on its events, representing the orders in which these events were executed and/or persisted.

Definition 1. *An execution graph G is a tuple $\langle \mathtt{E}, I, \mathtt{po}, \mathtt{rf}, \mathtt{co} \rangle$, where:*

- $\mathtt{E}$ *is a set of* events.
- $I \subseteq \mathtt{E}$ *is a set of* initialization *events, comprising a single write event $w \in \mathtt{W}_l$, for each location l.*
- $\mathtt{po} \subset \mathtt{E} \times \mathtt{E}$ *is the* program order*, which totally orders the events of each thread, and the initialization events before all other events: $I \times (\mathtt{E} \setminus I) \subseteq \mathtt{po}$.*
- $\mathtt{rf} \subseteq (\mathtt{E} \cap \mathtt{WU}) \times (\mathtt{E} \cap \mathtt{RU})$ *is the* reads-from *relation, relating events of the same location with matching values, i.e., $\langle a, b \rangle \in \mathtt{rf}$ implies that $\mathtt{loc}(a) = \mathtt{loc}(b)$ and $\mathtt{val}_W(a) = \mathtt{val}_R(b)$. Additionally, $\mathtt{rf}$ matches every read or read-modify-write to exactly on write or read-modify-write.*
- $\mathtt{co} \subseteq (\mathtt{E} \cap \mathtt{WU}) \times (\mathtt{E} \cap \mathtt{WU})$ *is the* coherence order*, defined as the disjoint union of relations $\{\mathtt{co}_l\}_{l \in \mathsf{Loc}}$, where each $\mathtt{co}_l$ is a strict total order on $\mathtt{E} \cap \mathtt{WU}_l$.*

As an example of an execution graph, in Fig. 1 we can see the only execution graph of the program P.

In the sequel, we often call execution graphs simply executions or graphs.

We define the inverse of a relation X, as $X^{-1} \triangleq \{\langle b, a \rangle \mid \langle a, b \rangle \in X\}$. We divide relations into their same-thread (internal) and different thread (external) parts, suffixed $\mathtt{i}$ and $\mathtt{e}$ respectively. For example, we write $\mathtt{rf\_i}$ for the relation $\mathtt{rf} \cap (\mathtt{po} \cup \mathtt{po}^{-1})$, and $\mathtt{co\_e}$ for the relation $\mathtt{co} \setminus (\mathtt{po} \cup \mathtt{po}^{-1})$. Additionally, given a set of events A, we write $[A]$ for the identity relation $\{\langle x, x \rangle \mid x \in A\}$. Given two relations p,q on A, we write $p;q$ for their composition. Finally, we write $rng(p)$ for the codomain of a relation p.

Each memory model defines its own consistency predicate that imposes a number of additional constraints on execution graphs. For example, *sequential consistency* (SC) [16] requires that $\mathtt{po} \cup \mathtt{rf} \cup \mathtt{co} \cup \mathtt{fr}$ be acyclic, where $\mathtt{fr} \triangleq (\mathtt{rf}^{-1}; \mathtt{co}) \setminus [\mathtt{E}]$ is the *from-reads* relation, ordering a read event r before a write event that is $\mathtt{co}$-later that the write event that r reads-from.

The x86 memory model defines the *preserved program order* $\mathtt{ppo} \triangleq \mathtt{po} \setminus (\mathtt{W} \times \mathtt{R})$ as the largest subset of $\mathtt{po}$ that avoids ordering writes with respect to $\mathtt{po}$-later reads. The x86 consistency predicate requires that: 1. $(\mathtt{rf\_i} \cup \mathtt{co\_i} \cup \mathtt{fr\_i}) \subseteq \mathtt{po}$, and 2. the *ordered-before* relation $\mathtt{ob} \triangleq (\mathtt{ppo} \cup \mathtt{rf\_e} \cup \mathtt{co\_e} \cup \mathtt{fr\_e})^+$ be irreflexive.

2.2 Modeling the Persistency Semantics of x86

To write programs with useful persistency behaviors, we introduce two types of *flush* instructions (flush and $\mathsf{flush_{opt}}$), which operate on a given cache line, and the *store fence* instruction, which waits for preceding flush instructions to complete. Accordingly, event labels are extended to also include:

- a *flush* label $\mathtt{FL}(l)$ for the cache line containing location l,
- a *flush-opt* label $\mathtt{FO}(l)$ for the cache line containing location l, and
- a *store fence* label $\mathtt{SF}$.

To simplify the presentation, we will henceforth elide the handling of flush-opt instructions, and following [14], we assume cache lines contain a single location. Our results extend straightforwardly to cover flush-opt instructions and larger cache lines, for which we refer the reader to our appendix.)

The persistency semantics of x86 is modeled by an additional constraint that describes the values of each location that can be observed after a crash. The precise definition of this constraint is where the various x86 persistency models in the literature differ.

The original persistent x86 model (Px86) [24] designates a subset of the execution's events—which includes the write events—as durable, and keeps track in execution graphs of the *non-volatile-order*, $\texttt{nvo}$, a total order on the durable events reflecting the order that they persisted. We note that $\texttt{nvo}$ contains information which is frequently irrelevant for verification, as only the last ($\texttt{nvo}$-maximal) write to each location that persisted in each location is important.

Two additional models, $\text{Px86}_{\text{view}}$ [3] and PTSO_{syn} [14], have been developed for the persistent x86 architecture, both equivalent to the Px86 model in the absence of I/O instructions. Their respective declarative models, $\text{Px86}_{\text{axiom}}$ and $\text{DPTSO}_{\text{syn}}$[1], avoid the use of the $\texttt{nvo}$ total order, and are defined similarly, with the main difference being that $\text{Px86}_{\text{axiom}}$ tracks an additional non-derived relation, thus making it less attractive for model checking. The recent PEx86 model [22], which extends Px86 to account for non-temporal writes and Intel-x86 memory types, also uses this relation to define an execution's consistency.

An important difference in $\text{DPTSO}_{\text{syn}}$ is that execution graphs of a program now also include *partial* executions, i.e., executions that crashed before they fully executed. For example, the program P in Fig. 1 contains four executions, depending on which of the instructions were executed before the crash, if any.

$\text{DPTSO}_{\text{syn}}$ captures the notion of the recovered write after a crash using a *memory assignment* μ, with $\mu \in \mathsf{Loc} \to (\mathtt{E} \cap \mathtt{WU})$, that maps each location l to the last persisted event of that location. The definition of the execution graph is extended accordingly to include the memory assignment.

To determine an execution's consistency, $\text{DPTSO}_{\text{syn}}$ defines the *derived TSO propagation order* $\texttt{dtpo}$, and extends $\texttt{ob}$ to also include $\texttt{dtpo}$, i.e.,

$$\texttt{ob} \triangleq (\texttt{ppo} \cup \texttt{rf\_e} \cup \texttt{co\_e} \cup \texttt{fr\_e} \cup \texttt{dtpo})^+$$

The $\texttt{ppo}$ relation is also redefined to reflect the additional allowed reorderings introduced by the new instructions. For example, flush instructions can be reordered w.r.t. to later load instructions. We refer the reader to our appendix for the precise definition of $\texttt{ppo}$.

The $\texttt{dtpo}$ orders a flush event before all the write events in the same location that did not persist, i.e., they are $\texttt{co}$-after the write event that was recovered after the crash. Intuitively, since the flush events are synchronous, every such write must have happened after the flush, otherwise it should have also persisted.

[1] Khyzha et al. [14] actually present two versions of $\text{DPTSO}_{\text{syn}}$. Throughout this paper we use the second version, which uses the coherence order to define consistency

As an example, consider again the program P in Fig. 1 where all the variables are zero-initialized, and assume that after recovery the variable d contains the initial value, while f reads the value one. Then, $\texttt{dtpo}$ would order the flush instruction before the $\texttt{ppo}$-earlier write instruction to d, leading to an $\texttt{ob}$ loop, which renders the execution inconsistent.

$$\texttt{dtpo} \triangleq \bigcup_{x \in \mathsf{Loc}} [\texttt{FL}_x] \times rng(\mu; \texttt{co}; [\texttt{WU}_x])$$

3 Overview

Programs that use NVM do not differ from regular volatile-memory programs in the way they access the memory. Programmers, however, have some expectations about the persistent state of their programs, e.g., that some data structure will be in a consistent state even if the program crashes mid-execution.

We formalize such expectations as *persistency invariants*, i.e., assertions which must hold at any post-crash state of the program. We illustrate with the following example. The program in §3 writes some data to the variable d, flushes the cache-line of d, and finally sets the flag f. The programmer's intention is that after a crash, if the flag f is observed to be set then the write to d will also be observed. This can be made explicit by annotating the program with the persistency invariant $f \Rightarrow d$.

```
d := 1;
flush d;
f := 1;
```

P

$$\begin{tikzcd}
\texttt{init} \arrow[d] \arrow[d, dotted, bend left] \arrow[ddd, dotted, bend left=60] \\
\texttt{W}(d,1) \arrow[d] \\
\texttt{flush}(d) \arrow[d] \\
\texttt{W}(f,1)
\end{tikzcd}$$

```
if (f)
  assert(d);
```

Rec

Fig. 1: A program P and its execution graph, along with its recovery routine Rec

3.1 Modeling Recovered Values

It is convenient to think of a persistency invariant as a special routine that runs after the program crashes and checks for any violation of the property of interest. We assume the that such recovery routines do not contain any write instructions. The recovery routine for our previous example is depicted in Fig. 1. Following Kokologiannakis et al. [15], since a crash can occur at any point during the execution of a program, one can model the recovery routine as an additional thread running in parallel to the code, which is subject to somewhat different constraints regarding the possible values it can read.

To encode those constraints, we extend the set of event labels (§ 2.1) to include recovery read labels, $\texttt{Rec}(l, v_R)$, which correspond to the read instructions of the recovery routine. We also rewrite the definition of $\texttt{dtpo}$ so that it does not require the memory assignment μ, instead recovering it from the writes that the recovery reads read from. To this end, recall that $\texttt{dtpo}$ orders the flush events on a location x before the writes that happened $\texttt{co}$-after the last persisted write of x, which are exactly the events in $rng([\texttt{Rec}_x]; \texttt{rf}^{-1}; \texttt{co})$. By introducing a new relation *flush-before* ($\texttt{fb}$) which orders every flush event with the recovery reads on the same location, we can rewrite $\texttt{dtpo}$ simply as $\texttt{fb}; \texttt{fr}$.

3.2 Symbolic Verification

Symbolic verification requires to construct a logical formula Φ that captures the program, its semantics, and its specification. Here we provide a high-level overview of the construction of Φ leaving the details for §5 and §6.

Representing Execution Graphs. To represent the set of possible execution graphs in Φ, we associate with each instruction of the program an event in the execution graph, and introduce variables denoting the various relations between events of the graph. So, for example, for each pair $\langle w, r\rangle$ of a write event w and a read event r, we introduce the variable $rf_{w,r}$ which is set whenever r reads from w.

Since, however, it is possible that not all instructions of the program will be executed, for each instruction n, we associate a formula $\texttt{enabled}(n)$ representing whether the instruction was actually executed, i.e., the control flow reached it. For memory access instructions, we also associate terms such as $\texttt{loc}(n)$ containing the location accessed and $\texttt{val}(n)$ the value read/written.

These variables and terms allow us to express their intended meaning as a number of basic constraints stating, for example, that $rf_{w,r}$ implies that $\texttt{loc}(w) = \texttt{loc}(r)$ and $\texttt{val}(w) = \texttt{val}(r)$, that each read event reads from some write event, and that $\texttt{co}_l$ is a total order for each location l. The exact constraints are shown in § 5.2.

Execution Graph Consistency. Apart from these basic axioms, we also need to encode the specific consistency predicate of the memory model, which is typically an acyclicity constraint on a set of relations including $\texttt{rf}$, $\texttt{co}$, and $\texttt{fr}$. Prior work identified that the cubic encoding stemming from the relation composition needed for $\texttt{fr}$ dominates the resulting formula [27, 28], and proposed *abstraction refinement* to circumvent it [11, 27, 28].

We adapt the approach of He et al. [11], avoiding completely the encoding of the consistency predicate and delegating consistency checking of the explored execution graph to a custom theory solver, which judges the satisfiability of assignment to the variables concerning the memory model (e.g., rf and co). We discuss the details of our theory solver for $\text{DPTSO}_{\text{syn}}$ in §6.

Modeling Crashes. The final issue we must address is how to encode the notion of a crash, i.e., the possibility that some instructions were not executed because the program terminated prematurely. This is necessary because the semantics of programs under $\text{DPTSO}_{\text{syn}}$ (and similarly in other models) is defined w.r.t. partial execution graphs.

For example, consider the program P in Fig. 1, and a recovery routine that asserts that the value recovered for d is 1. The approach outlined so far would deem this program safe because it would only consider the full execution where the write to d is followed by a flush.

The straightforward approach is to lift our encoding of `enabled`, so that it reflects not only whether control flow reached the corresponding instruction, but also whether the program did not crash until that point. To do so, we need to include one additional boolean variable for each node, capturing whether execution crashed just before the execution of the instruction. We discuss this approach further in §5.

To partially alleviate the need for these additional variables, we present in §4 an adaptation of the DPTSO$_{\text{syn}}$ semantics which defines consistent executions only in terms of full execution graphs. We discuss in §5 the modifications needed in the encoding to support our adaptation.

4 Adapting the DPTSO$_{\text{syn}}$ model

In this section, we reformulate DPTSO$_{\text{syn}}$ in terms of full execution graphs and show that our reformulated model, DPTSO$_{\text{syn,full}}$, is equivalent to DPTSO$_{\text{syn}}$.

To define DPTSO$_{\text{syn,full}}$, we first have to adapt the definition of `dtpo` concerning the synchronous nature of the execution of flush operations. We can no longer simply assume that all flush operations have executed before any write that was not persisted, because the crash may well have happened much before those flushes. Instead, only on the flushes that are *observed* to have been executed should be ordered before any non-persisted writes. Such flushes are those in the `porf`-prefix of a write that has been observed after the crash, where `porf` $\triangleq (\texttt{po} \cup \texttt{rf})^{+}$. Formally, this is:

$$\texttt{dtpo} \triangleq \bigcup_{x \in \mathsf{Loc}} dom([\texttt{FL}_x]; \texttt{porf}; \mu^{-1}) \times rng(\mu; \texttt{co}; [\texttt{WU}_x])$$

We illustrate our argument using the example in Fig. 1. The program consists of two write instructions to different locations, separated by a flush instruction to the first location. Under DPTSO$_{\text{syn}}$, it is not consistent to recover the value 1 for f and the initial value for d, since the flush event is `dtpo`-before the `ppo`-later write, thus creating an `ob` cycle. However, it is also not consistent to recover the initial value for d, regardless of the recovered value for f, for the same reason.

If we interpret a graph as any possible execution prefix that resulted from a crash, we would want to still disallow the former behavior, while allowing the latter. Indeed, both executions that crash before the flush instruction permit the behavior in question.

Intuitively, the only reason to rule out these executions is if we can *observe* that the flush instruction indeed happened, i.e., it is in the `porf`-prefix of a write that was recovered after the crash, and thus the corresponding instruction was executed. This is the case if we recover the write to f after the crash. In this scenario, the flush has executed, and is thus included in $\texttt{dtpo}$, resulting in a $\texttt{ob}$ cycle.

We next establish the equivalence between the two models with the following two lemmas. Their full proofs can be found in our technical appendix.

Lemma 1. *If a partial execution G generated by a program P is consistent under* $\text{DPTSO}_{\text{syn}}$*, then there is a full execution G' generated by P that extends G and is consistent under* $\text{DPTSO}_{\text{syn,full}}$*.*

Proof sketch. We generate G' by repeatedly adding events to G following the program in a way that respects $\texttt{po}$, and making each event *coherence-maximal* at the point it was added. A write event is coherence-maximal if it is the $\texttt{co}$-latest event, and a read event is coherence-maximal if it reads from the coherence-maximal write. Observe that this construction avoids adding any edge towards the events of G, as well as any *dtpo* edge that starts from the new events, which leads to G' being $\text{DPTSO}_{\text{syn,full}}$-consistent. □

Lemma 2. *If a full execution G' generated by a program P is consistent under* $\text{DPTSO}_{\text{syn,full}}$*, then there is a* $\texttt{porf}$*-prefix of G' that is* $\text{DPTSO}_{\text{syn}}$*-consistent.*

Proof sketch. Take $G \triangleq G'|_{dom(\texttt{porf};\mu^{-1})}$ to be the $\texttt{porf}$-prefix of the recovered events of G'. Observe that G is $\text{DPTSO}_{\text{syn,full}}$-consistent and that $\text{DPTSO}_{\text{syn}}$ and $\text{DPTSO}_{\text{syn,full}}$ only differ in the definition of $\texttt{dtpo}$, which, by construction of G, gives rise to the same relation. Therefore G is also $\text{DPTSO}_{\text{syn}}$-consistent, as required. □

5 Symbolic Encoding

5.1 From Verification to Formula Satisfiability

Following the standard conventions in bounded model checking, we assume that programs are loop-free and in *static single assignment* form (SSA) [4], whereby each variable is assigned to only once. Conversion to such format is possible by bounding the loop iteration depth and standard compiler code transformations (e.g., introducing fresh variable names for each assignment to a variable). From this form, a logical formula Φ_{SSA} is generated, which represents the data and control flow. For shared read memory accesses, the value that is read is left unspecified and is restricted by a formula Φ_{MM} that captures the memory model's semantics. Lastly, the program's specification is encoded in a formula Φ_{SPEC}, which indicates the violation of some property. The program is deemed safe if $\Phi \triangleq \Phi_{SSA} \wedge \Phi_{MM} \wedge \Phi_{SPEC}$ is unsatisfiable, which can be checked by an SMT solver. Existing techniques differ on how Φ_{MM} is encoded.

5.2 Memory model encoding

Along with the SSA form, an *event graph* is constructed, with each node corresponding to a memory event. As discussed in §3, each node n is associated with formulas $\texttt{loc}(N), \texttt{val}(n)$, and $\texttt{enabled}(n)$. A node also contains an event label, specifying the type of the instruction it corresponds to.

As an example, consider the program *Rec* in Fig. 1. The event graph will contain two events r_f and r_d, for the load instructions to f and d, respectively. The Φ_{SPEC} component of the formula Φ that corresponds to *Rec* is $\texttt{enabled}(d) \wedge \texttt{val}(d) = 0$, where $\texttt{enabled}(d) \triangleq \texttt{enabled}(f) \wedge \texttt{val}(f) \neq 0$ and $\texttt{enabled}(f) \triangleq$ *true*. Both $\texttt{val}(f) \triangleq u_f$ and $\texttt{val}(d) \triangleq u_d$ are left unspecified, and will be restricted by Φ_{MM}.

Following He et al. [11], we encode directly into propositional logic only some basic axioms about the memory model (e.g., that every read reads from some write), whose size is at most quadratic in the size of the program.

Specifically, we introduce one boolean variable $rf_{w,r}$ for each pair of write event w and read event r denoting the presence of a `rf`-edge from w to r, and one boolean variable $co_{w,w'}$ for each pair of write events denoting the presence of a `co`-edge from w to w'.

Given a read event r and a pair w, w' of write events, we encode the following basic axioms:

$$\texttt{enabled}(r) \implies \bigvee_{w \in \texttt{W}} rf_{w,r}$$

$$rf_{w,r} \implies \texttt{enabled}(w) \wedge \texttt{enabled}(r) \wedge \texttt{valw}(w) = \texttt{valr}(r) \wedge \texttt{loc}(w) = \texttt{loc}(r)$$

$$co_{w,w'} \vee co_{w',w} \iff \texttt{enabled}(w) \wedge \texttt{enabled}(w') \wedge \texttt{loc}(w) = \texttt{loc}(w')$$

The first two axioms state that every enabled read reads from some write, which is also enabled and acts on the same location. The third axiom captures the totality of `co` for same-location writes.

Note that we do not encode the functionality of `rf`—that a read cannot read from two different writes—because this constraint does not affect the consistency of a plain execution graph (an `rf` relation violating functionality can be modified to one satisfying it by removing `rf` edges).

As an optimization, for events that we can statically determine that they do not access the same location, we avoid introducing new variables and encoding the corresponding constraints.

5.3 Encoding x86 Consistency

To capture the reordering semantics of x86, we also have to add some additional variables that correspond to the ppo edges, instead of relying on the statically predetermined `po` edges. To this end, for each pair $\langle x, y\rangle$ of ppo-related events, we add a boolean variable $ppo_{x,y}$, and require this variable to be set only when both x and y are enabled.

As an optimization, we avoid encoding the transitive closure of ppo, i.e., we avoid introducing a new variable $ppo_{x,y}$ if it can be derived from a pair of variables $ppo_{x,z}$ and $ppo_{z,y}$.

5.4 Encoding DPTSO$_{\text{syn}}$

Finally, to fully encode DPTSO$_{\text{syn}}$ we need to account for 1. the additional `dtpo` edges 2. the fact that, due to a possible crash, a prefix of the program could have been executed.

As discussed in §3, `dtpo` can be rewritten as `fb`;`fr`, where `fb` is a relation ordering every flush event f on a location x to all the recovery read events r on the same location ($\texttt{Rec}_x$). Thus it suffices to add a new boolean variable $\mathit{fb}_{f,r}$ for each such pair of events, and capture the intended meaning with the following constraint:

$$\mathit{fb}_{f,r} = \texttt{enabled}(f) \wedge \texttt{enabled}(r) \wedge \texttt{loc}(f) = \texttt{loc}(r)$$

A straightforward way to encode the notion of a crash, is to further add a new boolean variable crash_n, for each node n of the event graph, reflecting the fact that execution crashed just before the execution of the corresponding memory access. Encapsulating this inside the $\texttt{enabled}(n)$ formula of each node, so that it now signifies that control flow reached the memory accessing instruction without crashing, gives us a full encoding for DPTSO$_{\text{syn}}$ without the need of any additional change.

5.5 Alternative Crash Encoding

Alternatively, we can employ our adaptation (DPTSO$_{\text{syn,full}}$) of DPTSO$_{\text{syn}}$ to partially circumvent the need for these additional *crash* variables.

DPTSO$_{\text{syn,full}}$ defines the semantics of x86 programs in terms of full execution graphs, and changes the definition of `dtpo` to achieve this. Following the same reasoning as in §3, it is easy to see that `dtpo` can again be rewritten as `fb`;`fr`. Now, however, only the flush events that are in the `porf`-prefix of a recovery read event take part in `fb`.

To capture this, we again introduce a boolean variable *crash* for each flush event f and modify the fb constraint to:

$$\mathit{fb}_{f,r} = \texttt{enabled}(f) \wedge \texttt{enabled}(r) \wedge \texttt{loc}(f) = \texttt{loc}(r) \wedge \neg \mathit{crash}_f$$

The intended meaning is that if an enabled node f has its crash_f variable set to true, it is was not executed due to a crash, and thus it cannot be `porf`-before a recovery read event. This is checked and enforced by our custom theory solver (§6).

6 Theory solver for DPTSO$_{\text{syn}}$

6.1 Preliminaries

Given a formula involving atoms from some first-order theories, DPLL(T) [9] extends DPLL [5, 6] by replacing each atom with a new boolean variable, creating its *boolean abstraction*, whose satisfiability is determined by the SAT core of the solver. In case a model is produced, i.e., the boolean abstraction is satisfiable, the theory solvers should be consulted to judge whether the model is also satisfiable in the background theories.

This procedure can also take place *online*, with the theory solvers checking the consistency of *partial* assignments as they are being explored by the SAT solver. In case an inconsistency in detected, a *conflict clause* is generated that captures the inconsistency. The conflict clause is *propagated* to the SAT solver, which initiates a *backjump*, reverting the last N assignments. The conflict clause prevents the same assignment from being explored, and additionally providing some knowledge of the background theory to the SAT solver. The latter is also supported independently of an inconsistency's existence, i.e., the theory solver can propagate additional clauses to assist the SAT solver's exploration.

6.2 Z3 User Propagator

We base our implementation of the theory solver on Z3's user propagator infrastructure, which allows implementing a custom theory solver externally without the need to modify the Z3 codebase.

The user propagator allows the client of Z3's library to track some of its boolean variables, and register a callback that is initiated each time a value is assigned to one of them. The callback's implementation can respond by propagating a logical consequence, whose antecedent is a subset of the set variables, and the consequent is an arbitrary boolean expression. In case the consequent is *false*, the negation of the antecedent corresponds to the conflict clause.

The user propagator's interface provides two additional callbacks to inform the solver about (1) *backtracking* points, and (2) initiation of a backtrack, so that the theory solver reverts all assignments up to the last backtracking point.

6.3 Implementation

Given the event graph (2.1) of the program together with its (static) `po` edges, our theory solver is responsible for judging the satisfiability of the assignments to the rf, co, fb, and $crash$ variables, which corresponds to the consistency of the execution graph that is being explored by the SAT solver.

To detect violations of DPTSO$_{\text{syn}}$ consistency, it needs to check for 1. `rf_i`, `co_i`, or `fr_i` edges that contradict `po`, and 2. cycles consisting of `ppo`, `rf_e`, `co_e`, `fr_e`, and `fb` edges. We note that `fr` edges are derived from their constituent edges ($\texttt{fr} \triangleq \texttt{rf}^{-1}; \texttt{co}$).

Our adaptation DPTSO$_{\text{syn,full}}$ additionally requires detecting paths of `po` and `rf` edges, which start from a flush event f with $crash_f$ set to true, and end in a recovery read event.

Detecting Inconsistent Assignments The non-trivial violations (i.e., excluding $(\texttt{rf\_i} \cup \texttt{co\_i} \cup \texttt{fr\_i}) \nsubseteq \texttt{po}$) require an algorithm to detect a cycle or a certain path in the event graph.

These algorithms need to be incremental, in order to quickly rule out inconsistent assignments, and amendable for efficient backtracking, i.e., to revert a suffix of their operations without the need to store a huge amount of state.

To incrementally detect `ob` cycles we use the incremental cycle detection (ICD) on sparse graphs of Bender et al. [2], following He et al. [11]. As noted by the authors, the correctness of the algorithm is preserved in a decremental setting, without the need to revert the changes in the computed order.

To incrementally detect `porf` paths, we use Italiano [12]'s incremental transitive closure algorithm (Italiano-ITC) on the `po` and `rf` edges of the graph, together with the optimizations suggested by Frigioni et al. [8]. Finally, extending the algorithm to support backtracking is trivial, as we only need to revert the value of the matrix's elements to false.

Clearly, the theory for $\mathrm{DPTSO}_{\mathrm{syn,full}}$ (and $\mathrm{DPTSO}_{\mathrm{syn}}$) is decidable; the satisfiability of an assignment reduces to the two aforementioned problems.

Explaining Inconsistencies Apart from detecting inconsistencies, our theory solver needs to succinctly explain them to the SAT core, by generating a conflict clause. To achieve this, we associate each edge with its *reason*, the conjunction of atoms that justify the edge's existence. For the `rf`, `co`, `fb`, and `ppo` edges, this is just the corresponding atom, set by the SAT core during the construction of model. For `fr`, it is the conjunction of the reasons of the constituent edges. We lift the notion of reasons to paths, defining the reason of a path as the the conjunction of the reasons of each constituent edge.

When an `ob` cycle is detected, during the addition of an edge $e = \langle x, y \rangle$, we find the path p from the node y to node x that contains the fewest edges assigned with a reason, i.e., all apart from the static `ppo` edges, and propagate to the SAT core the contradiction: $\mathit{reason}(p) \wedge \mathit{reason}(e) \implies \mathit{false}$.

Similarly, when a `porf` path p is detected that originates from a node x, whose crash variable crash_x is set to true, and ends in a recovery read r, we propagate to the SAT core that $\mathit{reason}(p) \wedge \mathit{crash}_x \implies \mathit{false}$.

7 Evaluation

In this section, we evaluate the overall performance of an implementation of our approach and compare our two different encodings of the x86 semantics.

To evaluate our approach, we have implemented a prototype verification tool for C programs that use NVM memory. Our tool uses LLVM/clang to transform the input program into SSA form, generates a formula as described in §5, and calls a version of Z3 [19] containing our custom theory solver to check its satisfiability. If the generated formula is satisfiable, an appropriate error message is reported back to the user.

As benchmarks, we took three recent *durably linearizable* [13] libraries from the literature: the read-write register library of Wei et al. [26] (FLIT), the persistent queue of Friedman et al. [7], and the persistent set of Zuriel et al. [29]. For each library, we constructed several multithreaded client programs that call the various methods of the libraries, In each of these benchmarks, we wrote down a persistency invariant that checks (consequences of) durable linearizability: e.g., if a certain method has been executed, then its effects have persisted. A typical invariant for a program that performs an enqueue operation followed by a write instruction might say that if the write is observed, then the enqueue operation is observed as well.

This way we obtain a set of *safe* benchmarks, i.e., whose invariants hold. Removing some of the flush operations gives us a set of *unsafe* benchmarks, where the invariants are violated.

Experimental Setup Our experiments are conducted on a Dell OptiPlex 7050 system, running Debian 11, with an Intel(R) Core(TM) i5-6600 CPU and 16 GB of RAM. We used version 4.8.17 of z3.

7.1 Overall Performance

We first evaluate the overall performance of our tool using the $\text{DPTSO}_{\text{syn}}$ encoding. For this purpose, we consider only the safe benchmarks, which are presented in Table 1. For each benchmark, along with the verification time (in seconds), we report the number of nodes in the event graph, to indicate the size of the benchmark. The name represents the client itself; for example, `e3+3dw` is a client that uses the queue library, and consists of one thread performing three enqeue operations, and three threads performing a dequeue operation, followed by a write.

The benchmarks are small client programs of persistent libraries with up to 4 threads, each invoking a couple of library operations. The library methods vary in complexity, Flit is the simplest, with each method containing at most 4 memory accesses, while the set library is the most complex, as it can be observed by the large number of nodes even when there are only two executing threads, which is the reason that our tool scales much worse for the client using it. As it can be seen, our tool succeeds in verifying these medium-sized clients, with running time ranging from under 1 second to a bit over 7 minutes.

We note that we do not compare against any existing tools because, to our knowledge, none is complete for (bounded) NVM programs. While Yat [17] and Jaaru [10] can find bugs in NVM programs, they are both incomplete and may miss behaviors arising in multi-threaded programs.

7.2 Comparison of $\textbf{DPTSO}_{\textbf{syn}}$ and $\textbf{DPTSO}_{\textbf{syn,full}}$ encodings

We next evaluate the two encodings we proposed to incorporate the notion of crash. The first encoding is based on $\text{DPTSO}_{\text{syn}}$ and represents partial graphs using an auxiliary `enabled` variable for each program node (§5). The second encoding is based on $\text{DPTSO}_{\text{syn,full}}$ (§4) and partially avoids the need of encoding the possible crashed executions of a program.

Our results are presented in Fig. 2 as a scatter diagram comparing the verification time (in seconds) of each benchmark with the two different encodings.

Table 1: Safe benchmarks: flit, queue, set ($\text{DPTSO}_{\text{syn}}$)

	Time	Nodes
ldld	0.28	29
ww	0.28	24
u+u-w	0.29	33
w+w-w	0.29	30
w+rw-w	0.30	33
u+u+u+u-w	0.39	47
2uu+uu-w	0.46	60
3uu	0.50	53
uu+uu	0.50	39
2uu+2uu-w2	0.87	75
4uu	2.08	67
2uu+2uu-w	2.39	74

	Time	Nodes
dw	0.35	70
e+d	1.46	145
ee+dw	1.65	205
e+e	3.07	157
ee+d	3.98	213
e+de	6.63	222
ee+ddw	8.54	272
ee+dd	10.99	269
e+e+d	18.16	213
e2+2dw	34.38	332
e+e+d+d	36.70	269
e+e+e	48.88	225
ee+ddw+d	59.62	328
e+de+e	60.94	281
e+de+e+d	227.13	337
e3+3dw	427.54	459

	Time	Nodes
iw	10.78	287
i	11.87	286
irw	34.13	471
ir	37.42	469
iw+rw	64.70	472
i+crw	89.31	488
i+i-w	260.89	524
i+i	416.12	505
ii+r	1151.50	688

We have also categorized our benchmarks depending on the verification result, i.e., whether the assertion holds (safe) or does not hold (unsafe).

As we observe in Fig. 2, the two encodings yield similar performance, with the relative difference never exceeding an order of magnitude. Sadly, there is no clear trend suggesting that either encoding leads to better performance. In principle, $\text{DPTSO}_{\text{syn,full}}$ partially eliminates the need for encoding the semantics of a crash, adding only some crash variables for flush events. However, by not fully encoding this, the solver always has to explore a full execution, even though a part of it is irrelevant. In contrast, while the $\text{DPTSO}_{\text{syn}}$ encoding leads to a larger state space, only the events before the crash take part in the axioms that concern the memory model, and so the basic axioms of § 5.2 are trivially satisfied for crashed events.

8 Related Work

Several researchers have formalized the persistency semantics of the x86 architecture as an extension of the original x86-TSO memory consistency model [21]. The first such model, Px86 [24], treats flush operations as asynchronous. This is corrected in later models by Khyzha et al. [14] and Cho et al. [3], who treat flush operation as synchronous. More recently, [22] extended those formalizations to cover additional features of the Intel-x86 architecture, such as non-temporal writes and memory types.

There are many verification approaches that deal with multi-threaded programs under various memory consistency models. Among the symbolic techniques, Alglave et al. [1] model program executions as a collection of partial orders and encode acyclicity constraints using integer difference logic. YOGAR-CBMC [27, 28] employs abstraction refinement to verify multi-threaded programs under sequential consistency, and weak memory models, accordingly. He et al. [11]

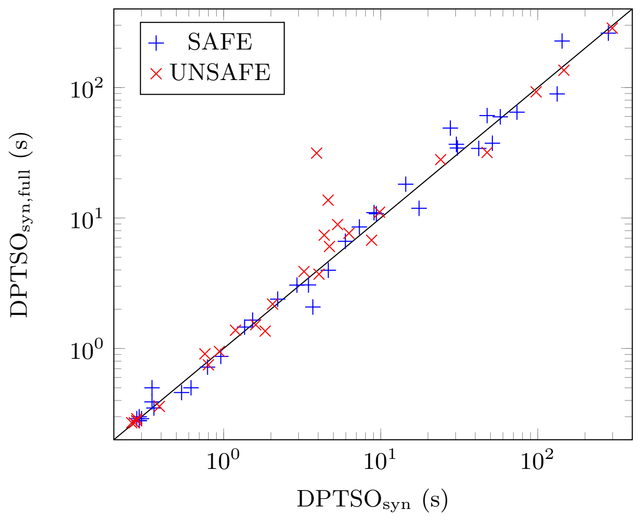

Fig. 2: Comparing the encoding of $\mathrm{DPTSO_{syn}}$ and $\mathrm{DPTSO_{syn,full}}$

propose a new ordering consistency theory for dealing with the concurrency related fragment of the encoding, and a theory solver that incrementally checks for consistency of the explored executions.

In contrast, there is much less work on model checking programs that use persistent memory. We are aware of two such works. Yat [17] eagerly explores post-crash states by injecting crashes in a collected trace, while Jaaru [10] explores only the subset of pre-crash states that is relevant in the post-crash execution. Nevertheless, both approaches are not complete for multi-threaded programs since they do not explore the concurrency-induced nondeterminism.

9 Conclusion

In this paper, we have presented an automated approach for proving invariants about the persistent state of concurrent (bounded) NVM programs. Our approach is based on symbolic model checking and uses a custom theory solver to encode certain aspects of the memory model that would otherwise lead to huge formulas. Finally, we have considered two encodings of partial executions without, however, observing any significant difference in performance between them. It may, however, be the case that the two approaches would yield a noticeable difference in performance if used in different contexts, e.g., with a stateless model checker.

Acknowledgments This work was supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 101003349).

References

[1] Jade Alglave, Daniel Kroening, and Michael Tautschnig. “Partial orders for efficient bounded model checking of concurrent software”. In: *CAV 2013*. Vol. 8044. LNCS. Berlin, Heidelberg: Springer, 2013, pp. 141–157. DOI: 10.1007/978-3-642-39799-8_9.

[2] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Robert E. Tarjan. “A New Approach to Incremental Cycle Detection and Related Problems”. In: *ACM Trans. Algorithms* 12.2 (Dec. 2015). ISSN: 1549-6325. DOI: 10.1145/2756553. URL: https://doi.org/10.1145/2756553.

[3] 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. Virtual, Canada: Association for Computing Machinery, 2021, pp. 16–31. ISBN: 9781450383912. DOI: 10.1145/3453483.3454027. URL: https://doi.org/10.1145/3453483.3454027.

[4] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. “Efficiently Computing Static Single Assignment Form and the Control Dependence Graph”. In: *ACM Trans. Program. Lang. Syst.* 13.4 (Oct. 1991), pp. 451–490. ISSN: 0164-0925. DOI: 10.1145/115372.115320. URL: https://doi.org/10.1145/115372.115320.

[5] Martin Davis, George Logemann, and Donald Loveland. “A Machine Program for Theorem-Proving”. In: *Commun. ACM* 5.7 (July 1962), pp. 394–397. ISSN: 0001-0782. DOI: 10.1145/368273.368557. URL: https://doi.org/10.1145/368273.368557.

[6] Martin Davis and Hilary Putnam. “A Computing Procedure for Quantification Theory”. In: *J. ACM* 7.3 (July 1960), pp. 201–215. ISSN: 0004-5411. DOI: 10.1145/321033.321034. URL: https://doi.org/10.1145/321033.321034.

[7] Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. “A Persistent Lock-Free Queue for Non-Volatile Memory”. In: *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP ’18. Vienna, Austria: Association for Computing Machinery, 2018, pp. 28–40. ISBN: 9781450349826. DOI: 10.1145/3178487.3178490. URL: https://doi.org/10.1145/3178487.3178490.

[8] Daniele Frigioni, Tobias Miller, Umberto Nanni, and Christos Zaroliagis. “An Experimental Study of Dynamic Algorithms for Transitive Closure”. In: *ACM J. Exp. Algorithmics* 6 (Dec. 2002), 9–es. ISSN: 1084-6654. DOI: 10.1145/945394.945403. URL: https://doi.org/10.1145/945394.945403.

[9] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. “DPLL(T): Fast Decision Procedures”. In: *Computer Aided Verification*. Ed. by Rajeev Alur and Doron A. Peled. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 175–188. ISBN: 978-3-540-27813-9.

[10] 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. New York, NY, USA: Association for Computing Machinery, 2021, pp. 415–428. ISBN: 9781450383172. URL: https://doi.org/10.1145/3445814.3446735.

[11] 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. Virtual, Canada: Association for Computing Machinery, 2021, pp. 1264–1279. ISBN: 9781450383912. DOI: 10.1145/3453483.3454108. URL: https://doi.org/10.1145/3453483.3454108.

[12] G. F. Italiano. "Amortized Efficiency of a Path Retrieval Data Structure". In: *Theor. Comput. Sci.* 48.2–3 (Dec. 1987), pp. 273–281. ISSN: 0304-3975.

[13] Joseph Izraelevitz, Hammurabi Mendes, and Michael Scott. "Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model". In: vol. 9888. Sept. 2016, pp. 313–327. ISBN: 978-3-662-53425-0. DOI: 10.1007/978-3-662-53426-7_23.

[14] Artem Khyzha and Ori Lahav. "Taming X86-TSO Persistency". In: *Proc. ACM Program. Lang.* 5.POPL (Jan. 2021). DOI: 10.1145/3434328. URL: https://doi.org/10.1145/3434328.

[15] Michalis Kokologiannakis, Ilya Kaysin, Azalea Raad, and Viktor Vafeiadis. "PerSeVerE: Persistency semantics for verification under ext4". In: *Proc. ACM Program. Lang.* 5.POPL (Jan. 2021). DOI: 10.1145/3434324. URL: https://doi.org/10.1145/3434324.

[16] Leslie Lamport. "How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs". In: *IEEE Trans. Computers* 28.9 (Sept. 1979), pp. 690–691. DOI: 10.1109/TC.1979.1675439. URL: http://dx.doi.org/10.1109/TC.1979.1675439.

[17] Philip Lantz, Subramanya Dulloor, Sanjay Kumar, Rajesh Sankaran, and Jeff Jackson. "Yat: A Validation Framework for Persistent Memory Software". In: *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*. USENIX ATC'14. Philadelphia, PA: USENIX Association, 2014, pp. 433–438. ISBN: 9781931971102.

[18] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Manabi Khan. "PMTest: A Fast and Flexible Testing Framework for Persistent Memory Programs". In: *ASPLOS 2019*. Ed. by Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck. ACM, 2019, pp. 411–425. DOI: 10.1145/3297858.3304015.

[19] Leonardo de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340. ISBN: 978-3-540-78800-3.

[20] Ismail Oukid, Daniel Booss, Adrien Lespinasse, and Wolfgang Lehner. "On Testing Persistent-Memory-Based Software". In: DaMoN '16. ACM, 2016. ISBN: 9781450343190. DOI: 10.1145/2933349.2933354.

[21] Scott Owens, Susmit Sarkar, and Peter Sewell. "A better x86 memory model: x86-TSO". In: *TPHOLs 2009*. Munich, Germany: Springer, 2009, pp. 391–407. ISBN: 978-3-642-03358-2. DOI: 10.1007/978-3-642-03359-9_27. URL: http://dx.doi.org/10.1007/978-3-642-03359-9_27.

[22] 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". In: *Proc. ACM Program. Lang.* 6.POPL (Jan. 2022). DOI: 10.1145/3498683. URL: https://doi.org/10.1145/3498683.

[23] Azalea Raad and Viktor Vafeiadis. "Persistence semantics for weak memory: Integrating epoch persistency with the TSO memory model". In: *Proc. ACM Program. Lang.* 2.OOPSLA (Oct. 2018). DOI: 10.1145/3276507. URL: https://doi.org/10.1145/3276507.

[24] Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis. "Persistency semantics of the Intel-x86 architecture". In: *Proc. ACM Program. Lang.* 4 (POPL Dec. 20, 2019), 11:1–11:31. DOI: 10.1145/3371079. URL: https://doi.org/10.1145/3371079 (visited on 06/17/2020).

[25] Azalea Raad, John Wickerson, and Viktor Vafeiadis. "Weak persistency semantics from the ground up". In: *Proc. ACM Program. Lang.* 3 (OOPSLA Oct. 10, 2019), 135:1–135:27. DOI: 10.1145/3360561. URL: https://doi.org/10.1145/3360561 (visited on 02/07/2020).

[26] Yuanhao Wei, Naama Ben-David, Michal Friedman, Guy E. Blelloch, and Erez Petrank. "FliT: A Library for Simple and Efficient Persistent Algorithms". In: *CoRR* abs/2108.04202 (2021). arXiv: 2108.04202. URL: https://arxiv.org/abs/2108.04202.

[27] Liangze Yin, Wei Dong, Wanwei Liu, and Ji Wang. "Scheduling Constraint Based Abstraction Refinement for Multi-Threaded Program Verification". In: *IEEE Transactions on Software Engineering* PP (Aug. 2017). DOI: 10.1109/TSE.2018.2864122.

[28] 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*. New York, NY, USA: Association for Computing Machinery, 2018, pp. 645–655. ISBN: 9781450359375. URL: https://doi.org/10.1145/3238147.3238223.

[29] Yoav Zuriel, Michal Friedman, Gali Sheffi, Nachshon Cohen, and Erez Petrank. "Efficient Lock-Free Durable Sets". In: *Proc. ACM Program. Lang.* 3.OOPSLA (Oct. 2019). DOI: 10.1145/3360554. URL: https://doi.org/10.1145/3360554.