# Taming x86-TSO Persistency (Extended Version)

ARTEM KHYZHA, Tel Aviv University, Israel
ORI LAHAV, Tel Aviv University, Israel

We study the formal semantics of non-volatile memory in the x86-TSO architecture. We show that while the explicit persist operations in the recent model of Raad et al. from POPL'20 only enforce order between writes to the non-volatile memory, it is equivalent, in terms of reachable states, to a model whose explicit persist operations mandate that prior writes are actually written to the non-volatile memory. The latter provides a novel model that is much closer to common developers' understanding of persistency semantics. We further introduce a simpler and stronger sequentially consistent persistency model, develop a sound mapping from this model to x86, and establish a data-race-freedom guarantee providing programmers with a safe programming discipline. Our operational models are accompanied with equivalent declarative formulations, which facilitate our formal arguments, and may prove useful for program verification under x86 persistency.

CCS Concepts: • **Computer systems organization** → *Multicore architectures*; • **Software and its engineering** → *Semantics*; • **Theory of computation** → *Concurrency*; *Program semantics.*

Additional Key Words and Phrases: persistency, non-volatile memory, x86-TSO, weak memory models, concurrency

**ACM Reference Format:**
Artem Khyzha and Ori Lahav. 2021. Taming x86-TSO Persistency (Extended Version). *Proc. ACM Program. Lang.* 1, POPL, Article 1 (January 2021), 55 pages.

## 1 INTRODUCTION

Non-volatile memory (a.k.a. persistent memory) preserves its contents in case of a system failure and thus allows the implementation of crash-safe systems. On new Intel machines non-volatile memory coexists with standard (volatile) memory. Their performance are largely comparable, and it is believed that non-volatile memory may replace standard memory in the future [Pelley et al. 2014]. Nevertheless, in all modern machines, writes are not performed directly to memory, and the caches in between the CPU and the memory are expected to remain volatile (losing their contents upon a crash) [Izraelevitz et al. 2016b]. Thus, writes may propagate to the non-volatile memory later than the time they were issued by the processor, and possibly not even in the order in which they were issued, which may easily compromise the system's ability to recover to a consistent state upon a failure [Bhandari et al. 2012]. This complexity, which, for concurrent programs, comes on top of the complexity of the memory consistency model, results in counterintuitive behaviors, and makes the programming on such machines very challenging.

As history has shown for consistency models in multicore systems, having formal semantics of the underlying persistency model is a paramount precondition for understanding such intricate systems, as well as for programming and reasoning about programs under such systems, and for mapping (i.e., compiling) from one model to another.

The starting point for this paper is the recent work of Raad et al. [2020] that in extensive collaboration with engineers at Intel formalized an extension of the x86-TSO memory model of Owens et al. [2009] to account for Intel-x86 persistency semantics [Intel 2019]. Roughly speaking, in order to formally justify certain outcomes that are possible after crash but can never be observed

Authors' addresses: Artem Khyzha, Tel Aviv University, Israel, artkhyzha@mail.tau.ac.il; Ori Lahav, Tel Aviv University, Israel, orilahav@tau.ac.il.

2021. 2475-1421/2021/1-ART1 $15.00
https://doi.org/

in normal (non-crashing) executions, their model, called Px86, employs two levels of buffers—per thread store buffers and a global persistence buffer sitting between the store buffers and the non-volatile memory.

There are, however, significant gaps between the Px86 model and developers and researchers' common (often informal) understanding of persistent memory systems.

**First,** Px86's explicit persist instructions are *"asynchronous"*. These are instructions that allow different levels of control over how writes persist (i.e., propagate to the non-volatile memory): *flush* instructions for persisting single cache lines and more efficient *flush-optimal* instructions that require a following store fence (*sfence*) to ensure their completion. In Px86 these instructions are asynchronous: propagating these instructions from the store buffer (making them globally visible) does not block until certain writes persist, but rather enforces restrictions on the order in which writes persist. For example, rather then guaranteeing that a certain cache line has to persist when flush is propagated from the store buffer, it only ensures that prior writes to that cache line must persist before any subsequent writes (under some appropriate definition of "prior" and "subsequent"). Similarly, Px86's sfence instructions provide such guarantees for flush-optimal instructions executed before the sfence, but does not ensure that any cache line actually persisted. In fact, for any program under Px86, it is always possible that writes do not persist at all—the system may always crash with the contents of the very initial non-volatile memory.

We observe that Px86's asynchronous explicit persist instructions lie in sharp contrast with a variety of previous work and developers' guides, ranging from theory to practice, that assumed, sometimes implicitly, *"synchronous"* explicit persist instructions that allow the programmer to assert that certain write must have persisted at certain program points (e.g., [Arulraj et al. 2018; Chen and Jin 2015; David et al. 2018; Friedman et al. 2020, 2018; Gogte et al. 2018; Izraelevitz et al. 2016b; Kolli et al. 2017, 2016; Lersch et al. 2019; Liu et al. 2020; Oukid et al. 2016; Scargall 2020; Venkataraman et al. 2011; Wang et al. 2018; Yang et al. 2015; Zuriel et al. 2019]). For example, Izraelevitz et al. [2016b]'s psync instruction blocks until all previous explicit persist institutions "have actually reached persistent memory", but such instruction cannot be implemented in Px86.

**Second,** the store buffers of Px86 are not standard first-in-first-out (FIFO) buffers. In addition to pending writes, as in usual TSO store buffers, store buffers of Px86 include pending explicit persist instructions. While pending writes preserve their order in the store buffers, the order involving the pending persist instructions is not necessarily maintained. For example, a pending flush-optimal instruction may propagate from the store buffer after a pending write also in case that the flush-optimal instruction was issued by the processor *before* the write. Indeed, without this (and similar) out-of-order propagation steps, Px86 becomes too strong so it forbids certain observable behaviors. We find the exact conditions on the store buffers propagation order to be rather intricate, making manual reasoning about possible outcomes rather cumbersome.

**Third,** Px86 lacks a formal connection to an SC-based model. Developers often prefer sequentially consistent concurrency semantics (SC). They may trust a compiler to place sufficient (preferably not excessive) barriers for ensuring SC when programming against an underlying relaxed memory model, or rely on a data-race-freedom guarantee (DRF) ensuring that well synchronized programs cannot expose weak memory behaviors. However, it is unclear how to derive a simpler well-behaved SC persistency model from Px86. The straightforward solution of discarding the store buffers from the model, thus creating direct links between the processors and the persistence buffer, is senseless for Px86. Indeed, if applied to Px86, it would result in an overly strong semantics, which, in particular, completely identifies the two kinds of explicit persist instructions ("flush" and "flush-optimal"), since the difference between them in Px86 emerges solely from propagation restrictions from the store buffers. In fact, in Px86, even certain behaviors of *single threaded* programs can be only accounted for by the effect of the store buffer.

*Does this mean that the data structures, algorithms, and principled approaches developed before having the formal* Px86 *model are futile w.r.t.* Px86*?* The main goal of the current paper is to bridge the gap between Px86 and developers and researchers' common understanding, and establish a negative answer to this question.

Our first contribution is an alternative x86-TSO operational persistency model that is provably equivalent to Px86, and is closer, to the best of our understanding, to developers' mental model of x86 persistency. Our model, which we call $\mathrm{PTSO}_{\mathrm{syn}}$, has *synchronous* explicit persist instructions, which, when they are propagated from the store buffer, do block the execution until certain writes persist. (In the case of flush-optimal, the subsequent sfence instruction is the one blocking.) Out-of-order propagation from the store buffers is also significantly confined in our $\mathrm{PTSO}_{\mathrm{syn}}$ model (but not avoided altogether, see Ex. 4.3). In addition, $\mathrm{PTSO}_{\mathrm{syn}}$ employs per-cache-line persistence FIFO buffers, which, we believe, are reflecting the guarantees on the persistence order of writes more directly than the persistence (non-FIFO) buffer of Px86. (This is not a mere technicality, due to the way explicit persist instructions are handled in Px86, its persistence buffer has to include pending writes of all cache-lines.)

The equivalence notion we use to relate Px86 and $\mathrm{PTSO}_{\mathrm{syn}}$ is state-based: it deems two models equivalent if the set of reachable program states (possibly with crashes) in the models coincide. Since a program may always start by inspecting the memory, this equivalence notion is sufficiently strong to ensure that every content of the non-volatile memory after a crash that is observable in one model is also observable in the other. Roughly speaking, our equivalence argument builds on the intuition that crashing before an asynchronous flush instruction completes is observationally indistinguishable from crashing before a synchronous flush instruction propagates from the store buffer. Making this intuition into a proof and applying it for the full model including both kinds of explicit persist instructions is technically challenging (we use two additional intermediate systems between Px86 and $\mathrm{PTSO}_{\mathrm{syn}}$).

Our second contribution is an SC persistency model that is formally related to our TSO persistency model. The SC model, which we call PSC, is naturally obtained by discarding the store buffers in $\mathrm{PTSO}_{\mathrm{syn}}$. Unlike for Px86, the resulting model, to our best understanding, precisely captures the developers' understanding. In particular, the difficulties described above for Px86 are addressed by $\mathrm{PTSO}_{\mathrm{syn}}$: even without store buffers the different kinds of explicit persist instructions (flush and flush-optimal) have different semantics in $\mathrm{PTSO}_{\mathrm{syn}}$, and store buffers are never needed in single threaded programs.

We establish two results relating PSC and $\mathrm{PTSO}_{\mathrm{syn}}$. The first is a sound mapping from PSC to $\mathrm{PTSO}_{\mathrm{syn}}$, intended to be used as a compilation scheme that ensures simpler and more well-behaved semantics on x86 machines. This mapping extends the standard mapping of SC to TSO: in addition to placing a memory fence (mfence) between writes and subsequent reads to different locations, it also places store fences (sfence) between writes and subsequent flush-optimal instructions to different locations (the latter is only required when there is no intervening write or read operation between the write and the flush-optimal, thus allowing a barrier-free compilation of standard uses of flush-optimal). The second result is a DRF-guarantee for $\mathrm{PTSO}_{\mathrm{syn}}$ w.r.t. PSC. This guarantee ensures PSC-semantics for programs that are race-free *under* PSC *semantics*, and thus provide a safe programming discipline against $\mathrm{PTSO}_{\mathrm{syn}}$ that can be followed without even knowing $\mathrm{PTSO}_{\mathrm{syn}}$. To achieve this, the standard notion of a data race is extend to include races between flush-optimal instructions and writes. We note that following our precise definition of a data race, RMW (atomic read-modify-writes) instructions do not induce races, so that with a standard lock implementation, properly locked programs (using locks to avoid data races) are not considered racy. In fact, both of the mapping of PSC to $\mathrm{PTSO}_{\mathrm{syn}}$ and the DRF-guarantee are corollaries of a stronger and more precise theorem relating PSC and $\mathrm{PTSO}_{\mathrm{syn}}$ (see Thm. 7.8).

Finally, as a by-product of our work, we provide declarative (a.k.a. axiomatic) formulations for $\text{PTSO}_{\text{syn}}$ and PSC (which we have used for formally relating them). Our $\text{PTSO}_{\text{syn}}$ declarative model is more abstract than one in [Raad et al. 2020]. In particular, its execution graphs do not record total persistence order on so-called "durable" events (the 'non-volatile-order' of [Raad et al. 2020]). Instead, execution graphs are accompanied a mapping that assigns to every location the latest persisted write to that location. From that mapping, we derive an additional partial order on events that is used in our acyclicity consistency constraints. We believe that, by avoiding the existential quantification on all possible persistence orders, our declarative presentation of the persistency model may lend itself more easily to automatic verification using execution graphs, e.g., in the style of [Abdulla et al. 2018; Kokologiannakis et al. 2017].

*Outline.* The rest of this paper is organized as follows. In §2 we present our general formal framework for operational persistency models. In §3 we present Raad et al. [2020]'s Px86 persistency model. In §4 we introduce $\text{PTSO}_{\text{syn}}$ and outline the proof of equivalence of $\text{PTSO}_{\text{syn}}$ and Px86. In §5 we present our declarative formulation of $\text{PTSO}_{\text{syn}}$ and relate it to the operational semantics. In §6 we present the persistency SC-model derived from $\text{PTSO}_{\text{syn}}$, as well as its declarative formulation. In §7 we use the declarative semantics to formally relate Px86 and $\text{PTSO}_{\text{syn}}$. In §8 we present the related work and conclude.

*Additional Material.* Proofs of the theorems in this paper are given in the technical appendix available at [Khyzha and Lahav 2020].

## 2 AN OPERATIONAL FRAMEWORK FOR PERSISTENCY SPECIFICATIONS

In this section we present our general framework for defining operational persistency models. As standard in weak memory semantics, the operational semantics is obtained by synchronizing a program (a.k.a. *thread subsystem*) and a memory subsystem (a.k.a. *storage subsystem*). The novelty lies in the definition of *persistent* memory subsystems whose states have distinguished non-volatile components. When running a program under a persistent memory subsystem, we include non-deterministic "full system" crash transitions that initialize all *volatile* parts of the state.

We start with some notational preliminaries (§2.1), briefly discuss program semantics (§2.2), and then define persistent memory subsystems and their synchronization with programs (§2.3).

### 2.1 Preliminaries

*Sequences.* For a finite alphabet $\Sigma$, we denote by $\Sigma^*$ (respectively, $\Sigma^+$) the set of all sequences (non-empty sequences) over $\Sigma$. We use $\epsilon$ to denote the empty sequence. The length of a sequence $s$ is denoted by $|s|$ (in particular $|\epsilon| = 0$). We often identify a sequence $s$ over $\Sigma$ with its underlying function in $\{1, \dots, |s|\} \to \Sigma$, and write $s(k)$ for the symbol at position $1 \leq k \leq |s|$ in $s$. We write $\sigma \in s$ if $\sigma$ appears in $s$, that is if $s(k) = \sigma$ for some $1 \leq k \leq |s|$. We use "$\cdot$" for the concatenation of sequences, which is lifted to concatenation of sets of sequences in the obvious way. We identify symbols with sequences of length 1 or their singletons when needed (e.g., in expressions like $\sigma \cdot S$).

*Relations.* Given a relation $R$, $dom(R)$ denotes its domain; and $R^?$, $R^+$, and $R^*$ denote its reflexive, transitive, and reflexive-transitive closures. The inverse of $R$ is denoted by $R^{-1}$. The (left) composition of relations $R_1, R_2$ is denoted by $R_1 \mathbin{;} R_2$. We assume that ; binds tighter than $\cup$ and $\setminus$. We denote by $[A]$ the identity relation on a set $A$, and so $[A] \mathbin{;} R \mathbin{;} [B] = R \cap (A \times B)$.

*Labeled transition systems.* A *labeled transition system* (LTS) $A$ is a tuple $\langle Q, \Sigma, Q_{\text{Init}}, T \rangle$, where $Q$ is a set of *states*, $\Sigma$ is a finite *alphabet* (whose symbols are called *transition labels*), $Q_{\text{Init}} \subseteq Q$ is a set of *initial states*, and $T \subseteq Q \times \Sigma \times Q$ is a set of *transitions*. We denote by $A.\mathsf{Q}$, $A.\Sigma$, $A.\mathsf{Q}_{\text{Init}}$, and $A.\mathsf{T}$

the components of an LTS $A$. We write $\xrightarrow{\sigma}_A$ for the relation $\{\langle q, q'\rangle \mid \langle q, \sigma, q'\rangle \in A.\mathsf{T}\}$, and $\rightarrow_A$ for $\bigcup_{\sigma\in\Sigma} \xrightarrow{\sigma}_A$ . For a sequence $t \in A.\Sigma^*$, we write $\xrightarrow{t}_A$ for the composition $\xrightarrow{t(1)}_A$ ; ... ; $\xrightarrow{t(|t|)}_A$ . A sequence $t \in A.\Sigma^*$ such that $q_{\mathsf{Init}} \xrightarrow{t}_A q$ for some $q_{\mathsf{Init}} \in A.\mathsf{Q}_{\mathsf{Init}}$ and $q \in A.\mathsf{Q}$ is called a *trace* of $A$ (or an $A$-*trace*). We denote by $\mathrm{traces}(A)$ the set of all traces of $A$. A state $q \in A.\mathsf{Q}$ is called *reachable* in $A$ if $q_{\mathsf{Init}} \xrightarrow{t}_A q$ for some $q_{\mathsf{Init}} \in A.\mathsf{Q}_{\mathsf{Init}}$ and $t \in \mathrm{traces}(A)$.

*Observable traces.* Given an LTS $A$, we usually have a distinguished symbol $\epsilon$ included in $A.\Sigma$. We refer to transitions labeled with $\epsilon$ as *silent* transitions, while the other transition are called *observable* transitions. For a sequence $t \in (A.\Sigma \setminus \{\epsilon\})^*$, we write $\overset{t}{\Rightarrow}_A$ for the relation $\{\langle q, q'\rangle \mid q \xrightarrow{\epsilon}{}^*_A \xrightarrow{t(1)}_A \xrightarrow{\epsilon}{}^*_A \cdots \xrightarrow{\epsilon}{}^*_A \xrightarrow{t(|t|)}_A \xrightarrow{\epsilon}{}^*_A q'\}$. A sequence $t \in (A.\Sigma \setminus \{\epsilon\})^*$ such that $q_{\mathsf{Init}} \overset{t}{\Rightarrow}_A q$ for some $q_{\mathsf{Init}} \in A.\mathsf{Q}_{\mathsf{Init}}$ and $q \in A.\mathsf{Q}$ is called an *observable trace* of $A$ (or an $A$-*observable-trace*). We denote by $\mathrm{otraces}(A)$ the set of all observable traces of $A$.

## 2.2 Concurrent Programs Representation

To keep the presentation abstract, we do not provide here a concrete programming language, but rather represent programs as LTSs. For this matter, we let $\mathsf{Val} \subseteq \mathbb{N}$, $\mathsf{Loc} \subseteq \{\mathtt{x}, \mathtt{y}, ...\}$, and $\mathsf{Tid} \subseteq \{\mathtt{T}_1, \mathtt{T}_2, ..., \mathtt{T}_N\}$, be sets of *values*, (shared) memory *locations*, and *thread identifiers*. We assume that Val contains a distinguished value 0, used as the initial value for all locations.

Sequential programs are identified with LTSs whose transition labels are *event labels*, extended with $\epsilon$ for silent program transitions, as defined next.[1]

*Definition 2.1.* An *event label* is either a *read label* $\mathtt{R}(x, v_{\mathtt{R}})$, a *write label* $\mathtt{W}(x, v_{\mathtt{W}})$, a *read-modify-write (RMW) label* $\mathtt{RMW}(x, v_{\mathtt{R}}, v_{\mathtt{W}})$, a *failed compare-and-swap (CAS) label* $\mathtt{R\text{-}ex}(x, v_{\mathtt{R}})$, an *mfence label* MF, a *flush label* $\mathtt{FL}(x)$, a *flush-opt label* $\mathtt{FO}(x)$, or an *sfence label* SF, where $x \in \mathsf{Loc}$ and $v_{\mathtt{R}}, v_{\mathtt{W}} \in \mathsf{Val}$. We denote by Lab the set of all event labels. The functions typ, loc, $\mathtt{val}_{\mathtt{R}}$, and $\mathtt{val}_{\mathtt{W}}$ retrieve (when applicable) the type (R/W/RMW/R-ex/MF/FL/FO/SF), location ($x$), read value ($v_{\mathtt{R}}$), and written value ($v_{\mathtt{W}}$) of an event label.

Event labels correspond to the different interactions that a program may have with the persistent memory subsystem. In particular, we have several types of barrier labels: a memory fence (MF), a persistency per-location flush barrier ($\mathtt{FL}(x)$), an optimized persistency per-location flush barrier, called "flush-optimal" ($\mathtt{FO}(x)$), and a store fence (SF).[2] Roughly speaking, memory fences (MF) ensure the completion of all prior instructions, while store fences (SF) ensure that prior flush-optimal instructions have taken their effect. Memory access labels include plain reads and writes, as well as RMWs ($\mathtt{RMW}(x, v_{\mathtt{R}}, v_{\mathtt{W}})$) resulting from operations like compare-and-swap (CAS) and fetch-and-add. For failed CAS (a CAS that did not read the expected value) we use a special read label $\mathtt{R\text{-}ex}(x, v_{\mathtt{R}})$, which allows us to distinguish such transitions from plain reads and provide them with stronger semantics.[3] We note that our event labels are specific for the x86 persistency, but they can be easily extended and adapted for other models.

In turn, a (concurrent) program *Pr* is a top-level parallel composition of sequential programs, defined as a mapping assigning a sequential program to every $\tau \in \mathsf{Tid}$. A program *Pr* is also identified with an LTS, which is obtained by standard lifting of the LTSs representing its component

[1] In our examples we use a standard program syntax and assume a standard reading of programs as LTSs. To assist the reader, Appendix H provides a concrete example of how this can be done.

[2] In [Intel 2019], flush is referred to as CLFLUSH, flush-optimal is referred to as CLFLUSHOPT. Intel's CLWB instruction is equivalent to CLFLUSHOPT and may improve performance in certain cases [Raad et al. 2020].

[3] Some previous work, e.g., [Lahav et al. 2016; Raad et al. 2020], consider failed RMWs (arising from `lock cmpxchg` instructions) as plain reads, although failed RMWs induce a memory fence in TSO.

sequential programs. The transition labels of this LTS record the thread identifier of non-silent transitions, as defined next.

*Definition 2.2.* A *program transition label* is either $\langle \tau, l \rangle$ for $\tau \in \mathsf{Tid}$ and $l \in \mathsf{Lab}$ (*observable transition*) or $\epsilon$ (*silent transition*). We denote by $\mathsf{PTLab}$ the set of all program transition labels. We use the function `tid` and `lab` to return the thread identifier ($\tau$) and event label $l$ of a given transition label (when applicable). The functions `typ`, `loc`, $\mathtt{val}_{\mathsf{R}}$, and $\mathtt{val}_{\mathsf{W}}$ are lifted to transition labels in the obvious way (undefined for $\epsilon$-transitions).

The LTS induced by a (concurrent) program $Pr$ is over the alphabet $\mathsf{PTLab}$; its states are functions, denoted by $\overline{q}$, assigning a state in $Pr(\tau).\mathsf{Q}$ to every $\tau \in \mathsf{Tid}$; its initial states set is $\prod_\tau Pr(\tau).\mathsf{Q}_{\mathsf{Init}}$; and its transitions are "interleaved transitions" of $Pr$'s components, given by:

$$\frac{l \in \mathsf{Lab} \qquad \overline{q}(\tau) \xrightarrow{l}_{Pr(\tau)} q'}{\overline{q} \xrightarrow{\tau, l}_{Pr} \overline{q}[\tau \mapsto q']} \qquad\qquad \frac{\overline{q}(\tau) \xrightarrow{\epsilon}_{Pr(\tau)} q'}{\overline{q} \xrightarrow{\epsilon}_{Pr} \overline{q}[\tau \mapsto q']}$$

We refer to sequences over $\mathsf{PTLab} \setminus \{\epsilon\} = \mathsf{Tid} \times \mathsf{Lab}$ as *observable program traces.* Clearly, observable program traces are closed under "per-thread prefixes":

*Definition 2.3.* We denote by $t|_\tau$ the restriction of an observable program trace $t$ to transition labels of the form $\langle \tau, \_ \rangle$. An observable program trace $t'$ is *per-thread equivalent* to an observable program trace $t$, denoted by $t' \sim t$, if $t'|_\tau = t|_\tau$ for every $\tau \in \mathsf{Tid}$. In turn, $t'$ is a *per-thread prefix* of $t$, denoted by $t' \lesssim t$, if $t'$ is a (possibly trivial) prefix of some $t'' \sim t$ (equivalently, $t'|_\tau$ is a prefix of $t|_\tau$ for every $\tau \in \mathsf{Tid}$).

Proposition 2.4. *If $t$ is a $Pr$-observable-trace, then so is every $t' \lesssim t$.*

## 2.3 Persistent Systems

At the program level, the read values are arbitrary. It is the responsibility of the memory subsystem to specify what values can be read from each location at each point. Formally, the memory subsystem is another LTS over $\mathsf{PTLab}$, whose synchronization with the program gives us the possible behaviors of the whole system. For persistent memory subsystems, we require that each memory state is composed of a persistent memory $\mathsf{Loc} \to \mathsf{Val}$, which survived the crash, and a volatile part, whose exact structure varies from one system to another (e.g., TSO-based models will have store buffers in the volatile part and SC-based systems will not).

*Definition 2.5.* A *persistent memory subsystem* is an LTS $M$ that satisfies the following:

- $M.\Sigma = \mathsf{PTLab}$.
- $M.\mathsf{Q} = (\mathsf{Loc} \to \mathsf{Val}) \times \tilde{Q}$ where $\tilde{Q}$ is some set. We denote by $M.\tilde{\mathsf{Q}}$ the particular set $\tilde{Q}$ used in a persistent memory subsystem $M$. We usually denote states in $M.\mathsf{Q}$ as $q = \langle m, \tilde{m} \rangle$, where the two components ($m$ and $\tilde{m}$) of a state $q$ are respectively called the *non-volatile state* and the *volatile state.*[4]
- $M.\mathsf{Q}_{\mathsf{Init}} = (\mathsf{Loc} \to \mathsf{Val}) \times \tilde{Q}_{\mathsf{Init}}$ where $\tilde{Q}_{\mathsf{Init}}$ is some subset of $M.\tilde{\mathsf{Q}}$. We denote by $M.\tilde{\mathsf{Q}}_{\mathsf{Init}}$ the particular set $\tilde{Q}_{\mathsf{Init}}$ used in a persistent memory subsystem $M$.

In the systems defined below, the non-volatile states in $M.\tilde{\mathsf{Q}}$ consists a multiple buffers (store buffers and persistence buffers) that lose their contents upon crash. The transition labels of a persistent memory subsystem are pairs in $\mathsf{Tid} \times \mathsf{Lab}$, representing the thread identifier and the event label of the operation, or $\epsilon$ for internal (silent) memory actions (e.g., propagation from the

[4] When the elements of $M.\tilde{\mathsf{Q}}$ are tuples themselves, we often simplify the writing by flattening the states, e.g., $\langle m, \alpha, \beta \rangle$ instead of $\langle m, \langle \alpha, \beta \rangle \rangle$.

store buffers). We note that, given the requirements of Def. 2.5, to define a persistent memory subsystem $M$ it suffices to give its sets $M.\tilde{\mathsf{Q}}$ and $M.\tilde{\mathsf{Q}}_{\mathsf{Init}}$ of volatile states and initial volatile states, and its transition relation.

By synchronizing a program $Pr$ and a persistent memory subsystem $M$, and including non-deterministic crash transitions (labeled with $\lightning$), we obtain a *persistent system*, which we denote by $Pr \shortparallel M$:

*Definition 2.6.* A program $Pr$ and a persistent memory subsystem $M$ form a *persistent system*, denoted by $Pr \shortparallel M$. It is an LTS over the alphabet $\mathsf{PTLab} \cup \{\lightning\}$ whose set of states is $Pr.\mathsf{Q} \times (\mathsf{Loc} \to \mathsf{Val}) \times M.\tilde{\mathsf{Q}}$; its initial states set is $Pr.\mathsf{Q}_{\mathsf{Init}} \times \{m_{\mathsf{Init}}\} \times M.\tilde{\mathsf{Q}}_{\mathsf{Init}}$, where $m_{\mathsf{Init}} = \lambda x \in \mathsf{Loc}.\ 0$; and its transitions are "synchronized transitions" of $Pr$ and $M$, given by:

$$\frac{\overline{q} \xrightarrow{\tau,l}_{Pr} \overline{q}' \qquad \langle m, \tilde{m}\rangle \xrightarrow{\tau,l}_{M} \langle m', \tilde{m}'\rangle}{\langle \overline{q}, m, \tilde{m}\rangle \xrightarrow{\tau,l}_{Pr \shortparallel M} \langle \overline{q}', m', \tilde{m}'\rangle} \qquad \frac{\overline{q} \xrightarrow{\epsilon}_{Pr} \overline{q}'}{\langle \overline{q}, m, \tilde{m}\rangle \xrightarrow{\epsilon}_{Pr \shortparallel M} \langle \overline{q}', m, \tilde{m}\rangle}$$

$$\frac{\langle m, \tilde{m}\rangle \xrightarrow{\epsilon}_{M} \langle m', \tilde{m}'\rangle}{\langle \overline{q}, m, \tilde{m}\rangle \xrightarrow{\epsilon}_{Pr \shortparallel M} \langle \overline{q}, m', \tilde{m}'\rangle} \qquad \frac{\overline{q}_{\mathsf{Init}} \in Pr.\mathsf{Q}_{\mathsf{Init}} \qquad \tilde{m}_{\mathsf{Init}} \in M.\tilde{\mathsf{Q}}_{\mathsf{Init}}}{\langle \overline{q}, m, \tilde{m}\rangle \xrightarrow{\lightning}_{Pr \shortparallel M} \langle \overline{q}_{\mathsf{Init}}, m, \tilde{m}_{\mathsf{Init}}\rangle}$$

Crash transitions reinitialize the program state $\overline{q}$ (which corresponds to losing the program counter and the local stores) and the volatile component of the memory state $\tilde{m}$. The persistent memory $m$ is left intact.

Given the above definition of persistent system, we can define the set of reachable program states under a given persistent memory subsystem. Focused on safety properties, we use this notion to define when one persistent memory subsystem observationally refines another.

*Definition 2.7.* A program state $\overline{q} \in Pr.\mathsf{Q}$ is *reachable under a persistent memory subsystem $M$* if $\langle \overline{q}, m, \tilde{m}\rangle$ is reachable in $Pr \shortparallel M$ for some $\langle m, \tilde{m}\rangle \in M.\mathsf{Q}$.

*Definition 2.8.* A persistent memory subsystem $M_1$ *observationally refines* a persistent memory subsystem $M_2$ if for every program $Pr$, every program state $\overline{q} \in Pr.\mathsf{Q}$ that is reachable under $M_1$ is also reachable under $M_2$. We say that $M_1$ and $M_2$ are *observationally equivalent* if $M_1$ observationally refines $M_2$ and $M_2$ observationally refines $M_1$.

While the above refinement notion refers to reachable program states, it is also applicable for the reachable non-volatile memories. Indeed, a program may always start by asserting certain conditions reflecting the fact that the memory is in certain consistent state (which usually vacuously hold for the very initial memory $m_{\mathsf{Init}}$), thus capturing the state of the non-volatile memory in the program state itself.

*Remark 1.* Our notions of observational refinement and equivalence above are state-based. This is standard in formalizations of weak memory models, intended to support reasoning about safety properties (e.g., detect program assertion violations). In particular, if $M_1$ observationally refines $M_2$, the developer may safely assume $M_2$'s semantics when reasoning about reachable non-volatile memories under $M_1$. We note that a more refined notion of observation in a richer language, e.g., with I/O side-effects, may expose behaviors of $M_1$ that are not observable in $M_2$ even when $M_1$ and $M_2$ are observationally equivalent according to the definition above.

The following lemma allows us to establish refinements without considering *all programs* and *crashes*.

*Definition 2.9.* An observable trace $t$ of a persistent memory subsystem $M$ is called *$m_0$-to-$m$* if $\langle m_0, \tilde{m}_{\mathsf{Init}} \rangle \overset{t}{\Rightarrow}_M \langle m, \tilde{m} \rangle$ for some $\tilde{m}_{\mathsf{Init}} \in M.\tilde{\mathsf{Q}}_{\mathsf{Init}}$ and $\tilde{m} \in M.\tilde{\mathsf{Q}}$. Furthermore, $t$ is called *$m_0$-initialized* if it is $m_0$-to-$m$ for some $m$.

Lemma 2.10. *The following conditions together ensure that a persistent memory subsystem $M_1$ observationally refines a persistent memory subsystem $M_2$:*

*(i) Every $m_0$-initialized $M_1$-observable-trace is also an $m_0$-initialized $M_2$-observable-trace.*
*(ii) For every $m_0$-to-$m$ $M_1$-observable-trace $t_1$, some $t_2 \lesssim t_1$ is an $m_0$-to-$m$ $M_2$-observable-trace.*

Proof (outline). Consider any program state $\overline{q}$ reachable under $M_1$ with a trace $t = t_0 \cdot \lightning \cdot t_1 \cdot \ldots \cdot \lightning \cdot t_n$. Each crash resets the program state and the volatile state, but not the non-volatile state. We leverage condition (ii) in showing that $Pr \parallel M_2$ can reach each crash having the same non-volatile memory state as $Pr \parallel M_1$ (possibly with a shorter program trace). Therefore, when $Pr \parallel M_1$ proceeds with in $t_n$ after the last crash, $Pr \parallel M_2$ is able to proceed from exactly the same state. Then, condition (i) applied to $t_n$ immediately gives us that $\overline{q}$ is reachable under $M_2$. □

Intuitively speaking, condition (i) ensures that after the last system crash, the client can only observe behaviors of $M_1$ that are allowed by $M_2$, and condition (ii) ensures that the parts of the state that survives crashes that are observable in $M_1$ are also observable in $M_2$. Note that condition (ii) allows us (and we actually rely on it in our proofs) to reach the non-volatile memory in $M_1$ with a per-thread prefix of the program trace that reached that memory in $M_2$. Indeed, the program state is lost after the crash, and the client cannot observe what part of the program has been actually executed before the crash.

## 3 THE Px86 PERSISTENT MEMORY SUBSYSTEM

In this section we present Px86, the persistent memory subsystem by Raad et al. [2020] which models the persistency semantics of the Intel-x86 architecture.

*Remark 2.* Following discussions with Intel engineers, Raad et al. [2020] introduced *two* models: Px86$_{\text{man}}$ and Px86$_{\text{sim}}$. The first formalizes the (ambiguous and under specified) reference manual specification [Intel 2019]. The latter simplifies and strengthens the first while capturing the "behavior intended by the Intel engineers". The model studied here is Px86$_{\text{sim}}$, which we simply call Px86.

Px86 is an extension of the standard TSO model [Owens et al. 2009] with another layer called *persistence buffer*. This is a global buffer that contains writes that are pending to be persisted to the (non-volatile) memory as well as certain markers governing the persistence order. Store buffers are extended to include not only store instruction but also flush and sfence instructions. Both the (per-thread) store buffers and the (global) persistence buffer are volatile.

*Definition 3.1.* A *store buffer* is a finite sequence $b$ of event labels $l$ with $\mathtt{typ}(l) \in \{\mathtt{W}, \mathtt{FL}, \mathtt{FO}, \mathtt{SF}\}$. A *store-buffer mapping* is a function $B$ assigning a store buffer to every $\tau \in \mathsf{Tid}$. We denote by $B_\epsilon$, the initial store-buffer mapping assigning the empty sequence to every $\tau \in \mathsf{Tid}$.

*Definition 3.2.* A *persistence buffer* is a finite sequence $p$ of elements of the form $\mathtt{W}(x, v)$ or $\mathtt{PER}(x)$ (where $x \in \mathsf{Loc}$ and $v \in \mathsf{Val}$).

Like the memory, the persistence buffer is accessible by all threads. When thread $\tau$ reads from a shared location $x$ it obtains its latest accessible value of $x$, which is defined using the following get function applied on the current persistent memory $m$, persistence buffer $p$, and $\tau$'s store buffer $b$:

$$\mathsf{get}(m, p, b) \triangleq \lambda x.\ \begin{cases} v & b = b_1 \cdot \mathtt{W}(x, v) \cdot b_2 \wedge \mathtt{W}(x, \_) \notin b_2 \\ v & \mathtt{W}(x, \_) \notin b \wedge p = p_1 \cdot \mathtt{W}(x, v) \cdot p_2 \wedge \mathtt{W}(x, \_) \notin p_2 \\ m(x) & \text{otherwise} \end{cases}$$

$$m \in \mathrm{Loc} \to \mathrm{Val} \qquad p \in (\{\mathtt{W}(x,v) \mid x \in \mathrm{Loc}, v \in \mathrm{Val}\} \cup \{\mathtt{PER}(x) \mid x \in \mathrm{Loc}\})^*$$
$$B \in \mathrm{Tid} \to (\{\mathtt{W}(x,v) \mid x \in \mathrm{Loc}, v \in \mathrm{Val}\} \cup \{\mathtt{FL}(x) \mid x \in \mathrm{Loc}\} \cup \{\mathtt{FO}(x) \mid x \in \mathrm{Loc}\} \cup \{\mathtt{SF}\})^*$$
$$p_{\mathsf{Init}} \triangleq \epsilon \qquad B_{\mathsf{Init}} \triangleq \lambda\tau.\ \epsilon$$

WRITE/FLUSH/FLUSH-OPT/SFENCE
$$\dfrac{\begin{array}{c}\mathtt{typ}(l) \in \{\mathtt{W}, \mathtt{FL}, \mathtt{FO}, \mathtt{SF}\} \\ B' = B[\tau \mapsto B(\tau) \cdot l]\end{array}}{\langle m, p, B\rangle \xrightarrow{\tau,l}_{\mathrm{Px86}} \langle m, p, B'\rangle}$$

READ
$$\dfrac{\begin{array}{c}l = \mathtt{R}(x,v) \\ \mathtt{get}(m, p, B(\tau))(x) = v\end{array}}{\langle m, p, B\rangle \xrightarrow{\tau,l}_{\mathrm{Px86}} \langle m, p, B\rangle}$$

RMW
$$\dfrac{\begin{array}{c}l = \mathtt{RMW}(x, v_{\mathsf{R}}, v_{\mathsf{W}}) \\ \mathtt{get}(m, p, \epsilon)(x) = v_{\mathsf{R}} \\ B(\tau) = \epsilon \\ p' = p \cdot \mathtt{W}(x, v_{\mathsf{W}})\end{array}}{\langle m, p, B\rangle \xrightarrow{\tau,l}_{\mathrm{Px86}} \langle m, p', B\rangle}$$

RMW-FAIL
$$\dfrac{\begin{array}{c}l = \mathtt{R\text{-}ex}(x,v) \\ \mathtt{get}(m, p, \epsilon)(x) = v \\ B(\tau) = \epsilon\end{array}}{\langle m, p, B\rangle \xrightarrow{\tau,l}_{\mathrm{Px86}} \langle m, p, B\rangle}$$

MFENCE
$$\dfrac{\begin{array}{c}l = \mathtt{MF} \\ B(\tau) = \epsilon\end{array}}{\langle m, p, B\rangle \xrightarrow{\tau,l}_{\mathrm{Px86}} \langle m, p, B\rangle}$$

PROP-W
$$\dfrac{\begin{array}{c}B(\tau) = b_1 \cdot \mathtt{W}(x,v) \cdot b_2 \\ \mathtt{W}(\_,\_), \mathtt{FL}(\_), \mathtt{SF} \notin b_1 \\ B' = B[\tau \mapsto b_1 \cdot b_2] \qquad p' = p \cdot \mathtt{W}(x,v)\end{array}}{\langle m, p, B\rangle \xrightarrow{\epsilon}_{\mathrm{Px86}} \langle m, p', B'\rangle}$$

PROP-FL
$$\dfrac{\begin{array}{c}B(\tau) = b_1 \cdot \mathtt{FL}(x) \cdot b_2 \\ \mathtt{W}(\_,\_), \mathtt{FL}(\_), \mathtt{FO}(x), \mathtt{SF} \notin b_1 \\ B' = B[\tau \mapsto b_1 \cdot b_2] \qquad p' = p \cdot \mathtt{PER}(x)\end{array}}{\langle m, p, B\rangle \xrightarrow{\epsilon}_{\mathrm{Px86}} \langle m, p', B'\rangle}$$

PROP-FO
$$\dfrac{\begin{array}{c}B(\tau) = b_1 \cdot \mathtt{FO}(x) \cdot b_2 \\ \mathtt{W}(x,\_), \mathtt{FL}(x), \mathtt{SF} \notin b_1 \\ B' = B[\tau \mapsto b_1 \cdot b_2] \qquad p' = p \cdot \mathtt{PER}(x)\end{array}}{\langle m, p, B\rangle \xrightarrow{\epsilon}_{\mathrm{Px86}} \langle m, p', B'\rangle}$$

PROP-SF
$$\dfrac{\begin{array}{c}B(\tau) = \mathtt{SF} \cdot b \\ B' = B[\tau \mapsto b]\end{array}}{\langle m, p, B\rangle \xrightarrow{\epsilon}_{\mathrm{Px86}} \langle m, p, B'\rangle}$$

PERSIST-W
$$\dfrac{\begin{array}{c}p = p_1 \cdot \mathtt{W}(x,v) \cdot p_2 \\ \mathtt{W}(x,\_), \mathtt{PER}(\_) \notin p_1 \\ p' = p_1 \cdot p_2 \qquad m' = m[x \mapsto v]\end{array}}{\langle m, p, B\rangle \xrightarrow{\epsilon}_{\mathrm{Px86}} \langle m', p', B\rangle}$$

PERSIST-PER
$$\dfrac{\begin{array}{c}p = p_1 \cdot \mathtt{PER}(x) \cdot p_2 \\ \mathtt{W}(x,\_), \mathtt{PER}(\_) \notin p_1 \\ p' = p_1 \cdot p_2\end{array}}{\langle m, p, B\rangle \xrightarrow{\epsilon}_{\mathrm{Px86}} \langle m, p', B\rangle}$$

Fig. 1. The Px86 Persistent Memory Subsystem

Using these definitions, Px86 is presented in Fig. 1. Its set of volatile states, Px86.$\tilde{\mathrm{Q}}$, consists of all pairs $\langle p, B\rangle$, where $p$ is a persistence buffer and $B$ is a store-buffer mapping. Initially, all buffers are empty (Px86.$\tilde{\mathrm{Q}}_{\mathsf{Init}} = \{\langle \epsilon, B_\epsilon\rangle\}$).

The system's transitions are of three kinds: "issuing steps", "propagation steps", and "persistence steps". Steps of the first kind are defined as in standard TSO semantics, with the only extension being the fact that flush, flush-optimals and sfences instructions emit entries in the store buffer.

Propagation of writes from the store buffer (PROP-W) is both making the writes visible to other threads, and propagating them to the persistence buffer. Note that a write may propagate even when flush-optimals precede it in the store buffer (which means that they were issued before the write by the thread). Propagation of flushes and flush-optimals (PROP-FL and PROP-FO) adds a "PER-marker" to the persistence buffer, which later restricts the order in which writes persist. The difference between the two kinds of flushes is reflected in the conditions on their propagation. In particular, a flush-optimal may propagate even when writes to different locations precede it in the store buffer (which means that they were issued before the flush-optimal by the thread). Propagation of sfences simply removes the sfence entry, which is only used to restrict the order of propagation of other entries, and is discarded once it reaches the head of the store buffer.

Finally, persisting a write moves a write entry from the persistence buffer to the non-volatile memory (PERSIST-W). Writes to the same location persist in the same order in which they propagate. The PER-markers ensure that writes that propagated before some marker persist before writes that propagate after that marker. After the PER-markers play their role, they are discarded from the persistence buffer (PERSIST-PER).

We note that the step for (non-deterministic) system crashes is included in Def. 2.6 upon synchronizing the LTS of a program with the one of the Px86 memory subsystem. *Without crashes*, the effect of the persistence buffer is unobservable, and Px86 coincides with the standard TSO semantics.

*Example 3.3.* Consider the following four sequential programs:

| | | | |
|---|---|---|---|
| | x := 1 ; | x := 1 ; | x := 1 ; |
| x := 1 ; | fl(x) ; | fo(x) ; | fo(x) ; |
| y := 1 ; | y := 1 ; | y := 1 ; | sfence ; |
| | | | y := 1 ; |
| (*A*) ✓ | (*B*) ✗ | (*C*) ✓ | (*D*) ✗ |

To refer to particular program behaviors, we use colored boxes for denoting the last write that persisted for each locations (inducing a possible content of the non-volatile memory in a run of the program). When some location lacks such annotation (like x in the above examples), it means that none of its write persisted, so that its value in the non-volatile memory is 0 (the initial value). In particular, the behaviors annotated above all have $m \supseteq \{x \mapsto 0, y \mapsto 1\}$. It is easy to verify that Px86 allows/forbids each of these behaviors as specified by the corresponding ✓/✗ marking. In particular, example (C) demonstrates that propagating a write before a prior flush-optimal is essential. Indeed, the annotated behavior is obtained by propagating y := 1 from the store buffer before fo(x) (but necessarily after x := 1). Otherwise, y := 1 cannot persist without x := 1 persisting before.

*Remark 3.* To simplify the presentation, following Izraelevitz et al. [2016a], but unlike Raad et al. [2020], we conservatively assume that writes persist atomically at the location granularity (representing, e.g., machine words). Real machines provide granularity at the width of a cache line, and, assuming the programmer can faithfully control what locations are stored on same cache line, may provide stronger guarantees. Nevertheless, adapting our results to support cache line granularity is straightforward.

*Remark 4.* Persistent systems make programs responsible for recovery from crashes: after a crash, programs restart with reinitialized program state and the volatile component of the memory state. In contrast, Raad et al. [2020] define their system assuming a separate recovery program called a *recovery context*, which after a crash atomically advances program state from the initial one. In our technical development, we prefer to make minimal assumptions about the recovery mechanism. Nevertheless, by adjusting crash transitions in Def. 2.6, our framework and results can be easily extended to support Raad et al. [2020]'s recovery context.

## 4 THE PTSO$_{\text{SYN}}$ PERSISTENT MEMORY SUBSYSTEM

In this section we present our alternative persistent memory subsystem, which we call PTSO$_{\text{syn}}$, that is observationally equivalent to Px86. We list major differences between PTSO$_{\text{syn}}$ and Px86:

- PTSO$_{\text{syn}}$ has *synchronous flush instructions*—the propagation of a flush of location $x$ from the store buffer is blocking the execution until all writes to $x$ that propagated earlier have persisted.

We note that, as expected in a TSO-based model, flushes do not take their synchronous effect when they are issued by the thread, but rather have a delayed globally visible effect happening when they propagate from the store buffer.

- $\mathsf{PTSO}_{\mathsf{syn}}$ has *synchronous sfence instructions*—the propagation of an sfence from the store buffer is blocking the execution until all flush-optimals of the same thread that propagated earlier have taken their effect. The latter means that all writes to the location of the flush-optimal that propagated before the flush-optimal have persisted. Thus, flush-optimals serve as markers in the persistence buffer, that are only meaningful when an sfence (issued by the same thread that issued the flush-optimal) propagates from the store buffer. As for flushes, the effect of an sfence is not at its issue time but at its propagation time. We note that mfence and RMW operations (both when they fail and when they succeed) induce an implicit sfence.
- Rather than a global persistence buffer, $\mathsf{PTSO}_{\mathsf{syn}}$ employs *per-location* persistence buffers directly reflecting the fact that the persistence order has to agree with the propagation order only between writes to the same location, while writes to different locations may persist out of order.
- The store buffers of $\mathsf{PTSO}_{\mathsf{syn}}$ are "almost" FIFO buffers. With the exception of flush-optimals, entries may propagate from the store buffer only when they reach the head of the buffer. Flush-optimals may still "overtake" writes as well as flushes/flush-optimals of a different location. Example 4.3 below demonstrates why we need to allow the latter (there is a certain design choice here, see Remark 5).

To formally present $\mathsf{PTSO}_{\mathsf{syn}}$, we first define per-location persistence buffers and per-location-persistence-buffer mappings.

*Definition 4.1.* A *per-location persistence buffer* is a finite sequence $p$ of elements of the form $\mathtt{W}(v)$ or $\mathtt{FO}(\tau)$ (where $v \in \mathsf{Val}$ and $\tau \in \mathsf{Tid}$). A *per-location-persistence-buffer mapping* is a function $P$ assigning a per-location persistence buffer to every $x \in \mathsf{Loc}$. We denote by $P_\epsilon$, the initial per-location-persistence-buffer mapping assigning the empty sequence to every $x \in \mathsf{Loc}$.

Flush instructions under $\mathsf{PTSO}_{\mathsf{syn}}$ take effect upon their propagation, so, unlike in Px86, they do not add $\mathsf{PER}$-markers into the persistence buffers. For flush-optimals, instead of $\mathsf{PER}$-markers, we use (per location) $\mathtt{FO}(\tau)$ markers, where $\tau$ is the identifier of the thread that issued the instruction. In accordance with how Px86's sfence only blocks the propagation of the same thread's flush-optimals, the synchronous behavior of sfence must not wait for flush-optimals by different threads (see Ex. 4.4 below).

The (overloaded) get function is updated in the obvious way:

$$\mathrm{get}(m, p, b) \triangleq \lambda x. \begin{cases} v & b = b_1 \cdot \mathtt{W}(x, v) \cdot b_2 \land \mathtt{W}(x, \_) \notin b_2 \\ v & \mathtt{W}(x, \_) \notin b \land p = p_1 \cdot \mathtt{W}(v) \cdot p_2 \land \mathtt{W}(\_) \notin p_2 \\ m(x) & \text{otherwise} \end{cases}$$

For looking up a value for location $x$ by thread $\tau$, we apply get with $m$ being the current non-volatile memory, $p$ being $x$'s persistence buffer, $b$ being $\tau$'s store buffer

Using these definitions, $\mathsf{PTSO}_{\mathsf{syn}}$ is presented in Fig. 2. Its set of volatile states, $\mathsf{PTSO}_{\mathsf{syn}}.\tilde{\mathsf{Q}}$, consists of all pairs $\langle P, B\rangle$, where $P$ is a per-location-persistence-buffer mapping and $B$ is a store-buffer mapping. Initially, all buffers are empty ($\mathsf{PTSO}_{\mathsf{syn}}.\tilde{\mathsf{Q}}_{\mathsf{Init}} = \{\langle P_\epsilon, B_\epsilon\rangle\}$).

The differences of $\mathsf{PTSO}_{\mathsf{syn}}$ w.r.t. Px86 are highlighted in Fig. 2. First, the PROP-FL transition only occurs when $P(x) = \epsilon$ to ensure that all previously propagated writes have persisted. Second, the PROP-SFENCE transition (as well as RMW, RMW-FAIL, and MFENCE) only occurs when $\forall y.\ \mathtt{FO}(\tau) \notin P(y)$ holds to ensure that propagation of each sfence blocks until previous flush-optimals of the same thread have completed. Third, the PERSIST-W and PERSIST-FO transitions persist the entries from the per-location persistence buffers *in-order*. Finally, the PROP-W and PROP-FL transitions propagate

$$m \in \mathsf{Loc} \to \mathsf{Val} \qquad P \in \mathsf{Loc} \to (\{\mathtt{W}(v) \mid v \in \mathsf{Val}\} \cup \{\mathtt{FO}(\tau) \mid \tau \in \mathsf{Tid}\})^*$$
$$B \in \mathsf{Tid} \to (\{\mathtt{W}(x,v) \mid x \in \mathsf{Loc}, v \in \mathsf{Val}\} \cup \{\mathtt{FL}(x) \mid x \in \mathsf{Loc}\} \cup \{\mathtt{FO}(x) \mid x \in \mathsf{Loc}\} \cup \{\mathtt{SF}\})^*$$
$$P_{\mathsf{Init}} \triangleq \lambda x.\ \epsilon \qquad B_{\mathsf{Init}} \triangleq \lambda \tau.\ \epsilon$$

WRITE/FLUSH/FLUSH-OPT/SFENCE

$$\frac{\begin{array}{c}\mathsf{typ}(l) \in \{\mathtt{W}, \mathtt{FL}, \mathtt{FO}, \mathtt{SF}\} \\ B' = B[\tau \mapsto B(\tau) \cdot l]\end{array}}{\langle m, P, B\rangle \xrightarrow{\tau,l}_{\mathsf{PTSO_{syn}}} \langle m, P, B'\rangle}$$

READ

$$\frac{\begin{array}{c}l = \mathtt{R}(x, v) \\ \mathsf{get}(m, P(x), B(\tau))(x) = v\end{array}}{\langle m, P, B\rangle \xrightarrow{\tau,l}_{\mathsf{PTSO_{syn}}} \langle m, P, B\rangle}$$

RMW

$$\frac{\begin{array}{c}l = \mathtt{RMW}(x, v_{\mathtt{R}}, v_{\mathtt{W}}) \\ \mathsf{get}(m, P(x), \epsilon)(x) = v_{\mathtt{R}} \\ B(\tau) = \epsilon \\ \forall y.\ \mathtt{FO}(\tau) \notin P(y) \\ P' = P[x \mapsto P(x) \cdot \mathtt{W}(v_{\mathtt{W}})]\end{array}}{\langle m, P, B\rangle \xrightarrow{\tau,l}_{\mathsf{PTSO_{syn}}} \langle m, P', B\rangle}$$

RMW-FAIL

$$\frac{\begin{array}{c}l = \mathtt{R\text{-}ex}(x, v) \\ \mathsf{get}(m, P(x), \epsilon)(x) = v \\ B(\tau) = \epsilon \\ \forall y.\ \mathtt{FO}(\tau) \notin P(y)\end{array}}{\langle m, P, B\rangle \xrightarrow{\tau,l}_{\mathsf{PTSO_{syn}}} \langle m, P, B\rangle}$$

MFENCE

$$\frac{\begin{array}{c}l = \mathtt{MF} \\ B(\tau) = \epsilon \\ \forall y.\ \mathtt{FO}(\tau) \notin P(y)\end{array}}{\langle m, P, B\rangle \xrightarrow{\tau,l}_{\mathsf{PTSO_{syn}}} \langle m, P, B\rangle}$$

PROP-W

$$\frac{\begin{array}{c}B(\tau) = \mathtt{W}(x, v) \cdot b \qquad B' = B[\tau \mapsto b] \\ P' = P[x \mapsto P(x) \cdot \mathtt{W}(v)]\end{array}}{\langle m, P, B\rangle \xrightarrow{\epsilon}_{\mathsf{PTSO_{syn}}} \langle m, P', B'\rangle}$$

PROP-FL

$$\frac{\begin{array}{c}B(\tau) = \mathtt{FL}(x) \cdot b \qquad B' = B[\tau \mapsto b] \\ P(x) = \epsilon\end{array}}{\langle m, P, B\rangle \xrightarrow{\epsilon}_{\mathsf{PTSO_{syn}}} \langle m, P, B'\rangle}$$

PROP-FO

$$\frac{\begin{array}{c}B(\tau) = b_1 \cdot \mathtt{FO}(x) \cdot b_2 \\ \mathtt{W}(x, \_), \mathtt{FL}(x), \mathtt{FO}(x), \mathtt{SF} \notin b_1 \\ B' = B[\tau \mapsto b_1 \cdot b_2] \qquad P' = P[x \mapsto P(x) \cdot \mathtt{FO}(\tau)]\end{array}}{\langle m, P, B\rangle \xrightarrow{\epsilon}_{\mathsf{PTSO_{syn}}} \langle m, P', B'\rangle}$$

PROP-SF

$$\frac{\begin{array}{c}B(\tau) = \mathtt{SF} \cdot b \qquad B' = B[\tau \mapsto b] \\ \forall y.\ \mathtt{FO}(\tau) \notin P(y)\end{array}}{\langle m, P, B\rangle \xrightarrow{\epsilon}_{\mathsf{PTSO_{syn}}} \langle m, P, B'\rangle}$$

PERSIST-W

$$\frac{\begin{array}{c}P(x) = \mathtt{W}(v) \cdot p \\ P' = P[x \mapsto p] \qquad m' = m[x \mapsto v]\end{array}}{\langle m, P, B\rangle \xrightarrow{\epsilon}_{\mathsf{PTSO_{syn}}} \langle m', P', B\rangle}$$

PERSIST-FO

$$\frac{\begin{array}{c}P(x) = \mathtt{FO}(\_) \cdot p \\ P' = P[x \mapsto p]\end{array}}{\langle m, P, B\rangle \xrightarrow{\epsilon}_{\mathsf{PTSO_{syn}}} \langle m, P', B\rangle}$$

Fig. 2. The $\mathsf{PTSO_{syn}}$ Persistent Memory Subsystem (differences w.r.t. Px86 are highlighted)

entries from the *head* of a store buffer, so only PROP-FO transitions may not use the store buffers as perfect FIFO queues.

*Example 4.2.* It is instructive to refer back to the simple programs in Ex. 3.3 and see how same judgments are obtained for $\mathsf{PTSO_{syn}}$ albeit in a different way. In particular, in these example the propagation order must follow the issue order. Then, the behavior of program (C) is not explained by out-of-order propagation, but rather by using the fact that $\mathtt{x} := 1$ and $\mathtt{y} := 1$ are propagated to different persistence buffers, and thus can persist in an order opposite to their propagation order.

*Example 4.3.* As mentioned above, while $\mathsf{PTSO_{syn}}$ forbids propagating writes/flushes/sfences before propagating prior entries, this is still not the case for flush-optimals that can propagate before prior write/flushes/flush-optimals.

The program on the right demonstrates such case. The annotated outcome is allowed in Px86 (and thus, has to be allowed in $\mathrm{PTSO_{syn}}$). The fact that `y := 3` persisted implies that `y := 2` propagated after `y := 1`. Now, since writes propagate in order, we obtain that `y := 2` propagated after `x := 1`. Had we required that `fo(x)` must propagate after `y := 2`, we would obtain that `fo(x)` must propagate after `x := 1`. In turn, due to the sfence instruction, this would forbid `z := 1` from persisting before `x := 1` has persisted.

```
x := 1 ;       || y := 2 ;
y := 1 ;       || fo(x) ;
if y = 2 then  || sfence ;
  y := 3 ;     || z := 1 ;
```

*Remark 5.* There is an alternative formulation for $\mathrm{PTSO_{syn}}$ that always propagates flush-optimals from the *head* of the store buffer. This simplification comes at the expense of complicating how flush-optimals are added into the store buffer upon issuing. Concretely, we can have a FLUSH-OPT step that does not put the new $\mathtt{FO}(x)$ entry in the tail of the store buffer (omit $\mathtt{FO}(x)$ from the WRITE/FLUSH/FLUSH-OPT/SFENCE issuing step). Instead, the step looks inside the buffer and puts the $\mathtt{FO}(x)$-entry immediately after the last pending entry $l$ with $\mathtt{loc}(l) = x$ or $\mathtt{typ}(l) = \mathtt{SF}$ (or at the head of the buffer is no such entry exists):

FLUSH-OPT$_1$

$$\frac{\begin{array}{c} l = \mathtt{FO}(x) \\ B(\tau) = b_{\text{head}} \cdot \alpha \cdot b_{\text{tail}} \qquad \mathtt{loc}(\alpha) = x \lor \alpha = \mathtt{SF} \\ \mathtt{W}(x,\_), \mathtt{FL}(x), \mathtt{FO}(x), \mathtt{SF} \notin b_{\text{tail}} \\ B' = B[\tau \mapsto b_{\text{head}} \cdot \alpha \cdot l \cdot b_{\text{tail}}] \end{array}}{\langle m, P, B\rangle \xrightarrow{\tau,l}_{\mathrm{PTSO_{syn}}} \langle m, P, B'\rangle}$$

FLUSH-OPT$_2$

$$\frac{\begin{array}{c} l = \mathtt{FO}(x) \\ \\ \mathtt{W}(x,\_), \mathtt{FL}(x), \mathtt{FO}(x), \mathtt{SF} \notin B(\tau) \\ B' = B[\tau \mapsto l \cdot B(\tau)] \end{array}}{\langle m, P, B\rangle \xrightarrow{\tau,l}_{\mathrm{PTSO_{syn}}} \langle m, P, B'\rangle}$$

This alternative reduces the level of non-determinism in the system. Roughly speaking, it is equivalent to eagerly taking PROP-FO-steps, which is sound, since delaying a PROP-FO-step may only put more constraints on the rest of the run. We suspect that insertions not in the tail of the buffer (even if done in deterministic positions) may appear slightly less intuitive than eliminations not from the head of the buffer, and so we continue with $\mathrm{PTSO_{syn}}$ as formulated in Fig. 2.

*Example 4.4.* An sfence (or an sfence-inducing operation: mfence and RMW) performed by one thread does not affect flush-optimals by other threads. To achieve this, $\mathrm{PTSO_{syn}}$ records thread identifiers in $\mathtt{FO}$-entries in the persistence buffer. (In Px86, this is captured by the fact that sfence only affects the propagation order from the (per-thread) store buffers.)

The program on the right demonstrates how this works. The annotated behavior is allowed by $\mathrm{PTSO_{syn}}$: the flush-optimal entry in `x`'s persistence buffer has to be in that buffer at the point the sfence is issued (since the second thread has already observed `y := 1`). But, since it is an sfence coming from the store buffer of the second thread, and the flush-optimal entry is by the first thread, the sfence has no effect in this case.

```
x := 1 ; || a := y ;  // 1
fo(x) ;  || sfence ;
y := 1 ; || if a = 1 then
         ||   z := 1 ;
```

The next lemma (used to prove Thm. 5.29 below) ensures that we can safely assume that crashes only happen when all store buffers are empty (i.e., ending with $B_\epsilon \triangleq \lambda\tau.\ \epsilon$). (Clearly, such assumption is wrong for the persistence buffers). Intuitively, it follows from the fact that we can always remove from a trace all thread operations starting from the first write/flush/sfence operation that did not propagate from the store buffer before the crash. These can only affect the volatile part of the state.

LEMMA 4.5. *Suppose that* $\langle m_0, P_\epsilon, B_\epsilon\rangle \overset{t}{\Rightarrow}_{\mathrm{PTSO_{syn}}} \langle m, P, B\rangle$. *Then:*

- $\langle m_0, P_\epsilon, B_\epsilon\rangle \overset{t}{\Rightarrow}_{\mathrm{PTSO_{syn}}} \langle m', P', B_\epsilon\rangle$ *for some* $m'$ *and* $P'$.
- $\langle m_0, P_\epsilon, B_\epsilon\rangle \overset{t'}{\Rightarrow}_{\mathrm{PTSO_{syn}}} \langle m, P, B_\epsilon\rangle$ *for some* $t' \lesssim t$.

## 4.1 Observational Equivalence of Px86 and PTSO$_{\mathsf{syn}}$

Our first main result is stated in the following theorem.

Theorem 4.6. Px86 *and* PTSO$_{\mathsf{syn}}$ *are observationally equivalent.*

We briefly outline the key steps in the proof of this theorem. The full proof presented in Appendix B.5 formalizes the following ideas by using *instrumented* memory subsystems and employing two different intermediate systems that bridge the gap between Px86 and PTSO$_{\mathsf{syn}}$.

We utilize Lemma 2.10, which splits the task of proving Theorem 4.6 into four parts:

(A) Every $m_0$-initialized PTSO$_{\mathsf{syn}}$-observable-trace is also an $m_0$-initialized Px86-observable-trace.
(B) For every $m_0$-to-$m$ PTSO$_{\mathsf{syn}}$-observable-trace $t$, some $t' \lesssim t$ is an $m_0$-to-$m$ Px86-observable-trace.
(C) Every $m_0$-initialized Px86-observable-trace is also an $m_0$-initialized PTSO$_{\mathsf{syn}}$-observable-trace.
(D) For every $m_0$-to-$m$ Px86-observable-trace $t$, some $t' \lesssim t$ is an $m_0$-to-$m$ PTSO$_{\mathsf{syn}}$-observable-trace.

Part (A) requires showing that Px86 allows the same observable behaviors as PTSO$_{\mathsf{syn}}$ regardless of the final memory. This part is straightforward: we perform silent PERSIST-W and PERSIST-FO steps at the end of the PTSO$_{\mathsf{syn}}$ run to completely drain the persistence buffers, and then move all the persistence steps to be immediately after corresponding propagation steps. It is then easy to demonstrate that Px86 can simulate such sequence of steps.

Part (B) requires showing that Px86 can survive crashes with the same non-volatile state as PTSO$_{\mathsf{syn}}$. We note that this cannot be always achieved by executing the exact same sequence of steps under PTSO$_{\mathsf{syn}}$ and Px86. Example 3.3(C) illustrates a case in point: If PTSO$_{\mathsf{syn}}$ propagates all of the instructions, and only persists the write $\mathtt{y} := 1$, to achieve the same result, Px86 needs to propagate $\mathtt{y} := 1$ ahead of propagating $\mathtt{fo}(\mathtt{x})$ (otherwise, the PERSIST-W step for $\mathtt{y} := 1$ would require persisting $\mathtt{fo}(\mathtt{x})$ first, resulting in a non-volatile state different from PTSO$_{\mathsf{syn}}$'s). Our proof strategy for part (B) is to reach the same non-volatile memory by omitting all propagation steps of non-persisting flush-optimals from the run. We prove that this results in a trace that can be transformed into a Px86-observable-trace.

Part (C) requires showing that PTSO$_{\mathsf{syn}}$ allows the same observable behaviors as Px86 regardless of the final memory. In order to satisfy stronger constraints on the content of the persistence buffers upon the propagation steps of PTSO$_{\mathsf{syn}}$, we employ a transformation like the one from part (A) and obtain a trace of Px86, in which every persisted instruction is persisted immediately after it is propagated. Unlike part (A), it is not trivial that PTSO$_{\mathsf{syn}}$ can simulate such a trace due to its more strict constraints on the propagation from the store buffers. We overcome this challenge by eagerly propagating and persisting flush-optimals as we construct an equivalent run of PTSO$_{\mathsf{syn}}$ (as a part of a forward simulation argument).

Part (D) requires showing that PTSO$_{\mathsf{syn}}$ can survive crashes with the same non-volatile state as Px86. This cannot be always achieved by executing the exact same sequence of steps under Px86 and PTSO$_{\mathsf{syn}}$, since they do not lead to the same non-volatile states: the synchronous semantics of flush, sfence, mfence and RMW instructions under PTSO$_{\mathsf{syn}}$ makes instructions persist earlier. However, the program state is lost after the crash, so at that point the client cannot observe outcomes of instructions that did not persist. Therefore, crashing before a flush/flush-optimal instruction persists is observationally indistinguishable from crashing before it propagates from the store buffer. These intuitions allow us to reach the non-volatile memory in PTSO$_{\mathsf{syn}}$ with a per-thread-prefix of the program trace that reached that memory in Px86. More concretely, we trim the sequence of steps of Px86 to a per-thread prefix in order to remove all propagation steps of non-persisting flush/flush-optimal instructions, and then move the persistence steps of the persisting instructions to be immediately after their propagation, which is made possible by certain commutativity properties of persistence steps. This way, we essentially obtain a $\text{PTSO}_{\text{syn}}$-observable-trace, which, as in part (C), formally requires the eager propagation and persistence of flush-optimals.

## 5 DECLARATIVE SEMANTICS

In this section we provide an alternative characterization of $\text{PTSO}_{\text{syn}}$ (and, due to the equivalence theorem, also of Px86) that is declarative (a.k.a. axiomatic) rather than operational. In such semantics, instead of considering machine traces that are totally ordered by definition, one aims to abstract from arbitrary choices of the order of operations, and maintain such order only when it is necessary to do so. Accordingly, behaviors of concurrent systems are represented as partial orders rather than total ones. This more abstract approach, while may be less intuitive to work with, often leads to much more succinct presentations, and has shown to be beneficial for comparing models and mapping from one model to another (see, e.g., [Podkopaev et al. 2019; Sarkar et al. 2012; Wickerson et al. 2017]), reasoning about program transformations (see, e.g., [Vafeiadis et al. 2015]), and bounded model checking (see, e.g., [Abdulla et al. 2018; Kokologiannakis et al. 2017]). In the current paper, the declarative semantics is instrumental for establishing the DRF and mapping theorem in §7.

We present two different declarative models of $\text{PTSO}_{\text{syn}}$. Roughly speaking, the first, called $\text{DPTSO}_{\text{syn}}$, is an extension the declarative TSO model in [Lahav et al. 2016], and it is closer to the operational semantics as it tracks the propagation order. The second, called $\text{DPTSO}^{\text{mo}}_{\text{syn}}$, is an extension the declarative TSO model in [Alglave et al. 2014] that employs per-location propagation orders on writes only, but ignores some of the program order edges.

### 5.1 A Declarative Framework for Persistency Specifications

Before introducing the declarative models, we present the general notions used to assign declarative semantics to persistent systems (see Def. 2.6). This requires several modifications of the standard declarative approach that does not handle persistency. First, we define execution graphs, each of which represents a particular behavior. We start with their nodes, called *events*.

*Definition 5.1.* An *event* is a triple $e = \langle \tau, n, l \rangle$, where $\tau \in \text{Tid} \cup \{\bot\}$ is a thread identifier ($\bot$ is used for *initialization events*), $n \in \mathbb{N}$ is a serial number, and $l \in \text{Lab}$ is an event label (as defined in Def. 2.1). The functions $\texttt{tid}$, $\#$, and $\texttt{lab}$ return the thread identifier, serial number, and label of an event. The functions $\texttt{typ}$, $\texttt{loc}$, $\texttt{val}_{\text{R}}$, and $\texttt{val}_{\text{W}}$ are lifted to events in the obvious way. We denote by $\text{E}$ the set of all events, and by $\text{Init}$ the set of initialization events, i.e., $\text{Init} \triangleq \{e \in \text{E} \mid \texttt{tid}(e) = \bot\}$. We use W, R, RMW, R-ex, MF, FL, FO, and SF for the sets of all events of the respective type (e.g., $\text{R} \triangleq \{e \in \text{E} \mid \texttt{typ}(e) = \text{R}\}$). Sub/superscripts are used to restrict these sets to certain location (e.g., $\text{W}_x = \{w \in \text{W} \mid \texttt{loc}(w) = x\}$) and/or thread identifier (e.g., $\text{E}^\tau = \{e \in \text{E} \mid \texttt{tid}(e) = \tau\}$).

Our representation of events induces a *sequenced-before* partial order on events, where $e_1 < e_2$ holds iff ($e_1 \in \text{Init}$ and $e_2 \notin \text{Init}$) or ($e_1, e_2 \notin \text{Init}$, $\texttt{tid}(e_1) = \texttt{tid}(e_2)$, and $\#(e_1) < \#(e_2)$). That is, initialization events precede all non-initialization events, and events of the same thread are ordered according to their serial numbers.

Next, a (standard) mapping justifies every read with a corresponding write event:

*Definition 5.2.* A relation $rf$ is a *reads-from* relation for a set $E$ of events if the following hold:

- $rf \subseteq (E \cap (\text{W} \cup \text{RMW})) \times (E \cap (\text{R} \cup \text{RMW} \cup \text{R-ex}))$.
- If $\langle w, r \rangle \in rf$, then $\texttt{loc}(w) = \texttt{loc}(r)$ and $\texttt{val}_{\text{W}}(w) = \texttt{val}_{\text{R}}(r)$.
- If $\langle w_1, r \rangle, \langle w_2, r \rangle \in rf$, then $w_1 = w_2$ (that is, $rf^{-1}$ is functional).
- $\forall r \in E \cap (\text{R} \cup \text{RMW} \cup \text{R-ex}).\ \exists w.\ \langle w, r \rangle \in rf$ (each read event reads from some write event).

The "non-volatile outcome" of an execution graph is recorded in *memory assignments*:

*Definition 5.3.* A *memory assignment* $\mu$ for a set $E$ of events is a function assigning an event in $E \cap (\mathsf{W}_x \cup \mathsf{RMW}_x)$ to every location $x \in \mathsf{Loc}$.

Intuitively speaking, $\mu$ records the last write in the graph that persisted before the crash. Using the above notions, we formally define execution graphs.

*Definition 5.4.* An *execution graph* is a tuple $G = \langle E, rf, \mu \rangle$, where $E$ is a finite set of events, $rf$ is a reads-from relation for $E$, and $\mu$ is a memory assignment for $E$. The components of $G$ are denoted by $G.\mathsf{E}$, $G.\mathtt{rf}$, and $G.\mathsf{M}$. For a set $A \subseteq \mathsf{E}$, we write $G.A$ for $G.\mathsf{E} \cap A$ (e.g., $G.\mathsf{W}_x = G.\mathsf{E} \cap \mathsf{W}_x$). In addition, derived relations and functions are defined as follows:

$$G.\mathsf{po} \triangleq \{\langle e_1, e_2 \rangle \in G.\mathsf{E} \times G.\mathsf{E} \mid e_1 < e_2\} \qquad (\textit{program order})$$

$$G.\mathtt{rfe} \triangleq G.\mathtt{rf} \setminus G.\mathsf{po} \qquad (\textit{external reads-from})$$

$$m(G) \triangleq \lambda x.\ \mathtt{val}_{\mathtt{W}}(G.\mathsf{M}(x)) \qquad (\textit{induced persistent memory})$$

Our execution graphs are always *initialized* with some initial memory:

*Definition 5.5.* Given $m : \mathsf{Loc} \to \mathsf{Val}$, an execution graph $G$ is *$m$-initialized* if $G.\mathsf{E} \cap \mathsf{Init} = \{\langle \bot, 0, \mathtt{W}(x, m(x)) \rangle \mid x \in \mathsf{Loc}\}$. We say that $G$ is *initialized* if it is $m$-initialized for some $m : \mathsf{Loc} \to \mathsf{Val}$. We denote by $m_{\mathsf{Init}}(G)$ the (unique) function $m$ for which $G$ is $m$-initialized.

A declarative characterization of a persistent memory subsystem is captured by the set of execution graphs that the subsystem allows. Intuitively speaking, the conditions it enforces on $G.\mathtt{rf}$ correspond to the consistency aspect of the memory subsystem; and those on $G.\mathsf{M}$ correspond to its persistency aspect.

*Definition 5.6.* A *declarative persistency model* is a set $D$ of execution graphs. We refer to the elements of $D$ as *$D$-consistent* execution graphs.

Now, to use a declarative persistency model for specifying the possible behaviors of programs (namely, what program states are reachable under a given model $D$), we need to formally associate execution graphs with programs. The next definition uses the characterization of programs as LTSs to provide this association. (Note that at this stage $G.\mathtt{rf}$ and $G.\mathsf{M}$ are completely arbitrary.)

*Notation 5.7.* For a set $E$ of events, thread identifier $\tau \in \mathsf{Tid}$, and event label $l \in \mathsf{Lab}$, we write $\mathsf{NextEvent}(E, \tau, l)$ to denote the event given by $\langle \tau, \max\{\#(e) \mid e \in G.\mathsf{E}^{\tau}\} + 1, l \rangle$.

*Definition 5.8.* An execution graph $G$ is *generated by a program $Pr$ with final state $\overline{q}$* if $\langle \overline{q}_{\mathsf{Init}}, E_0 \rangle \to^* \langle \overline{q}, G.\mathsf{E} \rangle$ for some $\overline{q}_{\mathsf{Init}} \in Pr.\mathsf{Q}_{\mathsf{Init}}$ and $E_0 \subseteq \mathsf{Init}$, where $\to$ is defined by:

$$\frac{\overline{q} \xrightarrow{\tau, l}_{Pr} \overline{q}'}{\langle \overline{q}, E \rangle \to \langle \overline{q}', E \cup \{\mathsf{NextEvent}(E, \tau, l)\} \rangle} \qquad \frac{\overline{q} \xrightarrow{\epsilon}_{Pr} \overline{q}'}{\langle \overline{q}, E \rangle \to \langle \overline{q}', E \rangle}$$

We say that $G$ is *generated by $Pr$* if it is generated by $Pr$ with *some* final state.

The following alternative characterization of the association of graphs and programs, based on traces, is useful below.

*Definition 5.9.* An observable program trace $t \in (\mathsf{Tid} \times \mathsf{Lab})^*$ is *induced* by an execution graph $G$ if $t = \langle \mathtt{tid}(e_1), \mathtt{lab}(e_1) \rangle, \dots, \langle \mathtt{tid}(e_n), \mathtt{lab}(e_n) \rangle$ for some enumeration $e_1, \dots, e_n$ of $G.\mathsf{E} \setminus \mathsf{Init}$ that respects $G.\mathsf{po}$ (i.e., $\langle e_i, e_j \rangle \in G.\mathsf{po}$ implies that $i < j$). We denote by $\mathsf{traces}(G)$ the set of all observable program trace that are induced by $G$.

Proposition 5.10. *Let $t \in \mathsf{traces}(G)$. Then, $\mathsf{traces}(G) = \{t' \in (\mathsf{Tid} \times \mathsf{Lab})^* \mid t' \sim t\}$ (where $\sim$ is per-thread equivalence of observable program traces, see Def. 2.3).*

Proposition 5.11. *If $G$ is generated by $Pr$ with final state $\overline{q}$, then for every $t \in \mathsf{traces}(G)$, we have $\overline{q}_{\mathsf{Init}} \overset{t}{\Rightarrow}_{Pr} \overline{q}$ for some $\overline{q}_{\mathsf{Init}} \in Pr.\mathsf{Q}_{\mathsf{Init}}$.*

Proposition 5.12. *If $\overline{q}_{\mathsf{Init}} \overset{t}{\Rightarrow}_{Pr} \overline{q}$ for some $\overline{q}_{\mathsf{Init}} \in Pr.\mathsf{Q}_{\mathsf{Init}}$ and $t \in \mathsf{traces}(G)$, then $G$ is generated by $Pr$ with final state $\overline{q}$.*

Now, following [Raad et al. 2020], reachability of program states under a declarative persistency model $D$ is defined using "chains" of $D$-consistent execution graphs, each of which represents the behavior obtained between two consecutive crashes. Examples 5.21 and 5.22 below illustrate some execution graph chains for simple programs.

*Definition 5.13.* A program state $\overline{q} \in Pr.\mathsf{Q}$ is *reachable under a declarative persistency model $D$* if there exist $D$-consistent execution graphs $G_0, \ldots, G_n$ such that:

- For every $0 \leq i \leq n-1$, $G_i$ is generated by $Pr$.
- $G_n$ is generated by $Pr$ with final state $\overline{q}$.
- $G_0$ is $m_{\mathsf{Init}}$-initialized (where $m_{\mathsf{Init}} = \lambda x \in \mathsf{Loc}.\ 0$).
- For every $1 \leq i \leq n$, $G_i$ is $m(G_{i-1})$-initialized.

In the sequel, we provide declarative formulations for (operational) persistent memory subsystems (see Def. 2.5). Observational refinements (and equivalence) between a persistent memory subsystem $M$ and a declarative persistency model $D$ are defined just like observational refinements between persistent memory subsystems (see Def. 2.8), comparing reachable program states under $M$ (using Def. 2.7) to reachable program states under $D$ (using Def. 5.13).

The following lemmas are useful establishing refinements without considering *all programs* and *crashes* (compare with Lemma 2.10). In both lemmas $M$ denotes a persistent memory subsystem $M$, and $D$ denotes a declarative persistency model.

Lemma 5.14. *The following conditions together ensure that $M$ observationally refines $D$:*

*(i) For every $m_0$-initialized $M$-observable-trace $t$, there exists a $D$-consistent $m_0$-initialized execution graph $G$ such that $t \in \mathsf{traces}(G)$.*

*(ii) For every $m_0$-to-$m$ $M$-observable-trace $t$, there exist $t' \lesssim t$ and $D$-consistent $m_0$-initialized execution graph such that $t' \in \mathsf{traces}(G)$ and $m(G) = m$.*

Lemma 5.15. *If for every $D$-consistent initialized execution graph $G$, some $t \in \mathsf{traces}(G)$ is an $m_{\mathsf{Init}}(G)$-to-$m(G)$ $M$-observable-trace, then $D$ observationally refines $M$.*

## 5.2 The DPTSO$_{\mathsf{syn}}$ Declarative Persistency Model

In this section we define the declarative DPTSO$_{\mathsf{syn}}$ model. As in (standard) TSO models [Lahav et al. 2016; Owens et al. 2009], DPTSO$_{\mathsf{syn}}$-consistency requires one to justify an execution graph with a *TSO propagation order* (*tpo*), which, roughly speaking, corresponds to the order in which the events in the graph are propagated from the store buffers.

*Definition 5.16.* The set of *propagated events*, denoted by P, is given by:

$$\mathsf{P} \triangleq \mathsf{W} \cup \mathsf{RMW} \cup \mathsf{R\text{-}ex} \cup \mathsf{MF} \cup \mathsf{FL} \cup \mathsf{FO} \cup \mathsf{SF} \qquad (= \mathsf{E} \setminus \mathsf{R}).$$

Given an execution graph $G$, a strict total order *tpo* on $G.\mathsf{P}$ is called a *TSO propagation order* for $G$.

DPTSO$_{\mathsf{syn}}$-consistency sets several conditions on the TSO propagation order that, except for one novel condition related to persistency, are adopted from the model in [Lahav et al. 2016] (which, in turn, is a variant of the model in [Owens et al. 2009]). To define these conditions, we use the standard "from-read" derived relation, which places a read (or RMW) $r$ before a write (or RMW) $w$

when $r$ reads from a write that was propagated before $w$. We parametrize this concept by the order on writes. (Here we only need $R = tpo$, but we reuse this definition in Def. 5.25 with a different $R$.)

*Definition 5.17.* The *from-read* (a.k.a. *reads-before*) relation for an execution graph $G$ and a strict partial order $R$ on $G.\mathsf{E}$, denoted by $G.\mathtt{fr}(R)$, is defined by:

$$G.\mathtt{fr}(R) \triangleq \bigcup_{x \in \mathsf{Loc}} ([\mathsf{R}_x \cup \mathsf{RMW}_x \cup \mathsf{R\text{-}ex}_x] \,;\, G.\mathtt{rf}^{-1} \,;\, R \,;\, [\mathsf{W}_x \cup \mathsf{RMW}_x]) \setminus [\mathsf{E}].$$

Next, for persistency, we use one more derived relation. Since flushes and sfences in $\mathrm{PTSO_{syn}}$ take effect at the moment they propagate from the store buffer, we can *derive* the existence of a propagation order from any flush event to location $x$ (or flush-optimal to $x$ followed by sfence) to any write $w$ to $x$ that propagated from the store buffer after $G.\mathsf{M}(x)$ persisted. Indeed, if the propagation order went in the opposite direction, we would be forced to persist $w$ and overwrite $G.\mathsf{M}(x)$, but the latter corresponds the last persisted write to $x$. This derived order is formalized as follows. (Again, we need $R = tpo$, but this definition is reused in Def. 5.25 with a different $R$.)

*Definition 5.18.* The *derived TSO propagation order* for an execution graph $G$ and a strict partial order $R$ on $G.\mathsf{E}$, denoted by $G.\mathtt{dtpo}(R)$, is defined by:

$$G.\mathtt{dtpo}(R) \triangleq \bigcup_{x \in \mathsf{Loc}} G.\mathsf{FLO}_x \times \{w \in \mathsf{W}_x \cup \mathsf{RMW}_x \mid \langle G.\mathsf{M}(x), w\rangle \in R\}$$

where $G.\mathsf{FLO}_x$ is the following set:

$$G.\mathsf{FLO}_x \triangleq G.\mathsf{FL}_x \cup (\mathsf{FO}_x \cap \mathit{dom}(G.\mathsf{po} \,;\, [\mathsf{RMW} \cup \mathsf{R\text{-}ex} \cup \mathsf{MF} \cup \mathsf{SF}])).$$

Using $\mathtt{fr}$ and $\mathtt{dtpo}$, $\mathrm{DPTSO_{syn}}$-consistency is defined as follows.

*Definition 5.19.* The declarative persistency model $\mathrm{DPTSO_{syn}}$ consists of all execution graphs $G$ for which there exists a propagation order $tpo$ for $G$ such that the following hold:

(1) For every $a, b \in \mathsf{P}$, except for the case that $a \in \mathsf{W} \cup \mathsf{FL} \cup \mathsf{FO}$, $b \in \mathsf{FO}$, and $\mathtt{loc}(a) \neq \mathtt{loc}(b)$, if $\langle a, b\rangle \in G.\mathsf{po}$, then $\langle a, b\rangle \in tpo$.
(2) $tpo^? \,;\, G.\mathtt{rfe} \,;\, G.\mathsf{po}^?$ is irreflexive.
(3) $G.\mathtt{fr}(tpo) \,;\, G.\mathtt{rfe}^? \,;\, G.\mathsf{po}$ is irreflexive.
(4) $G.\mathtt{fr}(tpo) \,;\, tpo$ is irreflexive.
(5) $G.\mathtt{fr}(tpo) \,;\, tpo \,;\, G.\mathtt{rfe} \,;\, G.\mathsf{po}$ is irreflexive.
(6) $G.\mathtt{fr}(tpo) \,;\, tpo \,;\, [\mathsf{RMW} \cup \mathsf{R\text{-}ex} \cup \mathsf{MF}] \,;\, G.\mathsf{po}$ is irreflexive.
(7) $G.\mathtt{dtpo}(tpo) \,;\, tpo$ is irreflexive.

Conditions (1) – (6) take care of the concurrency part of the model. They are taken from [Lahav et al. 2016] and slightly adapted to take into account the fact that our propagation order also orders FL, FO, and SF events which do not exist in non-persistent TSO models.[5] The only conditions that affect the propagation order on such events are (1) and (2). Condition (1) forces the propagation order to agree with the program order, except for the order between a W/FL/FO-event and a subsequent FO-event to a different location. This corresponds to the fact that propagation from $\mathrm{PTSO_{syn}}$'s store buffers is *in-order*, except for out-of-order propagation of FO's, which can "overtake" preceding writes, flushes, and flush-optimals to different locations. In turn, condition (2) ensures that if a read event observes some write $w$ in the persistence buffer (or persistent memory) via $G.\mathtt{rfe}$, then subsequent events (including FL/FO/SF-events) are necessarily propagated from the store buffer after the write $w$.

[5] Another technical difference is that we ensure here that failed CAS instructions, represented as R-ex events, are also acting as mfences, while in [Lahav et al. 2016; Raad et al. 2020] they are not distinguished from plain reads.

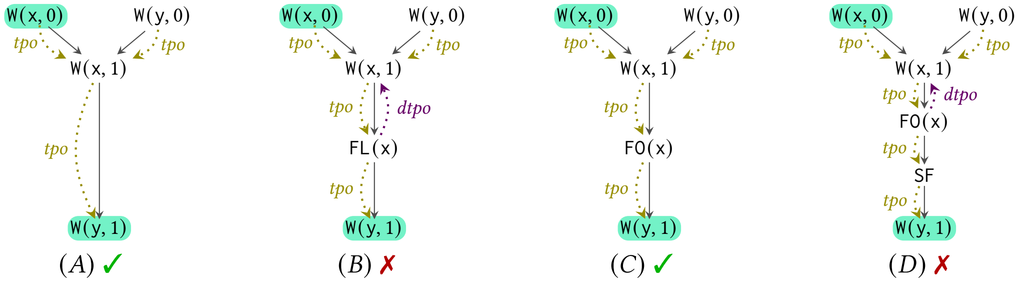
Condition (7) is our novel constraint. It is the only condition required for the persistency part of the model. The approach in [Raad et al. 2020] for Px86 requires the existence of a persistence order, reflecting the order in which writes persist (after they propagate), and enforce certain condition on this order. This makes the semantics less abstract (in the sense that it is closer to operational traces). Instead, we use the derived propagation order (induced by the graph component, $G.\mathtt{M}$), and require that it must agree with the propagation order itself. This condition ensures that if a write $w$ to location $x$ propagated from the store buffer before some flush to $x$, then the last persisted write cannot be a write that propagated *before* $w$. The same holds if $w$ propagated before some flush-optimal to $x$ that is followed by an sfence by the same thread (or any other instruction that has the effect of an sfence).

The following simple lemma is useful below.

Lemma 5.20. *Let $tpo$ be a propagation order for an execution graph $G$ for which the conditions of Def. 5.19 hold. Then, $G.\mathtt{dtpo}(tpo) \subseteq tpo$.*

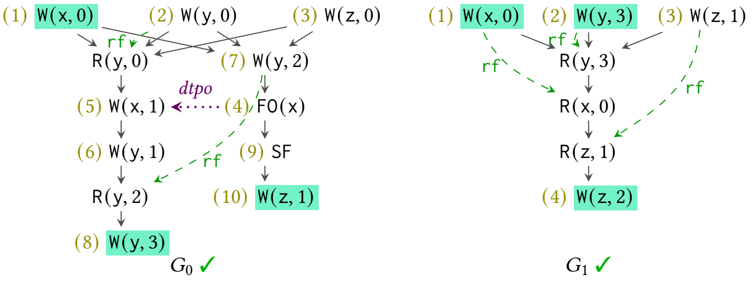
Proof. Easily follows from the fact that $tpo$ is total on $G.\mathtt{P}$ and the last condition in Def. 5.19. □

*Example 5.21.* The execution graphs depicted below correspond to the annotated behaviors of the simple sequential programs in Ex. 3.3. For every location $x$, the event $G.\mathtt{M}(x)$ is highlighted. The solid edges are program order edges. In each graph, we also depict the $tpo$-edges that are forced in order to satisfy conditions (1) – (6) above, and the $G.\mathtt{dtpo}(tpo)$-edges they induce. Execution graphs (A) and (C) are DPTSO$_{\mathsf{syn}}$-consistent, while (B) and (D) violate condition (7) above.



*Example 5.22.* The following example (variant of Ex. 4.3) demonstrates a non-volatile outcome that is justified with a sequence of two DPTSO$_{\mathsf{syn}}$-consistent execution graphs. In the graphs below we use serial numbers $(n)$ to present a possible valid $tpo$ relation Note that, for the first graph, it is crucial that program order from a write to an FO-event of a different location does not enforce a $tpo$-order in the same direction (otherwise, the graph would violate condition (7) above).

```
if (y = 3) then     ||
  if (x = 0) then   ||
    if (z = 1) then || y := 2 ;
      z := 2 ;      || fo(x) ;
x := 1 ;            || sfence ;
y := 1 ;            || z := 1 ;
if y = 2 then       ||
  y := 3 ;          ||
```



## 5.3 An Equivalent Declarative Persistency Model: DPTSO$^{\mathsf{mo}}_{\mathsf{syn}}$

We present an equivalent more abstract declarative model that requires existential quantification over *modification orders*, rather than over propagation orders (total orders of $G.\mathtt{P}$). Modification orders totally order writes (including RMWs) to the same location, leaving unspecified the order between other events, as well as the order between writes to different locations. This alternative formulation has a global nature: it identifies an "happens-before" relation and requires acyclicity this relation. In particular, it allows us to relate $\mathsf{PTSO}_{\mathsf{syn}}$ to an SC persistency model (see §7).

Unlike in SC, in TSO we cannot include $G.\mathsf{po}$ in the "happens-before" relation. Instead, we use a restricted subset, which consists of the program order edges that are "preserved".

*Definition 5.23.* The *preserved program order* relation for an execution graph $G$, denoted by $G.\mathsf{ppo}$, is defined by:

$$G.\mathsf{ppo} \triangleq \left\{ \langle a,b\rangle \in G.\mathsf{po} \;\middle|\; \begin{array}{l} (a \in \mathsf{W} \cup \mathsf{FL} \cup \mathsf{FO} \cup \mathsf{SF} \implies b \notin \mathsf{R}) \land \\ (a \in \mathsf{W} \cup \mathsf{FL} \cup \mathsf{FO} \land \mathtt{loc}(a) \neq \mathtt{loc}(b) \implies b \notin \mathsf{FO}) \end{array} \right\}$$

This definition extends the (non-persistent) preserved program order of TSO that is given by $\{\langle a,b\rangle \in G.\mathsf{po} \mid a \in \mathsf{W} \implies b \notin \mathsf{R}\}$ [Alglave et al. 2014].

Using ppo, we state a global acyclicity condition, and show that it must hold in $\mathsf{DPTSO}_{\mathsf{syn}}$-consistent executions.

Lemma 5.24. *Let $tpo$ be a propagation order for an execution graph $G$ for which the conditions of Def. 5.19 hold. Then, $G.\mathsf{ppo} \cup G.\mathtt{rfe} \cup tpo \cup G.\mathtt{fr}(tpo)$ is acyclic.*

Proof (outline). The proof considers a cycle in $G.\mathsf{ppo} \cup G.\mathtt{rfe} \cup tpo \cup G.\mathtt{fr}(tpo)$ of minimal length. The fact that $tpo$ is total on $G.\mathsf{P}$ and the minimality of the cycle imply that this cycle may contain at most two events in P. Then, each of the possible cases is handled using one of the conditions of Def. 5.19. □

We now switch from propagation orders to modification orders and formulate the alternative declarative model.

*Definition 5.25.* A relation $mo$ is a *modification order* for an execution graph $G$ if $mo$ is a disjoint union of relations $\{mo_x\}_{x\in\mathsf{Loc}}$ where each $mo_x$ is a strict total order on $G.\mathsf{E} \cap (\mathsf{W}_x \cup \mathsf{RMW}_x)$. Given a modification order $mo$ for $G$, the $\mathsf{PTSO}_{\mathsf{syn}}$-*happens-before* relation, denoted by $G.\mathsf{hb}(mo)$, is defined by:

$$G.\mathsf{hb}(mo) \triangleq (G.\mathsf{ppo} \cup G.\mathtt{rfe} \cup mo \cup G.\mathtt{fr}(mo) \cup G.\mathtt{dtpo}(mo))^+.$$

*Definition 5.26.* The declarative persistency model $\mathsf{DPTSO}^{\mathsf{mo}}_{\mathsf{syn}}$ consists of all execution graphs $G$ for which there exists a modification order $mo$ for $G$ such that the following hold:
(1) $G.\mathsf{hb}(mo)$ is irreflexive. (2) $G.\mathtt{fr}(mo) \,;\, G.\mathsf{po}$ is irreflexive.

In addition to requiring that the $\mathsf{PTSO}_{\mathsf{syn}}$-happens-before is irreflexive, Def. 5.26 forbids $G.\mathsf{po}$ to contradict $G.\mathtt{fr}(mo)$. Since program order edges from writes to reads are not included in $G.\mathsf{hb}(mo)$, the latter condition is needed to ensure "per-location-coherence" [Alglave et al. 2014].

*Example 5.27.* Revisiting Ex. 5.21 (B), in $\mathsf{DPTSO}^{\mathsf{mo}}_{\mathsf{syn}}$-inconsistency follows from the $G.\mathtt{dtpo}(mo) \,;$ ppo loop from the flush event ($mo$ is forced to agree with $G.\mathsf{po}$). In turn, the consistency of $G_0$ in Ex. 5.22 only requires to provide a modification order, which can have (1) → (5) for x, (2) → (6) → (7) → (8) for y, and (3) → (10) for z. Note that $mo$ does not order writes to different locations as well as the flush-optimal and the sfence events.

We prove the equivalence of $\mathsf{DPTSO}_{\mathsf{syn}}$ and $\mathsf{DPTSO}^{\mathsf{mo}}_{\mathsf{syn}}$.

Theorem 5.28. $\mathsf{DPTSO}_{\mathsf{syn}} = \mathsf{DPTSO}^{\mathsf{mo}}_{\mathsf{syn}}$.

Proof. For one direction, let $G$ be a $\mathsf{DPTSO_{syn}}$-consistent execution graph. Let $tpo$ be a propagation order for $G$ that satisfies the conditions of Def. 5.19. We define $mo \triangleq \bigcup_{x \in \mathsf{Loc}} [\mathsf{W}_x \cup \mathsf{RMW}_x] ; tpo ; [\mathsf{W}_x \cup \mathsf{RMW}_x]$. By definition, we have $G.\mathtt{fr}(mo) = G.\mathtt{fr}(tpo)$ and $G.\mathtt{dtpo}(mo) = G.\mathtt{dtpo}(tpo)$. Using Lemma 5.24 and Lemma 5.20, it follows that $mo$ satisfies the conditions of Def. 5.26, and so $G$ is $\mathsf{DPTSO^{mo}_{syn}}$-consistent.

For the converse, let $G$ be a $\mathsf{DPTSO^{mo}_{syn}}$-consistent execution graph. Let $mo$ be a modification order for $G$ that satisfies the conditions of Def. 5.26. Let $R$ be any total order on $G.\mathsf{E}$ extending $G.\mathtt{hb}(mo)$. Let $tpo \triangleq [\mathsf{P}] ; R ; [\mathsf{P}]$. Again, we have $G.\mathtt{fr}(tpo) = G.\mathtt{fr}(mo)$ and $G.\mathtt{dtpo}(tpo) = G.\mathtt{dtpo}(mo)$. This construction ensures that $G.\mathtt{ppo} \cup G.\mathtt{rfe} \cup tpo \cup G.\mathtt{fr}(tpo) \cup G.\mathtt{dtpo}(mo)$ is contained in $R$, and thus acyclic. Then, all conditions of Def. 5.19 follow. □

## 5.4 Equivalence of $\mathsf{PTSO_{syn}}$ and $\mathsf{DPTSO_{syn}}$

Using Lemmas 5.14 and 5.15, we show that $\mathsf{PTSO_{syn}}$ and $\mathsf{DPTSO_{syn}}$ are observationally equivalent. (Note that for showing that $\mathsf{DPTSO_{syn}}$ observationally refines $\mathsf{PTSO_{syn}}$, we use the Lemma 5.24.)

Theorem 5.29. *$\mathsf{PTSO_{syn}}$ and $\mathsf{DPTSO_{syn}}$ are observationally equivalent.*

The proof is given in Appendix C.

# 6 PERSISTENT MEMORY SUBSYSTEM: PSC

In this section we present an SC-based persistent memory subsystem, which we call PSC. This system is stronger, and thus easier to program with, than $\mathsf{PTSO_{syn}}$. From a formal verification point of view, assuming finite-state programs, in §6.1 we show that PSC can be represented as a *finite* transition system (like standard SC semantics), so that reachability of program states under PSC is trivially decidable (PSPACE-complete). In §6.2, we accompany the operational definition with an equivalent declarative one. The latter is used in §7 to relate $\mathsf{PTSO_{syn}}$ and PSC.

The persistent memory subsystem PSC is obtained from $\mathsf{PTSO_{syn}}$ by simply discarding the store buffers, thus creating direct links between the threads and the per-location persistence buffers. More concretely, issued writes go directly to the appropriate persistence buffer (made globally visible immediately when they are issued); issued flushes to location $x$ wait until the $x$-persistence-buffer has drained; issued flush-optimals go directly to the appropriate persistence buffer; and issued sfences wait until all writes before a flush-optimal entry (of the same thread issuing the sfence) in every per-location persistence buffer have persisted. As in $\mathsf{PTSO_{syn}}$, RMWs, failed RMWs, and mfences induce an sfence.[6] We note that *without crashes*, the effect of the persistence buffers is unobservable, and PSC trivially coincides with the standard SC semantics.

We note that, unlike for $\mathsf{PTSO_{syn}}$, discarding the store buffers from Px86 results in a model that is stronger than PSC, where flush and flush-optimals are equivalent (which makes sfences redundant), and providing this stronger semantics requires one to place barriers even for sequential programs.

To formally define PSC, we again use a "lookup" function (overloading again the get notation). In PSC, when thread $\tau$ reads from a shared location $x$ it obtains the latest accessible value of $x$, which is defined by applying the following get function on the current persistent memory $m$, and the current per-location persistence buffer $p$ for location $x$:

$$\mathrm{get}(m,p) \triangleq \lambda x. \begin{cases} v & p = p_1 \cdot \mathtt{W}(v) \cdot p_2 \wedge \mathtt{W}(\_) \notin p_2 \\ m(x) & \text{otherwise} \end{cases}$$

Using this definition, PSC is presented in Fig. 3. Its set of volatile states, $\mathsf{PSC}.\tilde{\mathsf{Q}}$, consists all per-location-persistence-buffer mappings. Initially all buffers are empty ($\mathsf{PSC}.\tilde{\mathsf{Q}}_{\mathsf{Init}} = \{P_\epsilon\}$).

[6] In PSC there is no need in mfences, as they are equivalent to sfences; we only keep them here for the sake uniformity.

$$m \in \mathsf{Loc} \to \mathsf{Val} \qquad P \in \mathsf{Loc} \to (\{\mathtt{W}(v) \mid v \in \mathsf{Val}\} \cup \{\mathtt{FO}(\tau) \mid \tau \in \mathsf{Tid}\})^* \qquad P_{\mathsf{Init}} \triangleq \lambda x.\ \epsilon$$

$$\textsc{write}\ \frac{l = \mathtt{W}(x,v) \quad P' = P[x \mapsto P(x) \cdot \mathtt{W}(v)]}{\langle m, P\rangle \xrightarrow{\tau,l}_{\mathsf{PSC}} \langle m, P'\rangle} \qquad \textsc{read}\ \frac{l = \mathtt{R}(x,v) \quad \mathsf{get}(m, P(x))(x) = v}{\langle m, P\rangle \xrightarrow{\tau,l}_{\mathsf{PSC}} \langle m, P\rangle}$$

$$\textsc{rmw}\ \frac{l = \mathtt{RMW}(x, v_{\mathtt{R}}, v_{\mathtt{W}}) \quad \mathsf{get}(m, P(x))(x) = v_{\mathtt{R}} \quad \forall y.\ \mathtt{FO}(\tau) \notin P(y) \quad P' = P[x \mapsto P(x) \cdot \mathtt{W}(v_{\mathtt{W}})]}{\langle m, P\rangle \xrightarrow{\tau,l}_{\mathsf{PSC}} \langle m, P'\rangle}$$

$$\textsc{rmw-fail}\ \frac{l = \mathtt{R\text{-}ex}(x,v) \quad \mathsf{get}(m, P(x))(x) = v \quad \forall y.\ \mathtt{FO}(\tau) \notin P(y)}{\langle m, P\rangle \xrightarrow{\tau,l}_{\mathsf{PSC}} \langle m, P\rangle}$$

$$\textsc{mfence/sfence}\ \frac{l \in \{\mathtt{MF}, \mathtt{SF}\} \quad \forall y.\ \mathtt{FO}(\tau) \notin P(y)}{\langle m, P\rangle \xrightarrow{\tau,l}_{\mathsf{PSC}} \langle m, P\rangle} \qquad \textsc{flush}\ \frac{l = \mathtt{FL}(x) \quad P(x) = \epsilon}{\langle m, P\rangle \xrightarrow{\tau,l}_{\mathsf{PSC}} \langle m, P\rangle} \qquad \textsc{flush-opt}\ \frac{l = \mathtt{FO}(x) \quad P' = P[x \mapsto P(x) \cdot \mathtt{FO}(\tau)]}{\langle m, P\rangle \xrightarrow{\tau,l}_{\mathsf{PSC}} \langle m, P'\rangle}$$

$$\textsc{persist-w}\ \frac{P(x) = \mathtt{W}(v) \cdot p \quad P' = P[x \mapsto p] \quad m' = m[x \mapsto v]}{\langle m, P\rangle \xrightarrow{\epsilon}_{\mathsf{PSC}} \langle m', P'\rangle} \qquad \textsc{persist-fo}\ \frac{P(x) = \mathtt{FO}(\_) \cdot p \quad P' = P[x \mapsto p]}{\langle m, P\rangle \xrightarrow{\epsilon}_{\mathsf{PSC}} \langle m, P'\rangle}$$

Fig. 3. The PSC Persistent Memory Subsystem

*Example 6.1.* Except for Examples 4.3 and 5.22, PSC provides the same allowed/forbidden judgments as $\mathsf{PTSO_{syn}}$ (and Px86) for all of the examples above. (Obviously, standard litmus tests, which are not related to persistency, differentiate the models.) The annotated behaviors in Examples 4.3 and 5.22 are, however, disallowed in PSC. Indeed, by removing the store buffers, PSC requires that the order of entries in each persistence buffer follows exactly the order of issuing of the corresponding instructions (even when they are issued by different threads).

### 6.1 An Equivalent Finite Persistent Memory Subsystem: $\mathsf{PSC}_{\mathtt{fin}}$

From a formal verification perspective, PSC has another important advantage w.r.t. $\mathsf{PTSO_{syn}}$. Assuming finite-state programs (i.e., finite sets of threads, values and locations, but still, possibly, loopy programs) the reachability problem under PSC (that is, checking whether a given program state $\overline{q}$ is reachable under PSC according to Def. 2.7) is computationally simple—PSPACE-complete—just like under standard SC semantics [Kozen 1977]. Since PSC is an infinite state system (the persistence buffer are unbounded), the PSPACE upper bound is not immediate. To establish this bound, we present an alternative persistent memory subsystem, called $\mathsf{PSC}_{\mathtt{fin}}$, that is observationally equivalent to PSC, and, assuming that Tid and Loc are finite, $\mathsf{PSC}_{\mathtt{fin}}$ is a *finite* LTS.

The system $\mathsf{PSC}_{\mathtt{fin}}$ is presented in Fig. 4. Its states keep track of a non-volatile memory $m$, a (volatile) mapping $\tilde{m}$ of the most recent value to each location, a (volatile) set $L$ of locations that still persist, and a (volatile) set $T$ of thread identifiers that may perform an sfence (or an sfence-inducing instruction). Every write (or RMW) to some location $x$ can "choose" to not persist, removing $x$ from $L$, and thus forbidding later writes to $x$ to persist. Importantly, once some write to $x$ did not persist (so we have $x \notin L$), flushes to $x$ cannot be anymore executed (the system deadlocks). A similar mechanism handles flush-optimals: once a flush-optimal y thread $\tau$ "chooses" to not persist, further writes to the same location may not persist, and, moreover, it removes $\tau$ from $T$, so that thread $\tau$ cannot anymore execute an sfence-inducing instruction (sfence, mfence, or RMW).

$$m \in \mathsf{Loc} \to \mathsf{Val} \qquad \tilde{m} \in \mathsf{Loc} \to \mathsf{Val} \qquad L \subseteq \mathsf{Loc} \qquad T \subseteq \mathsf{Tid}$$

$$\tilde{m}_{\mathsf{Init}} \triangleq \lambda x.\ 0 \qquad L_{\mathsf{Init}} \triangleq \mathsf{Loc} \qquad T_{\mathsf{Init}} \triangleq \mathsf{Tid}$$

$$\textsc{write-persist}\ \frac{l = \mathtt{W}(x, v) \qquad x \in L \\ m' = m[x \mapsto v] \qquad \tilde{m}' = \tilde{m}[x \mapsto v]}{\langle m, \tilde{m}, L, T \rangle \xrightarrow{\tau, l}_{\mathsf{PSC}_{\mathtt{fin}}} \langle m', \tilde{m}', L, T \rangle}$$

$$\textsc{write-no-persist}\ \frac{l = \mathtt{W}(x, v) \\ \tilde{m}' = \tilde{m}[x \mapsto v] \qquad L' = L \setminus \{x\}}{\langle m, \tilde{m}, L, T \rangle \xrightarrow{\tau, l}_{\mathsf{PSC}_{\mathtt{fin}}} \langle m, \tilde{m}', L', T \rangle}$$

$$\textsc{read}\ \frac{l = \mathtt{R}(x, v) \\ \tilde{m}(x) = v}{\langle m, \tilde{m}, L, T \rangle \xrightarrow{\tau, l}_{\mathsf{PSC}_{\mathtt{fin}}} \langle m, \tilde{m}, L, T \rangle}$$

$$\textsc{rmw-persist}\ \frac{l = \mathtt{RMW}(x, v_{\mathtt{R}}, v_{\mathtt{W}}) \qquad x \in L \\ \tilde{m}(x) = v_{\mathtt{R}} \qquad \tau \in T \\ m' = m[x \mapsto v_{\mathtt{W}}] \qquad \tilde{m}' = \tilde{m}[x \mapsto v_{\mathtt{W}}]}{\langle m, \tilde{m}, L, T \rangle \xrightarrow{\tau, l}_{\mathsf{PSC}_{\mathtt{fin}}} \langle m', \tilde{m}', L', T \rangle}$$

$$\textsc{rmw-no-persist}\ \frac{l = \mathtt{RMW}(x, v_{\mathtt{R}}, v_{\mathtt{W}}) \\ \tilde{m}(x) = v_{\mathtt{R}} \qquad \tau \in T \\ \tilde{m}' = \tilde{m}[x \mapsto v_{\mathtt{W}}] \qquad L' = L \setminus \{x\}}{\langle m, \tilde{m}, L, T \rangle \xrightarrow{\tau, l}_{\mathsf{PSC}_{\mathtt{fin}}} \langle m, \tilde{m}', L', T \rangle}$$

$$\textsc{rmw-fail}\ \frac{l = \mathtt{R\text{-}ex}(x, v) \\ \tilde{m}(x) = v \qquad \tau \in T}{\langle m, \tilde{m}, L, T \rangle \xrightarrow{\tau, l}_{\mathsf{PSC}_{\mathtt{fin}}} \langle m, \tilde{m}, L, T \rangle}$$

$$\textsc{mfence/sfence}\ \frac{l \in \{\mathtt{MF}, \mathtt{SF}\} \qquad \tau \in T}{\langle m, \tilde{m}, L, T \rangle \xrightarrow{\tau, l}_{\mathsf{PSC}_{\mathtt{fin}}} \langle m, \tilde{m}, L, T \rangle}$$

$$\textsc{flush}\ \frac{l = \mathtt{FL}(x) \qquad x \in L}{\langle m, \tilde{m}, L, T \rangle \xrightarrow{\tau, l}_{\mathsf{PSC}_{\mathtt{fin}}} \langle m, \tilde{m}, L, T \rangle}$$

$$\textsc{flush-opt-persist}\ \frac{l = \mathtt{FO}(x) \qquad x \in L}{\langle m, \tilde{m}, L, T \rangle \xrightarrow{\tau, l}_{\mathsf{PSC}_{\mathtt{fin}}} \langle m, \tilde{m}, L, T \rangle}$$

$$\textsc{flush-opt-no-persist}\ \frac{l = \mathtt{FO}(x) \qquad L' = L \setminus \{x\} \qquad T' = T \setminus \{\tau\}}{\langle m, \tilde{m}, L, T \rangle \xrightarrow{\tau, l}_{\mathsf{PSC}_{\mathtt{fin}}} \langle m, \tilde{m}, L', T' \rangle}$$

Fig. 4. The $\mathsf{PSC}_{\mathtt{fin}}$ Persistent Memory Subsystem

Theorem 6.2. *PSC and $\mathsf{PSC}_{\mathtt{fin}}$ are observationally equivalent.*

*Remark 6.* One may apply a construction like $\mathsf{PSC}_{\mathtt{fin}}$ for $\mathsf{PTSO}_{\mathsf{syn}}$, namely replacing the persistence buffers with a standard non-volatile memory $\tilde{m}$ and (finite) sets $L$ and $T$. For $\mathsf{PTSO}_{\mathsf{syn}}$ such construction does not lead to a finite-state machine, as we will still have unbounded store buffers. The non-primitive recursive lower bound established by Atig et al. [2010] for reachability under the standard x86-TSO semantics easily extends to $\mathsf{PTSO}_{\mathsf{syn}}$. Indeed, for programs that start by resetting all memory locations to 0 (the very initial value), reachability of program states under $\mathsf{PTSO}_{\mathsf{syn}}$ coincides with reachability under TSO. Abdulla et al. [2021] establish the decidability of reachability under Px86 (equivalently, under $\mathsf{PTSO}_{\mathsf{syn}}$) by introducing a rather intricate equivalent model that can be used in the framework of well-structured transition systems.

## 6.2 The DPSC Declarative Persistency Model

We present a declarative formulation of PSC, which we call DPSC. As $\mathsf{DPTSO}^{\mathit{mo}}_{\mathsf{syn}}$, it is based on an "happens-before" relation.

*Definition 6.3.* Given a modification order $\mathit{mo}$ for an execution graph $G$, the PSC-*happens-before* relation, denoted by $G.\mathtt{hb}_{\mathsf{PSC}}(\mathit{mo})$, is defined by:

$$G.\mathtt{hb}_{\mathsf{PSC}}(\mathit{mo}) \triangleq (G.\mathtt{po} \cup G.\mathtt{rf} \cup \mathit{mo} \cup G.\mathtt{fr}(\mathit{mo}) \cup G.\mathtt{dtpo}(\mathit{mo}))^{+}.$$

$G.\mathtt{hb}_{\mathsf{PSC}}(\mathit{mo})$ extends the standard happens-before relation that defines SC [Alglave et al. 2014] with the derived propagation order ($G.\mathtt{dtpo}(\mathit{mo})$). In turn, it extends the $\mathsf{PTSO}_{\mathsf{syn}}$-happens-before (see Def. 5.25) by including *all* program order edges rather than only the "preserved" ones. Consistency simply enforces the acyclicity of $G.\mathtt{hb}_{\mathsf{PSC}}(\mathit{mo})$:

*Definition 6.4.* The declarative persistency model DPSC consists of all execution graphs $G$ for which there exists a modification order $\mathit{mo}$ for $G$ such that $G.\mathtt{hb}_{\mathsf{PSC}}(\mathit{mo})$ is irreflexive.

Next, we state the equivalence of PSC and DPSC (the proof is given in Appendix F).

Theorem 6.5. PSC *and* DPSC *are observationally equivalent.*

## 7 RELATING $\mathsf{PTSO_{SYN}}$ AND PSC

In this section we develop a data-race-freedom (DRF) guarantee for $\mathsf{PTSO_{syn}}$ w.r.t. the stronger and simpler PSC model. This guarantee identifies certain forms of races and ensures that if all executions of a given program do not exhibit such races, then the program's states that are reachable under $\mathsf{PTSO_{syn}}$ are also reachable under PSC. Importantly, as standard in DRF guarantees, it suffices to verify the absence of races *under* PSC. Thus, programmers can adhere to a safe programming discipline that is formulated solely in terms of PSC.

To facilitate the exposition, we start with a simplified version of the DRF guarantee, and later strengthen the theorem by further restricting the notion of a race. The strengthened theorem is instrumental in deriving a sound mapping of programs from PSC to $\mathsf{PTSO_{syn}}$, which can be followed by compilers to ensure PSC semantics under x86-TSO.

### 7.1 A Simplified DRF Guarantee

The premise of the DRF result requires the absence of two kinds of races: (i) races between a write/RMW operation and a read accessing the same location; and (ii) races between write/RMW operation and a flush-optimal instruction to the same location. Write-write races are allowed. Similarly, racy reads are only *plain* reads, and not "R-ex's" that arise from failed CAS operations. In particular, this ensures that standard locks, implemented using a CAS for acquiring the lock (in a spinloop) and a plain write for releasing the lock, are race free and can be safely used to avoid races in programs. This frees us from the need to have lock and unlock primitives (e.g., as in [Owens 2010]), and still obtain an applicable DRF guarantee.

For the formal statement of the theorem, we define races and racy programs.

*Definition 7.1.* Given a read or a flush-optimal label $l$, we say that thread $\tau$ *exhibits an $l$-race* in a program state $\overline{q} \in Pr.\mathtt{Q}$ if $\overline{q}(\tau)$ enables $l$, while there exists a thread $\tau_{\mathtt{W}} \neq \tau$ such that $\overline{q}(\tau_{\mathtt{W}})$ enables an event label $l_{\mathtt{W}}$ with $\mathtt{typ}(l_{\mathtt{W}}) \in \{\mathtt{W}, \mathtt{RMW}\}$ and $\mathtt{loc}(l_{\mathtt{W}}) = \mathtt{loc}(l)$.

*Definition 7.2.* A program $Pr$ is *racy* if for some program state $\overline{q} \in Pr.\mathtt{Q}$ that is reachable under PSC, some thread $\tau$ exhibits an $l$-race for some read or flush-optimal label $l$.

The above notion of racy programs is operational (we believe it may be more easily applicable by developers compared to a declarative notion). It requires that under PSC, the program $Pr$ can reach a state $\overline{q}$ possibly after multiple crashes, where $\overline{q}$ enables both a write/RMW by some thread $\tau_{\mathtt{W}}$ and a read/flush-optimal of the same location by some other thread $\tau$. As mentioned above, Def. 7.2 formulates a property of programs *under the* PSC *model.*

Theorem 7.3. *For a* non-racy *program $Pr$, a program state $\overline{q} \in Pr.\mathtt{Q}$ is reachable under* $\mathsf{PTSO_{syn}}$ *iff it is reachable under* PSC.

The theorem is a direct corollary of the more general result in Thm. 7.8 below. A simple corollary of Thm. 7.3 is that single-threaded programs (e.g., those in Ex. 3.3) cannot observe the difference between $\mathsf{PTSO_{syn}}$ and PSC (due to the non-FIFO propagation of flush-optimals in $\mathsf{PTSO_{syn}}$, even this is not completely trivial).

*Example 7.4.* Since $\mathsf{PTSO_{syn}}$ allows the propagation of flush-optimals before previously issued writes to different locations, it is essential to include races on flush-optimals in the definition above.

Indeed, if races between writes and flush-optimals are not counted, then the program on the right is clearly race free. However, the annotated persistent memory ($\mathtt{z} = \mathtt{w} = 1$ but $\mathtt{x} = \mathtt{y} = 0$) is reachable under $\mathsf{PTSO}_{\mathsf{syn}}$ (by propagating each flush-optimal before the prior write), but not under PSC.

```
x := 1 ;    || y := 1 ;
fo(y) ;     || fo(x) ;
sfence ;    || sfence ;
z := 1 ;    || w := 1 ;
```

## 7.2 A Generalized DRF Guarantee and a PSC to $\mathsf{PTSO}_{\mathsf{syn}}$ Mapping

We refine our definition of races to be sufficiently precise for deriving a mapping scheme from PSC to $\mathsf{PTSO}_{\mathsf{syn}}$ as a corollary of the DRF guarantee. To do so, reads and flush-optimals are only considered racy if they are *unprotected*, as defined next.

*Definition 7.5.* Let $\rho = l_1, \dots, l_n$ be a sequence of event labels.

- A read label $\mathtt{R}(x, \_)$ is *unprotected* after $\rho$ if there is some $1 \leq i_{\mathtt{W}} \leq n$ such that $l_{i_{\mathtt{W}}} = \mathtt{W}(y, \_)$ with $y \neq x$ and for every $i_{\mathtt{W}} < j \leq n$ we have $l_j \notin \{\mathtt{W}(x, \_), \mathtt{RMW}(\_, \_, \_), \mathtt{R\text{-}ex}(\_, \_), \mathtt{MF}\}$.
- A flush-optimal label $\mathtt{FO}(x)$ is *unprotected* after $\rho$ if there is some $1 \leq i_{\mathtt{W}} \leq n$ such that $l_{i_{\mathtt{W}}} = \mathtt{W}(y, \_)$ with $y \neq x$ and for every $i_{\mathtt{W}} < j \leq n$ we have $l_j \notin \{\mathtt{W}(x, \_), \mathtt{RMW}(\_, \_, \_), \mathtt{R\text{-}ex}(\_, \_), \mathtt{MF}, \mathtt{SF}\}$.

Roughly speaking, unprotected labels are induced by read/flush-optimal instructions of location $x$ that follow a write instruction to a different location with no barrier, which can be either an RMW instruction, an mfence, or a write to $x$, intervening in between. Flush-optimals are also protected if an sfence barrier is placed between that preceding write and the flush-optimal instruction.

Using the last definitions, we define *strongly racy* programs.

*Notation 7.6.* For an observable program traces $t$ and thread $\tau$, we denote by $\mathsf{suffix}_{\tau}(t)$ the sequence of event labels corresponding to the maximal crashless suffix of $t|_{\tau}$ (i.e., $\mathsf{suffix}_{\tau}(t) = l_1, \dots, l_n$ when $\langle \tau, l_1 \rangle, \dots, \langle \tau, l_n \rangle$ is the maximal crashless suffix of the restriction of $t$ to transition labels of the form $\langle \tau, \_ \rangle$).

*Definition 7.7.* A program $Pr$ is *strongly racy* if there exist $\overline{q} \in Pr.\mathtt{Q}$, trace $t$, thread $\tau$, and a read or a flush-optimal label $l$ such that the following hold:

- $\overline{q}$ is reachable under PSC via the trace $t$ (i.e., $\langle \overline{q}_{\mathsf{Init}}, m_{\mathsf{Init}}, P_{\epsilon} \rangle \overset{t}{\Rightarrow}_{Pr \shortparallel \mathsf{PSC}} \langle \overline{q}, m, P \rangle$ for some $\overline{q}_{\mathsf{Init}} \in Pr.\mathtt{Q}_{\mathsf{Init}}$ and $\langle m, P \rangle \in \mathsf{PSC}.\mathtt{Q}$).
- $\tau$ exhibits an $l$-race in $\overline{q}$.
- $l$ is unprotected after $\mathsf{suffix}_{\tau}(t)$.

The generalized DRF result is stated in the next theorem.

Theorem 7.8. *For a program $Pr$ that is not strongly racy, a program state $\overline{q} \in Pr.\mathtt{Q}$ is reachable under* $\mathsf{PTSO}_{\mathsf{syn}}$ *iff it is reachable under* PSC.

Example 4.4 is an example of a program that is racy but not strongly racy. By Thm. 7.8, that program has only PSC-behaviors. Example 4.3 can be made not strongly racy: by adding an sfence instruction between $\mathtt{y} := 2$ and $\mathtt{fo}(\mathtt{x})$; by strengthening $\mathtt{fo}(\mathtt{x})$ to $\mathtt{fl}(\mathtt{x})$; *or* by replacing $\mathtt{y} := 2$ with an atomic exchange instruction (an RMW).

An immediate corollary of Thm. 7.8 is that programs that only use RMWs when writing to shared locations (e.g., [Morrison and Afek 2013]) may safely assume PSC semantics (all labels will be protected). More generally, by "protecting" all racy reads and flush-optimals, we can transform a given program and make it non-racy according to the definition above. In other words, we obtain a compilation scheme from a language with PSC semantics to x86. Since precise static analysis of races is hard, such scheme may over-approximate. Concretely, a sound scheme can:

(i) like the standard compilation from SC to TSO [Mapping 2019], place *mfences* separating all read-after-write pairs of different locations (when there is no RMW already in between); and

(ii) place *sfences* separating all flush-optimal-after-write pairs of different locations (when there is no RMW or other sfence already in between).

Moreover, since a write to $x$ between a write to some location $y \neq x$ and a flush-optimal to $x$ makes the flush protected, in the standard case where flush-optimal to some location $x$ immediately follows a write to $x$ (for ensuring a persistence order for that write), flush-optimals can be compiled without additional barriers. Similarly, the other standard use of a flush-optimal to $x$ after reading from $x$ (known as "flush-on-read" for ensuring a persistence order for writes that the thread relies on) does not require additional barriers as well—an mfence is anyway placed between writes to locations different than $x$ and the read from $x$ that precedes the flush-optimal. Thus, we believe that for most "real-world" programs the above scheme will not incur additional runtime overhead compared standard mappings from SC to x86 (see, e.g., [Liu et al. 2017; Marino et al. 2011; Singh et al. 2012] for performance studies).

To prove Thm. 7.8 we use the declarative formulations of $\mathsf{PTSO_{syn}}$ and PSC. First, we relate unprotected labels as defined in Def. 7.5 with unprotected events in the corresponding execution graph, as defined next.

*Definition 7.9.* Let $G$ be an execution graph. An event $e \in \mathsf{R} \cup \mathsf{FO}$ with $x = \mathtt{loc}(e)$ is *$G$-unprotected* if one of the following holds:

- $e \in G.\mathsf{R}$ and $\langle w, e\rangle \in G.\mathsf{po} \setminus (G.\mathsf{po} \,;[\mathsf{W}_x \cup \mathsf{RMW} \cup \mathsf{R\text{-}ex} \cup \mathsf{MF}]\,; G.\mathsf{po})$ for some $w \in \mathsf{W} \setminus \mathsf{Init}$ with $\mathtt{loc}(w) \neq x$.
- $e \in G.\mathsf{FO}$ and $\langle w, e\rangle \in G.\mathsf{po} \setminus (G.\mathsf{po} \,; [\mathsf{W}_x \cup \mathsf{RMW} \cup \mathsf{R\text{-}ex} \cup \mathsf{MF} \cup \mathsf{SF}] \,; G.\mathsf{po})$ for some $w \in \mathsf{W} \setminus \mathsf{Init}$ with $\mathtt{loc}(w) \neq x$.

Proposition 7.10. *Let $\tau \in \mathsf{Tid}$. Let $G$ and $G'$ be execution graphs such that $G'.\mathsf{E}^\tau = G.\mathsf{E}^\tau \cup \{e\}$ for some $G.\mathsf{po} \cup G.\mathtt{rf}$-maximal event $e$. If $e$ is $G'$-unprotected, then $\mathtt{lab}(e)$ is unprotected after $\mathsf{suffix}_\tau(t)$ for some observable program trace $t \in \mathsf{traces}(G)$.*

The next key lemma, establishing the DRF-guarantee "on the execution graph level", is needed for proving Thm. 7.8. Its proof utilizes $\mathsf{DPTSO^{mo}_{syn}}$, which is closer to DPSC than $\mathsf{DPTSO_{syn}}$.

Lemma 7.11. *Let $G$ be a $\mathsf{DPTSO_{syn}}$-consistent execution graph. Suppose that for every $w \in G.\mathsf{W} \cup G.\mathsf{RMW}$ and $G$-unprotected event $e \in \mathsf{R}_{\mathtt{loc}(w)} \cup \mathsf{FO}_{\mathtt{loc}(w)}$, we have either $\langle w, e\rangle \in (G.\mathsf{po} \cup G.\mathtt{rf})^+$ or $\langle e, w\rangle \in (G.\mathsf{po} \cup G.\mathtt{rf})^+$. Then, $G$ is DPSC-consistent.*

With Lemma 7.11, the proof of Thm. 7.8 extends the standard declarative DRF argument. Roughly speaking, we consider the first DPSC-inconsistent execution graph encountered in a chain of execution graphs for reaching a certain program state. Then, we show that a minimal DPSC-inconsistent prefix of that graph must entail a strong race as defined in Def. 7.7.

## 8 CONCLUSION AND RELATED WORK

We have presented an alternative x86-TSO persistency model, called $\mathsf{PTSO_{syn}}$, formulated it operationally and declaratively, and proved it to be observationally equivalent to Px86 when observations consist of reachable program states and non-volatile memories. To the best of our understanding, $\mathsf{PTSO_{syn}}$ captures the intuitive persistence guarantees (of flush-optimal and sfence instructions, in particular) widely present in the literature on data-structure design as well as on programming persistent memory (see [Intel 2015; Intel 2019; Scargall 2020]). We have also presented a formalization of an SC-based persistency model, called PSC, which is simpler and stronger than $\mathsf{PTSO_{syn}}$, and related it to $\mathsf{PTSO_{syn}}$ via a sound compilation scheme and a DRF-guarantee. We believe that the developments of data structures and language-level persistency constructs for non-volatile memory, such as listed in §1, may adopt $\mathsf{PTSO_{syn}}$ and PSC as their formal semantic foundations.

Our models may also simplify reasoning about persistency under x86-TSO both for programmers and automated verification tools.

We have already discussed in length the relation of our work to [Raad et al. 2020]. Next, we describe the relation to several other related work.

Pelley et al. [2014] (informally) explore a hardware co-design for memory persistency and memory consistency and propose a model of *epoch persistency* under sequential consistency, which splits thread executions into epochs with special *persist barriers*, so that the order of persistence is only enforced for writes from different epochs. Condit et al. [2009]; Joshi et al. [2015] propose hardware implementations for persist barriers to enable epoch persistency under x86-TSO. While x86-TSO does not provide a persist barrier, flush-optimals combined with an sfence instruction could be used to this end.

Kolli et al. [2016] conducted the first analysis of persistency under x86. They described the semantics induced by the use of CLWB and sfence instructions as *synchronous*, reaffirming our observation about the common understanding of persistency models. The PTSO model [Raad and Vafeiadis 2018], which was published before Px86, is a proposal for integrating epoch persistency with the x86-TSO semantics. It has synchronous explicit persist instructions and per-location persistence buffers like our $\mathsf{PTSO}_{\mathsf{syn}}$ model, but it is more complex (its persistence buffers are queues of sub-buffers, each of which records pending writes of a given epoch), and uses coarse-grained instructions for persisting *all* pending writes, which were deprecated in x86 [Rudoff 2019].

Kolli et al. [2017] propose a declarative language-level *acquire-release persistency* model offering new abstractions for programming for persistent memory in C/C++. In comparison, our work aims at providing a formal foundation for reasoning about the underlying architecture. Gogte et al. [2018] improved the model of [Kolli et al. 2017] by proposing a generic logging mechanism for synchronization-free regions that aims to achieve failure atomicity for data-race-free programs. We conjecture that our results (in particular our DRF guarantee relating $\mathsf{PTSO}_{\mathsf{syn}}$ and PSC) can serve as a semantic foundation in formally proving the failure-atomicity properties of their implementation.

Raad et al. [2019] proposed a general declarative framework for specifying persistency semantics and formulated a persistency model for ARM in this framework (which is less expressive than in x86). Our declarative models follow their framework, accounting for a specific outcomes using chains of execution graphs, but we refrain from employing an additional "non-volatile-order" for tracking the order in which stores are committed to the non-volatile memory. Instead, in the spirit of a theoretical model of [Izraelevitz et al. 2016b], which gives a declarative semantics of epoch persistency under release consistency (assuming both an analogue of the synchronous sfence and also an analogue of a deprecated coarse-grained flush instruction), we track the last persisted write for each location, and use it to derive constraints on existing partial orders. Thus, we believe that our declarative model is more abstract, and may provide a suitable basis for partial order reduction verification techniques (e.g., [Abdulla et al. 2018; Kokologiannakis et al. 2017]).

Finally, the decidability of reachability under Px86 was investigated in [Abdulla et al. 2021]. Using load buffers instead of store buffers, the authors presented a rather intricate model that is equivalent to Px86 and can be used in the framework of well-structured transition systems for establishing the decidability of reachability.

## ACKNOWLEDGMENTS

We thank Adam Morrison and the POPL'21 reviewers for their helpful feedback and insights. This research was supported by the Israel Science Foundation (grant number 5166651 and 2005/17). The second author was also supported by the Alon Young Faculty Fellowship.

## REFERENCES

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Ahmed Bouajjani, K. Narayan Kumar, and Prakash Saivasan. 2021. Deciding Reachability under Persistent x86-TSO. *Proc. ACM Program. Lang.* 5, POPL, Article 56 (Jan. 2021), 32 pages. https://doi.org/10.1145/3434337

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Tuan Phong Ngo. 2018. Optimal Stateless Model Checking under the Release-Acquire Semantics. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 135 (Oct. 2018), 29 pages. https://doi.org/10.1145/3276505

Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Trans. Program. Lang. Syst.* 36, 2, Article 7 (July 2014), 74 pages. https://doi.org/10.1145/2627752

Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. 2018. Bztree: A High-Performance Latch-Free Range Index for Non-Volatile Memory. *Proc. VLDB Endow.* 11, 5 (Jan. 2018), 553–565. https://doi.org/10.1145/3164135.3164147

Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. 2010. On the Verification Problem for Weak Memory Models. In *POPL*. ACM, New York, NY, USA, 7–18. https://doi.org/10.1145/1706299.1706303

Kumud Bhandari, Dhruva R. Chakrabarti, and Hans-Juergen Boehm. 2012. *Implications of CPU Caching on Byte-addressable Non-Volatile Memory Programming.* Technical Report HPL-2012-236. Hewlett-Packard.

Shimin Chen and Qin Jin. 2015. Persistent B+-Trees in Non-Volatile Main Memory. *Proc. VLDB Endow.* 8, 7 (Feb. 2015), 786–797. https://doi.org/10.14778/2752939.2752947

Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O Through Byte-addressable, Persistent Memory. In *SOSP*. ACM, New York, NY, USA, 133–146. https://doi.org/10.1145/1629575.1629589

Tudor David, Aleksandar Dragojević, Rachid Guerraoui, and Igor Zablotchi. 2018. Log-Free Concurrent Data Structures. In *USENIX ATC*. USENIX Association, USA, 373–385.

Michal Friedman, Naama Ben-David, Yuanhao Wei, Guy E. Blelloch, and Erez Petrank. 2020. NVTraverse: In NVRAM Data Structures, the Destination is More Important than the Journey. In *PLDI*. ACM, New York, NY, USA, 377–392. https://doi.org/10.1145/3385412.3386031

Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. 2018. A Persistent Lock-free Queue for Non-volatile Memory. In *PPoPP*. ACM, New York, NY, USA, 28–40. https://doi.org/10.1145/3178487.3178490

Vaibhav Gogte, Stephan Diestelhorst, William Wang, Satish Narayanasamy, Peter M. Chen, and Thomas F. Wenisch. 2018. Persistency for Synchronization-free Regions. In *PLDI*. ACM, New York, NY, USA, 46–61. https://doi.org/10.1145/3192366.3192367

Intel. 2015. Persistent Memory Programming. http://pmem.io/

Intel. 2019. Intel 64 and IA-32 Architectures Software Developer's Manual (Combined Volumes). https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf Order Number: 325462-069US.

Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. 2016a. Brief Announcement: Preserving Happens-before in Persistent Memory. In *SPAA*. ACM, New York, NY, USA, 157–159. https://doi.org/10.1145/2935764.2935810

Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. 2016b. Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model. In *DISC*. Springer Berlin Heidelberg, Berlin, Heidelberg, 313–327.

Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. 2015. Efficient Persist Barriers for Multicores. In *MICRO*. ACM, New York, NY, USA, 660–671. https://doi.org/10.1145/2830772.2830805

Artem Khyzha and Ori Lahav. 2020. Taming x86-TSO Persistency (Extended Version). https://arxiv.org/abs/2010.13593

Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. 2017. Effective Stateless Model Checking for C/C++ Concurrency. *Proc. ACM Program. Lang.* 2, POPL, Article 17 (Dec. 2017), 32 pages. https://doi.org/10.1145/3158105

Aasheesh Kolli, Vaibhav Gogte, Ali Saidi, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. 2017. Language-level Persistency. In *ISCA*. ACM, New York, NY, USA, 481–493. https://doi.org/10.1145/3079856.3080229

Aasheesh Kolli, Jeff Rosen, Stephan Diestelhorst, Ali Saidi, Steven Pelley, Sihang Liu, Peter M. Chen, and Thomas F. Wenisch. 2016. Delegated Persist Ordering. In *MICRO*. IEEE Press, Piscataway, NJ, USA, Article 58, 13 pages. http://dl.acm.org/citation.cfm?id=3195638.3195709

Dexter Kozen. 1977. Lower bounds for natural proof systems. In *SFCS*. IEEE Computer Society, Washington, 254–266. https://doi.org/10.1109/SFCS.1977.16

Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. 2016. Taming Release-Acquire Consistency. In *POPL*. ACM, New York, NY, USA, 649–662. https://doi.org/10.1145/2837614.2837643

Lucas Lersch, Xiangpeng Hao, Ismail Oukid, Tianzheng Wang, and Thomas Willhalm. 2019. Evaluating Persistent Memory Range Indexes. *Proc. VLDB Endow.* 13, 4 (Dec. 2019), 574–587. https://doi.org/10.14778/3372716.3372728

Jihang Liu, Shimin Chen, and Lujun Wang. 2020. LB+Trees: Optimizing Persistent Index Performance on 3DXPoint Memory. *Proc. VLDB Endow.* 13, 7 (March 2020), 1078–1090. https://doi.org/10.14778/3384345.3384355

Lun Liu, Todd Millstein, and Madanlal Musuvathi. 2017. A Volatile-by-Default JVM for Server Applications. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 49 (Oct. 2017), 25 pages. https://doi.org/10.1145/3133873

Mapping 2019. C/C++11 mappings to processors. Retrieved July 3, 2019 from http://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html

Daniel Marino, Abhayendra Singh, Todd Millstein, Madanlal Musuvathi, and Satish Narayanasamy. 2011. A Case for an SC-Preserving Compiler. In *PLDI*. ACM, New York, NY, USA, 199–210. https://doi.org/10.1145/1993498.1993522

Adam Morrison and Yehuda Afek. 2013. Fast Concurrent Queues for X86 Processors. In *PPoPP*. ACM, New York, NY, USA, 103–112. https://doi.org/10.1145/2442516.2442527

Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *SIGMOD*. ACM, New York, NY, USA, 371–386. https://doi.org/10.1145/2882903.2915251

Scott Owens. 2010. Reasoning About the Implementation of Concurrency Abstractions on x86-TSO. In *ECOOP*. Springer-Verlag, Berlin, Heidelberg, 478–503. http://dl.acm.org/citation.cfm?id=1883978.1884011

Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A Better x86 Memory Model: x86-TSO. In *TPHOLs*. Springer, Heidelberg, 391–407. https://doi.org/10.1007/978-3-642-03359-9_27

Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. 2014. Memory Persistency. In *ISCA*. IEEE Press, Piscataway, NJ, USA, 265–276. http://dl.acm.org/citation.cfm?id=2665671.2665712

Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. 2019. Bridging the Gap Between Programming Languages and Hardware Weak Memory Models. *Proc. ACM Program. Lang.* 3, POPL, Article 69 (Jan. 2019), 31 pages. https://doi.org/10.1145/3290382

Azalea Raad and Viktor Vafeiadis. 2018. Persistence Semantics for Weak Memory: Integrating Epoch Persistency with the TSO Memory Model. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 137 (Oct. 2018), 27 pages. https://doi.org/10.1145/3276507

Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis. 2020. Persistency Semantics of the Intel-x86 Architecture. *Proc. ACM Program. Lang.* 4, POPL, Article 11 (Jan. 2020), 31 pages. https://doi.org/10.1145/3371079

Azalea Raad, John Wickerson, and Viktor Vafeiadis. 2019. Weak Persistency Semantics from the Ground Up: Formalising the Persistency Semantics of ARMv8 and Transactional Models. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 135 (Oct. 2019), 27 pages. https://doi.org/10.1145/3360561

Andy M. Rudoff. 2019. Deprecating the PCOMMIT Instruction. https://software.intel.com/content/www/us/en/develop/blogs/deprecate-pcommit-instruction.html

Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams. 2012. Synchronising C/C++ and POWER. In *PLDI*. ACM, New York, NY, USA, 311–322. https://doi.org/10.1145/2254064.2254102

Steve Scargall. 2020. *Programming Persistent Memory: A Comprehensive Guide for Developers*. Apress Media, LLC. https://doi.org/10.1007/978-1-4842-4932-1

Abhayendra Singh, Satish Narayanasamy, Daniel Marino, Todd Millstein, and Madanlal Musuvathi. 2012. End-to-End Sequential Consistency. *SIGARCH Comput. Archit. News* 40, 3 (June 2012), 524–535. https://doi.org/10.1145/2366231.2337220

Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. 2015. Common Compiler Optimisations are Invalid in the C11 Memory Model and what we can do about it. In *POPL*. ACM, New York, NY, USA, 209–220. https://doi.org/10.1145/2676726.2676995

Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. 2011. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory. In *FAST*. USENIX Association, USA, 5.

Tianzheng Wang, Justin J. Levandoski, and Per-Åke Larson. 2018. Easy Lock-Free Indexing in Non-Volatile Memory. In *ICDE*. IEEE Computer Society, Los Alamitos, CA, USA, 461–472. https://doi.org/10.1109/ICDE.2018.00049

John Wickerson, Mark Batty, Tyler Sorensen, and George A. Constantinides. 2017. Automatically Comparing Memory Consistency Models. In *POPL*. ACM, New York, NY, USA, 190–204. https://doi.org/10.1145/3009837.3009838

Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. 2015. NV-Tree: Reducing Consistency Cost for NVM-Based Single Level Systems. In *FAST*. USENIX Association, USA, 167–181.

Yoav Zuriel, Michal Friedman, Gali Sheffi, Nachshon Cohen, and Erez Petrank. 2019. Efficient Lock-free Durable Sets. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 128 (Oct. 2019), 26 pages. https://doi.org/10.1145/3360554

## A PROOFS FOR SECTION 2

Proposition A.1. *For every observable program trace $t$ with $\lightning \notin t$:*

$$\langle \overline{q}, m, \tilde{m}\rangle \xRightarrow{t}_{Pr \parallel M} \langle \overline{q}', m', \tilde{m}'\rangle \iff (\overline{q} \xRightarrow{t}_{Pr} \overline{q}' \land \langle m, \tilde{m}\rangle \xRightarrow{t}_{M} \langle m', \tilde{m}'\rangle)$$

The proposition follows immediately from Definition 2.6.

Lemma 2.10. *The following conditions together ensure that a persistent memory subsystem $M_1$ observationally refines a persistent memory subsystem $M_2$:*

*(i) Every $m_0$-initialized $M_1$-observable-trace is also an $m_0$-initialized $M_2$-observable-trace.*
*(ii) For every $m_0$-to-$m$ $M_1$-observable-trace $t_1$, some $t_2 \lesssim t_1$ is an $m_0$-to-$m$ $M_2$-observable-trace.*

Proof. Suppose that $\overline{q} \in Pr.\mathsf{Q}$ is reachable under $M_1$. Then, by Def. 2.7, $\langle \overline{q}, m, \tilde{m}\rangle$ is reachable in $Pr \parallel M_1$ for some $\langle m, \tilde{m}\rangle \in M_1.\mathsf{Q}$. Thus, there exist crashless observable program traces $t_0, \dots, t_n$, initial program states $\overline{q}_0, \dots, \overline{q}_n \in Pr.\mathsf{Q}_{\mathsf{Init}}$, initial non-volatile memories $m_1, \dots, m_n \in \mathsf{Loc} \to \mathsf{Val}$, and initial volatile states $\tilde{m}_0, \dots, \tilde{m}_n \in M_1.\tilde{\mathsf{Q}}_{\mathsf{Init}}$, such that the following hold:

- $\langle \overline{q}_0, m_{\mathsf{Init}}, \tilde{m}_0\rangle \xRightarrow{t_0}_{Pr \parallel M_1} \langle \_, m_1, \_\rangle$, and $\langle \overline{q}_i, m_i, \tilde{m}_i\rangle \xRightarrow{t_i}_{Pr \parallel M_1} \langle \_, m_{i+1}, \_\rangle$ for every $1 \leq i \leq n-1$.
- $\langle \overline{q}_n, m_n, \tilde{m}_n\rangle \xRightarrow{t_n}_{Pr \parallel M_1} \langle \overline{q}, \_, \_\rangle$.

By Prop. A.1, it follows that:

- $\overline{q}_i \xRightarrow{t_i}_{Pr} \_$ for every $0 \leq i \leq n-1$, and $\overline{q}_n \xRightarrow{t_n}_{Pr} \overline{q}$.
- $t_0$ is an $m_{\mathsf{Init}}$-to-$m$ $M_1$-observable-trace, and $t_i$ is an $m_i$-to-$m_{i+1}$ $M_1$-observable-trace for every $1 \leq i \leq n-1$.
- $t_n$ is an $m_n$-initialized $M_1$-observable-trace.

Then, assumption (ii) entails that there exist $t'_0, \dots, t'_{n-1}$ such that the following hold:

- $t'_i \lesssim t_i$ for every $0 \leq i \leq n-1$.
- $t'_0$ is a $m_{\mathsf{Init}}$-to-$m_1$ $M_2$-observable-trace, and $t'_i$ is an $m_i$-to-$m_{i+1}$ $M_2$-observable-trace for every $1 \leq i \leq n-1$.

Therefore, there exist initial volatile states $\tilde{m}'_0, \dots, \tilde{m}'_{n-1} \in M_2.\tilde{\mathsf{Q}}_{\mathsf{Init}}$ such that:

$$\langle m_{\mathsf{Init}}, \tilde{m}'_0\rangle \xRightarrow{t'_0}_{M_2} \langle m_1, \_\rangle \text{ and } \langle m_i, \tilde{m}'_i\rangle \xRightarrow{t'_i}_{M_2} \langle m_{i+1}, \_\rangle \text{ for every } 1 \leq i \leq n-1.$$

Now, since $\overline{q}_i \xRightarrow{t_i}_{Pr} \_$ and $t'_i \lesssim t_i$ for every $0 \leq i \leq n-1$, by Prop. 2.4, we have $\overline{q}_i \xRightarrow{t'_i}_{Pr} \_$ for every $0 \leq i \leq n-1$. By Prop. A.1, it follows that:

$$\langle \overline{q}_0, m_{\mathsf{Init}}, \tilde{m}'_0\rangle \xRightarrow{t'_0}_{Pr \parallel M_2} \langle \_, m_1, \_\rangle \text{ and } \langle \overline{q}_i, m_i, \tilde{m}'_i\rangle \xRightarrow{t'_i}_{Pr \parallel M_2} \langle \_, m_{i+1}, \_\rangle \text{ for every } 1 \leq i \leq n-1 \qquad (1)$$

In addition, assumption (i) entails that $t_n$ is an $m_n$-initialized $M_2$-observable-trace. Therefore, there exists $\tilde{m}'_n \in M_2.\tilde{\mathsf{Q}}_{\mathsf{Init}}$ such that $\langle m_n, \tilde{m}'_n\rangle \xRightarrow{t_n}_{Pr \parallel M_2} \langle \_, \_\rangle$. Knowing that $\overline{q}_n \xRightarrow{t_n}_{Pr} \overline{q}$ holds, we conclude:

$$\langle \overline{q}_n, m_n, \tilde{m}'_n\rangle \xRightarrow{t_n}_{Pr \parallel M_2} \langle \overline{q}, \_, \_\rangle \qquad (2)$$

Putting Eq. (1) and Eq. (2) together, we have shown that there exist $t' = t'_0 \cdot \lightning \cdot \ldots \cdot \lightning \cdot t'_{n-1} \cdot \lightning \cdot t_n$ and $\tilde{m}'_0, \dots, \tilde{m}'_n \in M_2.\tilde{\mathsf{Q}}_{\mathsf{Init}}$ such that:

$$\langle \overline{q}_0, m_{\mathsf{Init}}, \tilde{m}'_0\rangle \xRightarrow{t'_0}_{Pr \parallel M_2} \langle \_, m_1, \_\rangle \xrightarrow{\lightning}_{Pr \parallel M_2} \langle \overline{q}_1, m_1, \tilde{m}'_1\rangle \xRightarrow{t'_1}_{Pr \parallel M_2} \cdots$$

$$\cdots \xRightarrow{t'_{n-1}}_{Pr \parallel M_2} \langle \_, m_n, \_\rangle \xrightarrow{\lightning}_{Pr \parallel M_2} \langle \overline{q}_n, m_n, \tilde{m}'_n\rangle \xRightarrow{t_n}_{Pr \parallel M_2} \langle \overline{q}, \_, \_\rangle,$$

meaning that $\overline{q}$ is reachable for $Pr$ under the persistent memory subsystem $M_2$. □

## B PROOFS FOR SECTION 4

To carry out our equivalence proofs we use *instrumented* versions of Px86 and $\mathsf{PTSO}_{\mathsf{syn}}$. We also introduce two additional (instrumented) persistent memory subsystems, $i\mathsf{PTSO}_1$ and $i\mathsf{PTSO}_2$, that serve as intermediate systems in our proof. In Appendix B.1 we generally define instrumented persistent memory subsystems. In Appendix B.2 we present the instrumented version of Px86. In Appendix B.3 we present $i\mathsf{PTSO}_1$ and $i\mathsf{PTSO}_2$. In Appendix B.4 we present the instrumented version of $\mathsf{PTSO}_{\mathsf{syn}}$. In Appendix B.5 we use these subsystems to establish the proof of Thm. 4.6. Finally, in Appendix B.6 we provide the proof of Lemma 4.5.

### B.1 Instrumented Persistent Memory Subsystems

Instrumented persistent memory subsystems are defined similarly to persistent memory subsystems, except for their transition labels (the alphabet of the LTS), which carry more information. In particular, the observable transition labels of the form $\langle \tau, l \rangle$ of persistent memory subsystems are augmented with an identifier $s \in \mathbb{N}$, which uniquely identifies the transition. The $\epsilon$-labels of silent transitions of persistent memory subsystems are made more informative as well. Hence, the transition labels of an instrumented persistent memory subsystem $iM$ consists of transition labels of the form $\langle \tau, l\#s \rangle$ (where $\tau \in \mathsf{Tid}$ and $l \in \mathsf{Lab}$) as well as a set denoted by $iM.i\Sigma$ of *instrumented silent transition labels*, which differs from one system to another. We assume that, like the instrumented non-silent transition labels, the instrumented silent transition labels also include an identifier $s \in \mathbb{N}$. We use the function $\#(\cdot)$ to retrieve this identifier from a given instrumented (silent or non-silent) transition label.

In the sequel, we use the same definition style and terminology that we used for persistent memory subsystems also in the context of instrumented persistent memory subsystems (e.g., defining only the volatile component of the state).

The following *erasure* function $\Lambda$ forgets the instrumentation in the transition labels.

*Definition B.1.* For a transition label $\alpha$ of an instrumented persistent memory subsystem $iM$, $\Lambda(\alpha)$ is defined as follows:

$$\Lambda(\alpha) \triangleq \begin{cases} \langle \tau, l \rangle & \alpha = \langle \tau, l\#s \rangle \\ \epsilon & \alpha \in iM.i\Sigma \end{cases}$$

The *erasure* of a trace $it$ of an instrumented persistent memory subsystem $iM$, denoted by $\Lambda(it)$, is the sequence obtained from $\Lambda(it(1)), \ldots, \Lambda(it(|it|))$ by omitting all $\epsilon$ labels.

As usual with instrumented operational semantics, it will be easy to see that the instrumentation does not affect the observable behaviors. Formally, we require the existence of an *erasure* (many-to-one) function from instrumented states to non-instrumented ones that satisfies certain conditions, as defined next.

*Definition B.2.* Let $M$ be a persistent memory subsystem and $iM$ be an instrumented persistent memory subsystem. A function $\Lambda : iM.\tilde{\mathsf{Q}} \to M.\tilde{\mathsf{Q}}$ is an *erasure function* if the following conditions hold:

- $M.\tilde{\mathsf{Q}}_{\mathsf{Init}} = \{\Lambda(\tilde{m}) \mid \tilde{m} \in iM.\tilde{\mathsf{Q}}_{\mathsf{Init}}\}$.
- If $\langle m, i\tilde{m} \rangle \xrightarrow{\tau, l\#s}_{iM} \langle m', i\tilde{m}' \rangle$, then $\langle m, \Lambda(i\tilde{m}) \rangle \xrightarrow{\tau, l}_{M} \langle m', \Lambda(i\tilde{m}') \rangle$.
- If $\langle m, i\tilde{m} \rangle \xrightarrow{\alpha}_{iM} \langle m', i\tilde{m}' \rangle$ for some $\alpha \in iM.i\Sigma$, then $\langle m, \Lambda(i\tilde{m}) \rangle \xrightarrow{\epsilon}_{M} \langle m', \Lambda(i\tilde{m}') \rangle$.
- If $\langle m, \Lambda(i\tilde{m}) \rangle \xrightarrow{\tau, l}_{M} \langle m', \tilde{m}' \rangle$, then $\langle m, i\tilde{m} \rangle \xrightarrow{\tau, l\#s}_{iM} \langle m', i\tilde{m}' \rangle$ for some $s \in \mathbb{N}$ and $i\tilde{m}' \in iM.\tilde{\mathsf{Q}}$ such that $\Lambda(i\tilde{m}') = \tilde{m}'$.
- If $\langle m, \Lambda(i\tilde{m}) \rangle \xrightarrow{\epsilon}_{M} \langle m', \tilde{m}' \rangle$, then $\langle m, i\tilde{m} \rangle \xrightarrow{\alpha}_{iM} \langle m', i\tilde{m}' \rangle$ for some $\alpha \in iM.i\Sigma$ and $i\tilde{m}' \in iM.\tilde{\mathsf{Q}}$ such that $\Lambda(i\tilde{m}') = \tilde{m}'$.

Given such function $\Lambda$, we say that $iM$ is a $\Lambda$-*instrumentation* of $M$. Furthermore, $iM$ is called an *instrumentation* of $M$ if it is a $\Lambda$-*instrumentation* of $M$ for some erasure function $\Lambda$.

Lemma B.3. *Let $iM$ be a $\Lambda$-instrumentation of a persistent memory subsystem $M$. Then, the following hold:*

- *For every $m_0, m \in \mathsf{Loc} \to \mathsf{Val}$, $i\tilde{m}_{\mathsf{Init}} \in iM.\tilde{\mathsf{Q}}_{\mathsf{Init}}$, $i\tilde{m} \in iM.\tilde{\mathsf{Q}}$, and $it$, if $\langle m_0, i\tilde{m}_{\mathsf{Init}}\rangle \xrightarrow{it}_{iM} \langle m, i\tilde{m}\rangle$, then $\langle m_0, \Lambda(\tilde{m}_{\mathsf{Init}})\rangle \xRightarrow{\Lambda(it)}_M \langle m, \Lambda(i\tilde{m})\rangle$.*
- *For every $m_0, m \in \mathsf{Loc} \to \mathsf{Val}$, $\tilde{m}_{\mathsf{Init}} \in M.\tilde{\mathsf{Q}}_{\mathsf{Init}}$, $\tilde{m} \in M.\tilde{\mathsf{Q}}$, and $t$, if $\langle m_0, \tilde{m}_{\mathsf{Init}}\rangle \xRightarrow{t}_M \langle m, \tilde{m}\rangle$, then $\langle m_0, i\tilde{m}_{\mathsf{Init}}\rangle \xrightarrow{it}_{iM} \langle m, i\tilde{m}\rangle$ for some $it$, $i\tilde{m}_{\mathsf{Init}} \in iM.\tilde{\mathsf{Q}}_{\mathsf{Init}}$, and $i\tilde{m} \in iM.\tilde{\mathsf{Q}}$ such that $\Lambda(it) = t$ and $\Lambda(i\tilde{m}) = \tilde{m}$.*

## B.2 *i*Px86: **Instrumented** Px86

The instrumented versions of our TSO-based persistent memory subsystems augment the entries of the persistent and store buffers with the identifier $s \in \mathbb{N}$ that was used in the label of the issuing step that added the entry to the buffer. For instance, we have entries of the form $\mathtt{W}(x, v)\#s$ in the persistence buffer instead of $\mathtt{W}(x, v)$; and $\mathtt{FL}(x)\#s$ in the store buffer instead of $\mathtt{FL}(x)$. Then, when propagating an entry with identifier $s$, we include $s$ in the instrumented silent transition label. This allows us to easily relate the transitions in which events are issued, propagated from store buffer, and persist. For instance, a write step generates a fresh identifier $s$ (included both in the transition label and in the new store buffer entry), that is (possibly) reused in a (exactly one) later PROP-W step, and further (possibly) reused in (exactly one) later PERSIST-W step.

*Definition B.4.* An *instrumented persistence buffer* is a finite sequence $ip$ of elements of the form $\alpha\#s$ where $\alpha$ is a persistence-buffer entry (of the form $\mathtt{W}(x, v)$ or $\mathtt{PER}(x)$) and $s \in \mathbb{N}$. An *instrumented store buffer* is a finite sequence $ib$ of elements of the form $\alpha\#s$ where $\alpha$ is a store-buffer entry (of the form $\mathtt{W}(x, v)$, $\mathtt{FL}(x)$, $\mathtt{FO}(x)$, or $\mathtt{SF}$) and $s \in \mathbb{N}$. An *instrumented store-buffer mapping* is a function $iB$ assigning an instrumented store buffer to every $\tau \in \mathsf{Tid}$.

*Definition B.5.* The *erasure of an instrumented persistence buffer* $ip$, denoted by $\Lambda(ip)$, is the persistence buffer obtained from $ip$ by omitting the identifier $s$ from all symbols. Similarly, the *erasure of an instrumented store buffer* $ib$, denoted by $\Lambda(ib)$, is the store buffer obtained from $ib$ by omitting the identifier $s$ from all symbols, and it is lifted to instrumented store-buffer mappings in the obvious way.

Using these definitions, *i*Px86 (*instrumented* Px86) is presented in Fig. 5. The functions `tid`, `typ`, `loc` are extended to *i*Px86.$i\Sigma$ in the obvious way (in particular, for $\alpha \in$ *i*Px86.$i\Sigma$, we have $\mathtt{typ}(\alpha) \in \{\mathtt{PropW}/\mathtt{PropFL}/\mathtt{PropFO}/\mathtt{PropSF}/\mathtt{PerW}/\mathtt{PerPER}\}$).

It is easy to see that *i*Px86 is an instrumentation of Px86.

Lemma B.6. *i*Px86 *is a $\Lambda$-instrumentation of* Px86 *for* $\Lambda \triangleq \lambda\langle ip, iB, S\rangle.\ \langle\Lambda(ip), \Lambda(iB)\rangle$.

$$
\begin{aligned}
i\text{Px86}.i\Sigma \triangleq{} & \{\langle\tau, \mathtt{PropW}(x)\#s\rangle \mid \tau \in \mathsf{Tid}, x \in \mathsf{Loc}, s \in \mathbb{N}\} \cup \{\langle\tau, \mathtt{PropFL}(x)\#s\rangle \mid \tau \in \mathsf{Tid}, x \in \mathsf{Loc}, s \in \mathbb{N}\} \\
& \cup \{\langle\tau, \mathtt{PropFO}(x)\#s\rangle \mid \tau \in \mathsf{Tid}, x \in \mathsf{Loc}, s \in \mathbb{N}\} \cup \{\langle\tau, \mathtt{PropSF}\#s\rangle \mid \tau \in \mathsf{Tid}, s \in \mathbb{N}\} \\
& \cup \{\mathtt{PerW}(x)\#s \mid x \in \mathsf{Loc}, s \in \mathbb{N}\} \cup \{\mathtt{PerPER}(x)\#s \mid x \in \mathsf{Loc}, s \in \mathbb{N}\}
\end{aligned}
$$

$$
m \in \mathsf{Loc} \rightarrow \mathsf{Val} \qquad ip \in (\{\mathtt{W}(x, v)\#s \mid x \in \mathsf{Loc}, v \in \mathsf{Val}, s \in \mathbb{N}\} \cup \{\mathtt{PER}(x)\#s \mid x \in \mathsf{Loc}, s \in \mathbb{N}\})^*
$$

$$
\begin{aligned}
iB \in \mathsf{Tid} \rightarrow (& \{\mathtt{W}(x, v)\#s \mid x \in \mathsf{Loc}, v \in \mathsf{Val}, s \in \mathbb{N}\} \cup \{\mathtt{FL}(x)\#s \mid x \in \mathsf{Loc}, \mathbb{N} \in \mathbb{N}\} \\
& \cup \{\mathtt{FO}(x)\#s \mid x \in \mathsf{Loc}, s \in \mathbb{N}\} \cup \{\mathtt{SF}\#s \mid s \in \mathbb{N}\})^* \qquad S \subseteq \mathbb{N}
\end{aligned}
$$

$$
ip_{\mathsf{Init}} \triangleq \epsilon \qquad iB_{\mathsf{Init}} \triangleq \lambda\tau.\ \epsilon \qquad S_{\mathsf{Init}} = \emptyset
$$

WRITE/FLUSH/FLUSH-OPT/SFENCE

$$
\frac{\begin{array}{c} S' = S \uplus \{s\} \\ \mathsf{typ}(l) \in \{\mathtt{W}, \mathtt{FL}, \mathtt{FO}, \mathtt{SF}\} \\ iB' = iB[\tau \mapsto iB(\tau) \cdot l\#s] \end{array}}{\langle m, ip, iB, S\rangle \xrightarrow{\tau, l\#s}_{i\mathsf{Px86}} \langle m, ip, iB', S'\rangle}
$$

READ

$$
\frac{\begin{array}{c} S' = S \uplus \{s\} \\ l = \mathtt{R}(x, v) \\ \mathsf{get}(m, \Lambda(ip), \Lambda(iB(\tau)))(x) = v \end{array}}{\langle m, ip, iB, S\rangle \xrightarrow{\tau, l\#s}_{i\mathsf{Px86}} \langle m, ip, iB, S'\rangle}
$$

RMW

$$
\frac{\begin{array}{c} S' = S \uplus \{s\} \\ l = \mathtt{RMW}(x, v_{\mathtt{R}}, v_{\mathtt{W}}) \\ \mathsf{get}(m, \Lambda(ip), \epsilon)(x) = v_{\mathtt{R}} \\ iB(\tau) = \epsilon \\ ip' = ip \cdot \mathtt{W}(x, v_{\mathtt{W}})\#s \end{array}}{\langle m, ip, iB, S\rangle \xrightarrow{\tau, l\#s}_{i\mathsf{Px86}} \langle m, ip', iB, S'\rangle}
$$

RMW-FAIL

$$
\frac{\begin{array}{c} S' = S \uplus \{s\} \\ l = \mathtt{R\text{-}ex}(x, v) \\ \mathsf{get}(m, \Lambda(ip), \epsilon)(x) = v \\ iB(\tau) = \epsilon \end{array}}{\langle m, ip, iB, S\rangle \xrightarrow{\tau, l\#s}_{i\mathsf{Px86}} \langle m, ip, iB, S'\rangle}
$$

MFENCE

$$
\frac{\begin{array}{c} S' = S \uplus \{s\} \\ l = \mathtt{MF} \\ \\ iB(\tau) = \epsilon \end{array}}{\langle m, ip, iB, S\rangle \xrightarrow{\tau, l\#s}_{i\mathsf{Px86}} \langle m, ip, iB, S'\rangle}
$$

PROP-W

$$
\frac{\begin{array}{c} L = \mathtt{PropW}(x)\#s \\ iB(\tau) = ib_1 \cdot \mathtt{W}(x, v)\#s \cdot ib_2 \\ \mathtt{W}(\_, \_)\#\_, \mathtt{FL}(\_)\#\_, \mathtt{SF}\#\_ \notin ib_1 \\ iB' = iB[\tau \mapsto ib_1 \cdot ib_2] \qquad ip' = ip \cdot \mathtt{W}(x, v)\#s \end{array}}{\langle m, ip, iB, S\rangle \xrightarrow{\tau, L}_{i\mathsf{Px86}} \langle m, ip', iB', S\rangle}
$$

PROP-FL

$$
\frac{\begin{array}{c} L = \mathtt{PropFL}(x)\#s \\ iB(\tau) = ib_1 \cdot \mathtt{FL}(x)\#s \cdot ib_2 \\ \mathtt{W}(\_, \_)\#\_, \mathtt{FL}(\_)\#\_, \mathtt{FO}(x)\#\_, \mathtt{SF}\#\_ \notin ib_1 \\ iB' = iB[\tau \mapsto ib_1 \cdot ib_2] \qquad ip' = ip \cdot \mathtt{PER}(x)\#s \end{array}}{\langle m, ip, iB, S\rangle \xrightarrow{\tau, L}_{i\mathsf{Px86}} \langle m, ip', iB', S\rangle}
$$

PROP-FO

$$
\frac{\begin{array}{c} L = \mathtt{PropFO}(x)\#s \\ iB(\tau) = ib_1 \cdot \mathtt{FO}(x)\#s \cdot ib_2 \\ \mathtt{W}(x, \_)\#\_, \mathtt{FL}(x)\#\_, \mathtt{SF}\#\_ \notin ib_1 \\ iB' = iB[\tau \mapsto ib_1 \cdot ib_2] \qquad ip' = ip \cdot \mathtt{PER}(x)\#s \end{array}}{\langle m, ip, iB, S\rangle \xrightarrow{\tau, L}_{i\mathsf{Px86}} \langle m, ip', iB', S\rangle}
$$

PROP-SF

$$
\frac{\begin{array}{c} L = \mathtt{PropSF}\#s \\ iB(\tau) = \mathtt{SF}\#s \cdot ib \\ \\ iB' = iB[\tau \mapsto ib] \end{array}}{\langle m, ip, iB, S\rangle \xrightarrow{\tau, L}_{i\mathsf{Px86}} \langle m, ip, iB', S\rangle}
$$

PERSIST-W

$$
\frac{\begin{array}{c} L = \mathtt{PerW}(x)\#s \\ ip = ip_1 \cdot \mathtt{W}(x, v)\#s \cdot ip_2 \\ \mathtt{W}(x, \_)\#\_, \mathtt{PER}(\_)\#\_ \notin ip_1 \\ ip' = ip_1 \cdot ip_2 \qquad m' = m[x \mapsto v] \end{array}}{\langle m, ip, iB, S\rangle \xrightarrow{L}_{i\mathsf{Px86}} \langle m', ip', iB, S\rangle}
$$

PERSIST-PER

$$
\frac{\begin{array}{c} L = \mathtt{PerPER}(x)\#s \\ ip = ip_1 \cdot \mathtt{PER}(x)\#s \cdot ip_2 \\ \mathtt{W}(x, \_)\#\_, \mathtt{PER}(\_)\#\_ \notin ip_1 \\ ip' = ip_1 \cdot ip_2 \end{array}}{\langle m, ip, iB, S\rangle \xrightarrow{L}_{i\mathsf{Px86}} \langle m, ip', iB, S\rangle}
$$

Fig. 5. The *i*Px86 Instrumented Persistent Memory Subsystem (the instrumentation is colored).

### B.3 Intermediate Systems $i$PTSO$_1$ and $i$PTSO$_2$

For the proof of equivalence of Px86 and PTSO$_{\mathsf{syn}}$, we use two intermediate instrumented persistent memory subsystems: $i$PTSO$_1$ and $i$PTSO$_2$. Next, we present these systems.

*Definition B.7.* An *instrumented per-location persistence buffer* is a finite sequence $ip$ of elements of the form $\alpha\#s$ where $\alpha$ is a per-location persistence buffer entry (of the form $\mathtt{W}(v)$ or $\mathtt{FO}(\tau)$) and $s \in \mathbb{N}$. An *instrumented per-location-persistence-buffer mapping* is a function $iP$ assigning an instrumented per-location persistence buffer to every $x \in \mathsf{Loc}$.

*Definition B.8.* The *erasure of an instrumented per-location persistence buffer* $ip$, denoted by $\Lambda(ip)$, is the per-location persistence buffer obtained from $ip$ by omitting the identifier $s$ from all symbols. It is lifted to instrumented per-location-persistence-buffer mappings in the obvious way.

$i$PTSO$_1$ is presented in Fig. 6. Note that the per-location-persistence-buffers of $i$PTSO$_1$ do not include $\mathtt{FO}(\tau)$-entries (these are used in the other systems below). The rules WRITE/FLUSH/FLUSH-OPT/SFENCE, MFENCE and PROP-SF are identical to the rules of $i$Px86. The rules READ, RMW, RMW-FAIL and PROP-W are analogous to those of $i$Px86 (they are trivially adjusted to operate with per-location persistence buffers).

The main feature of $i$PTSO$_1$ is that it makes all flush and flush-optimal instructions blocking. To this end, propagation of $\mathtt{FO}(x)$ and $\mathtt{FL}(x)$ is predicated upon $iP(x)$ being empty, and persistence steps for writes persist writes from the heads of the buffers.

$i$PTSO$_2$ is presented in Fig. 7. This instrumented persistent memory subsystem is similar to (the instrumented version of) PTSO$_{\mathsf{syn}}$ with the exception that its store buffers do not have the "almost" FIFO behavior of PTSO$_{\mathsf{syn}}$ and propagate entries out-of-order. We further highlight the differences w.r.t. $i$PTSO$_1$. Like $i$PTSO$_1$, PTSO$_2$ also has synchronous flush instructions, however, flush-optimal instructions are asynchronous. The PROP-FO transition is analogous to $i$Px86 (adjusted to the type of persistence buffers). PTSO$_2$ makes sfence instructions synchronous, as well as other serializing instructions, which results in RMW, RMW-FAIL, MFENCE and PROP-SF enforcing persistence of all flush-optimal instructions preceding the given one in program order as required by the constraint $(\forall y.\, \mathtt{FO}(\tau)\#\_ \notin iP(y))$. Finally, PERSIST-FO simply ensures that writes to a given location persist before the subsequent flush-optimal instruction.

### B.4 $i$PTSO$_{\mathsf{syn}}$: Instrumented PTSO$_{\mathsf{syn}}$

We will also need an instrumented version of PTSO$_{\mathsf{syn}}$, called $i$PTSO$_{\mathsf{syn}}$. This system is presented in Fig. 8. It is identical to $i$PTSO$_2$, except for some transitions (as highlighted in the figure). It is easy to see that $i$PTSO$_{\mathsf{syn}}$ is an instrumentation of PTSO$_{\mathsf{syn}}$.

LEMMA B.9. *$i$PTSO$_{\mathsf{syn}}$ is a $\Lambda$-instrumentation of PTSO$_{\mathsf{syn}}$ for $\Lambda \triangleq \lambda\langle iP, iB, S\rangle.\ \langle\Lambda(iP), \Lambda(iB)\rangle$.*

$$\begin{aligned}
i\mathrm{PTSO}_1.i\Sigma \triangleq{}& \{\langle \tau, \mathtt{PropW}(x)\#s\rangle \mid \tau \in \mathsf{Tid}, x \in \mathsf{Loc}, s \in \mathbb{N}\} \cup \{\langle \tau, \mathtt{PropFL}(x)\#s\rangle \mid \tau \in \mathsf{Tid}, x \in \mathsf{Loc}, s \in \mathbb{N}\}\\
&\cup \{\langle \tau, \mathtt{PropFO}(x)\#s\rangle \mid \tau \in \mathsf{Tid}, x \in \mathsf{Loc}, s \in \mathbb{N}\} \cup \{\langle \tau, \mathtt{PropSF}\#s\rangle \mid \tau \in \mathsf{Tid}, s \in \mathbb{N}\}\\
&\cup \{\mathtt{PerW}(x)\#s \mid x \in \mathsf{Loc}, s \in \mathbb{N}\}
\end{aligned}$$

$$m \in \mathsf{Loc} \rightarrow \mathsf{Val} \qquad iP \in \mathsf{Loc} \rightarrow \{\mathtt{W}(x, v)\#s \mid x \in \mathsf{Loc}, v \in \mathsf{Val}, s \in \mathbb{N}\}^*$$

$$\begin{aligned}
iB \in \mathsf{Tid} \rightarrow (&\{\mathtt{W}(x, v)\#s \mid x \in \mathsf{Loc}, v \in \mathsf{Val}, s \in \mathbb{N}\} \cup \{\mathtt{FL}(x)\#s \mid x \in \mathsf{Loc}, s \in \mathbb{N}\}\\
&\cup \{\mathtt{FO}(x)\#s \mid x \in \mathsf{Loc}, s \in \mathbb{N}\} \cup \{\mathtt{SF}\#s \mid s \in \mathbb{N}\})^* \qquad S \subseteq \mathbb{N}
\end{aligned}$$

$$iP_{\mathsf{Init}} \triangleq \lambda x.\ \epsilon \qquad iB_{\mathsf{Init}} \triangleq \lambda \tau.\ \epsilon \qquad S_{\mathsf{Init}} = \emptyset$$

WRITE/FLUSH/FLUSH-OPT/SFENCE

$$\frac{\begin{array}{c} S' = S \uplus \{s\} \\ \mathsf{typ}(l) \in \{\mathtt{W}, \mathtt{FL}, \mathtt{FO}, \mathtt{SF}\} \\ iB' = iB[\tau \mapsto iB(\tau) \cdot l\#s] \end{array}}{\langle m, iP, iB, S\rangle \xrightarrow{\tau, l\#s}_{i\mathrm{PTSO}_1} \langle m, iP, iB', S'\rangle}$$

READ

$$\frac{\begin{array}{c} S' = S \uplus \{s\} \\ l = \mathtt{R}(x, v) \\ \mathsf{get}(m, \Lambda(iP(x)), \Lambda(iB(\tau)))(x) = v \end{array}}{\langle m, iP, iB, S\rangle \xrightarrow{\tau, l\#s}_{i\mathrm{PTSO}_1} \langle m, iP, iB, S'\rangle}$$

RMW

$$\frac{\begin{array}{c} S' = S \uplus \{s\} \\ l = \mathtt{RMW}(x, v_{\mathrm{R}}, v_{\mathrm{W}}) \\ \mathsf{get}(m, \Lambda(iP(x)), \epsilon)(x) = v_{\mathrm{R}} \\ iB(\tau) = \epsilon \\ iP' = iP[x \mapsto iP(x) \cdot \mathtt{W}(v_{\mathrm{W}})\#s] \end{array}}{\langle m, iP, iB, S\rangle \xrightarrow{\tau, l\#s}_{i\mathrm{PTSO}_1} \langle m, iP', iB, S'\rangle}$$

RMW-FAIL

$$\frac{\begin{array}{c} S' = S \uplus \{s\} \\ l = \mathtt{R\text{-}ex}(x, v) \\ \mathsf{get}(m, \Lambda(iP(x)), \epsilon)(x) = v \\ iB(\tau) = \epsilon \end{array}}{\langle m, iP, iB, S\rangle \xrightarrow{\tau, l\#s}_{i\mathrm{PTSO}_1} \langle m, iP, iB, S'\rangle}$$

MFENCE

$$\frac{\begin{array}{c} S' = S \uplus \{s\} \\ l = \mathtt{MF} \\ iB(\tau) = \epsilon \end{array}}{\langle m, iP, iB, S\rangle \xrightarrow{\tau, l\#s}_{i\mathrm{PTSO}_1} \langle m, iP, iB, S'\rangle}$$

PROP-W

$$\frac{\begin{array}{c} L = \mathtt{PropW}(x)\#s \\ iB(\tau) = ib_1 \cdot \mathtt{W}(x, v)\#s \cdot ib_2 \\ \mathtt{W}(\_, \_)\#\_, \mathtt{FL}(\_)\#\_, \mathtt{SF}\#\_ \notin ib_1 \\ iB' = iB[\tau \mapsto ib_1 \cdot ib_2] \qquad iP' = iP[x \mapsto iP(x) \cdot \mathtt{W}(v)\#s] \end{array}}{\langle m, iP, iB, S\rangle \xrightarrow{\tau, L}_{i\mathrm{PTSO}_1} \langle m, iP', iB', S\rangle}$$

PROP-FL

$$\frac{\begin{array}{c} L = \mathtt{PropFL}(x)\#s \\ iB(\tau) = ib_1 \cdot \mathtt{FL}(x)\#s \cdot ib_2 \\ \mathtt{W}(\_, \_)\#\_, \mathtt{FL}(\_)\#\_, \mathtt{FO}(x)\#\_, \mathtt{SF}\#\_ \notin ib_1 \\ iP(x) = \epsilon \\ iB' = iB[\tau \mapsto ib_1 \cdot ib_2] \end{array}}{\langle m, iP, iB, S\rangle \xrightarrow{\tau, L}_{i\mathrm{PTSO}_1} \langle m, iP, iB', S\rangle}$$

PROP-FO

$$\frac{\begin{array}{c} L = \mathtt{PropFO}(x)\#s \\ iB(\tau) = ib_1 \cdot \mathtt{FO}(x)\#s \cdot ib_2 \\ \mathtt{W}(x, \_)\#\_, \mathtt{FL}(x)\#\_, \mathtt{SF}\#\_ \notin ib_1 \\ iP(x) = \epsilon \\ iB' = iB[\tau \mapsto ib_1 \cdot ib_2] \end{array}}{\langle m, iP, iB, S\rangle \xrightarrow{\tau, L}_{i\mathrm{PTSO}_1} \langle m, iP, iB', S\rangle}$$

PROP-SF

$$\frac{\begin{array}{c} L = \mathtt{PropSF}\#s \\ iB(\tau) = \mathtt{SF}\#s \cdot ib \\ iB' = iB[\tau \mapsto ib] \end{array}}{\langle m, iP, iB, S\rangle \xrightarrow{\tau, L}_{i\mathrm{PTSO}_1} \langle m, iP, iB', S\rangle}$$

PERSIST-W

$$\frac{\begin{array}{c} L = \mathtt{PerW}(x)\#s \\ iP(x) = \mathtt{W}(v)\#s \cdot ip \\ iP' = iP[x \mapsto ip] \qquad m' = m[x \mapsto v] \end{array}}{\langle m, iP, iB, S\rangle \xrightarrow{L}_{i\mathrm{PTSO}_1} \langle m', iP', iB, S\rangle}$$

Fig. 6. The $i$PTSO$_1$ Instrumented Persistent Memory Subsystem (differences w.r.t. $i$Px86 are highlighted)

$$
\begin{aligned}
i\text{PTSO}_2.i\Sigma \triangleq{} & \{\langle \tau, \mathtt{PropW}(x)\#s\rangle \mid \tau \in \mathsf{Tid}, x \in \mathsf{Loc}, s \in \mathbb{N}\} \cup \{\langle \tau, \mathtt{PropFL}(x)\#s\rangle \mid \tau \in \mathsf{Tid}, x \in \mathsf{Loc}, s \in \mathbb{N}\} \\
& \cup \{\langle \tau, \mathtt{PropFO}(x)\#s\rangle \mid \tau \in \mathsf{Tid}, x \in \mathsf{Loc}, s \in \mathbb{N}\} \cup \{\langle \tau, \mathtt{PropSF}\#s\rangle \mid \tau \in \mathsf{Tid}, s \in \mathbb{N}\} \\
& \cup \{\mathtt{PerW}(x)\#s \mid x \in \mathsf{Loc}, s \in \mathbb{N}\} \cup \{\mathtt{PerFO}(x)\#s \mid x \in \mathsf{Loc}, s \in \mathbb{N}\}
\end{aligned}
$$

$$
m \in \mathsf{Loc} \to \mathsf{Val} \qquad iP \in \mathsf{Loc} \to (\{\mathtt{W}(x, v)\#s \mid x \in \mathsf{Loc}, v \in \mathsf{Val}, s \in \mathbb{N}\} \cup \{\mathtt{FO}(\tau)\#s \mid \tau \in \mathsf{Tid}, s \in \mathbb{N}\})^*
$$

$$
\begin{aligned}
iB \in \mathsf{Tid} \to ( & \{\mathtt{W}(x, v)\#s \mid x \in \mathsf{Loc}, v \in \mathsf{Val}, s \in \mathbb{N}\} \cup \{\mathtt{FL}(x)\#s \mid x \in \mathsf{Loc}, s \in \mathbb{N}\} \\
& \cup \{\mathtt{FO}(x)\#s \mid x \in \mathsf{Loc}, s \in \mathbb{N}\} \cup \{\mathtt{SF}\#s \mid s \in \mathbb{N}\})^* \qquad S \subseteq \mathbb{N}
\end{aligned}
$$

$$
iP_{\mathsf{Init}} \triangleq \lambda x.\, \epsilon \qquad iB_{\mathsf{Init}} \triangleq \lambda \tau.\, \epsilon \qquad S_{\mathsf{Init}} = \emptyset
$$

WRITE/FLUSH/FLUSH-OPT/SFENCE

$$
\frac{\begin{array}{c} S' = S \uplus \{s\} \\ \mathsf{typ}(l) \in \{\mathtt{W}, \mathtt{FL}, \mathtt{FO}, \mathtt{SF}\} \\ iB' = iB[\tau \mapsto iB(\tau) \cdot l\#s] \end{array}}{\langle m, iP, iB, S\rangle \xrightarrow{\tau, l\#s}_{i\text{PTSO}_2} \langle m, iP, iB', S'\rangle}
$$

READ

$$
\frac{\begin{array}{c} S' = S \uplus \{s\} \\ l = \mathtt{R}(x, v) \\ \mathsf{get}(m, \Lambda(iP(x)), \Lambda(iB(\tau)))(x) = v \end{array}}{\langle m, iP, iB, S\rangle \xrightarrow{\tau, l\#s}_{i\text{PTSO}_2} \langle m, iP, iB, S'\rangle}
$$

RMW

$$
\frac{\begin{array}{c} S' = S \uplus \{s\} \\ l = \mathtt{RMW}(x, v_{\mathtt{R}}, v_{\mathtt{W}}) \\ \mathsf{get}(m, \Lambda(iP(x)), \epsilon)(x) = v_{\mathtt{R}} \\ iB(\tau) = \epsilon \\ \forall y.\ \mathtt{FO}(\tau)\#\_ \notin iP(y) \\ iP' = iP[x \mapsto iP(x) \cdot \mathtt{W}(v_{\mathtt{W}})\#s] \end{array}}{\langle m, iP, iB, S\rangle \xrightarrow{\tau, l\#s}_{i\text{PTSO}_2} \langle m, iP', iB, S'\rangle}
$$

RMW-FAIL

$$
\frac{\begin{array}{c} S' = S \uplus \{s\} \\ l = \mathtt{R\text{-}ex}(x, v) \\ \mathsf{get}(m, \Lambda(iP(x)), \epsilon)(x) = v \\ iB(\tau) = \epsilon \\ \forall y.\ \mathtt{FO}(\tau)\#\_ \notin iP(y) \end{array}}{\langle m, iP, iB, S\rangle \xrightarrow{\tau, l\#s}_{i\text{PTSO}_2} \langle m, iP, iB, S'\rangle}
$$

MFENCE

$$
\frac{\begin{array}{c} S' = S \uplus \{s\} \\ l = \mathtt{MF} \\ iB(\tau) = \epsilon \\ \forall y.\ \mathtt{FO}(\tau)\#\_ \notin iP(y) \end{array}}{\langle m, iP, iB, S\rangle \xrightarrow{\tau, l\#s}_{i\text{PTSO}_2} \langle m, iP, iB, S'\rangle}
$$

PROP-W

$$
\frac{\begin{array}{c} L = \mathtt{PropW}(x)\#s \\ iB(\tau) = ib_1 \cdot \mathtt{W}(x, v)\#s \cdot ib_2 \\ \mathtt{W}(\_, \_)\#\_, \mathtt{FL}(\_)\#\_, \mathtt{SF}\#\_ \notin ib_1 \\ iB' = iB[\tau \mapsto ib_1 \cdot ib_2] \qquad iP' = iP[x \mapsto iP(x) \cdot \mathtt{W}(v)\#s] \end{array}}{\langle m, iP, iB, S\rangle \xrightarrow{\tau, L}_{i\text{PTSO}_2} \langle m, iP', iB', S\rangle}
$$

PROP-FL

$$
\frac{\begin{array}{c} L = \mathtt{PropFL}(x)\#s \\ iB(\tau) = ib_1 \cdot \mathtt{FL}(x)\#s \cdot ib_2 \\ \mathtt{W}(\_, \_)\#\_, \mathtt{FL}(\_)\#\_, \mathtt{FO}(x)\#\_, \mathtt{SF}\#\_ \notin ib_1 \\ iP(x) = \epsilon \\ iB' = iB[\tau \mapsto ib_1 \cdot ib_2] \end{array}}{\langle m, iP, iB, S\rangle \xrightarrow{\tau, L}_{i\text{PTSO}_2} \langle m, iP, iB', S\rangle}
$$

PROP-FO

$$
\frac{\begin{array}{c} L = \mathtt{PropFO}(x)\#s \\ iB(\tau) = ib_1 \cdot \mathtt{FO}(x)\#s \cdot ib_2 \\ \mathtt{W}(x, \_)\#\_, \mathtt{FL}(x)\#\_, \mathtt{SF}\#\_ \notin ib_1 \\ iB' = iB[\tau \mapsto ib_1 \cdot ib_2] \qquad iP' = iP[x \mapsto iP(x) \cdot \mathtt{FO}(\tau)\#s] \end{array}}{\langle m, iP, iB, S\rangle \xrightarrow{\tau, L}_{i\text{PTSO}_2} \langle m, iP', iB', S\rangle}
$$

PROP-SF

$$
\frac{\begin{array}{c} L = \mathtt{PropSF}\#s \\ iB(\tau) = \mathtt{SF}\#s \cdot ib \\ \forall y.\ \mathtt{FO}(\tau)\#\_ \notin iP(y) \\ iB' = iB[\tau \mapsto ib] \end{array}}{\langle m, iP, iB, S\rangle \xrightarrow{\tau, L}_{i\text{PTSO}_2} \langle m, iP, iB', S\rangle}
$$

PERSIST-W

$$
\frac{\begin{array}{c} L = \mathtt{PerW}(x)\#s \\ iP(x) = \mathtt{W}(v)\#s \cdot ip \\ iP' = iP[x \mapsto ip] \qquad m' = m[x \mapsto v] \end{array}}{\langle m, iP, iB, S\rangle \xrightarrow{L}_{i\text{PTSO}_2} \langle m', iP', iB, S\rangle}
$$

PERSIST-FO

$$
\frac{\begin{array}{c} L = \mathtt{PerFO}(x)\#s \\ iP(x) = \mathtt{FO}(\tau)\#s \cdot ip \\ iP' = iP[x \mapsto ip] \end{array}}{\langle m, iP, iB, S\rangle \xrightarrow{L}_{i\text{PTSO}_2} \langle m, iP', iB, S\rangle}
$$

Fig. 7. The $i$PTSO$_2$ Instrumented Persistent Memory Subsystem (differences w.r.t. $i$PTSO$_1$ are highlighted)

$$
\begin{aligned}
i\mathsf{PTSO}_{\mathsf{syn}}.i\Sigma \triangleq{} & \{\langle \tau, \mathtt{PropW}(x)\#s\rangle \mid \tau \in \mathsf{Tid}, x \in \mathsf{Loc}, s \in \mathbb{N}\} \cup \{\langle \tau, \mathtt{PropFL}(x)\#s\rangle \mid \tau \in \mathsf{Tid}, x \in \mathsf{Loc}, s \in \mathbb{N}\} \\
& \cup \{\langle \tau, \mathtt{PropFO}(x)\#s\rangle \mid \tau \in \mathsf{Tid}, x \in \mathsf{Loc}, s \in \mathbb{N}\} \cup \{\langle \tau, \mathtt{PropSF}\#s\rangle \mid \tau \in \mathsf{Tid}, s \in \mathbb{N}\} \\
& \cup \{\mathtt{PerW}(x)\#s \mid x \in \mathsf{Loc}, s \in \mathbb{N}\} \cup \{\mathtt{PerFO}(x)\#s \mid x \in \mathsf{Loc}, s \in \mathbb{N}\}
\end{aligned}
$$

$$
m \in \mathsf{Loc} \to \mathsf{Val} \qquad iP \in \mathsf{Loc} \to (\{\mathtt{W}(x,v)\#s \mid x \in \mathsf{Loc}, v \in \mathsf{Val}, s \in \mathbb{N}\} \cup \{\mathtt{FO}(\tau)\#s \mid \tau \in \mathsf{Tid}, s \in \mathbb{N}\})^*
$$

$$
\begin{aligned}
iB \in \mathsf{Tid} \to (&\{\mathtt{W}(x,v)\#s \mid x \in \mathsf{Loc}, v \in \mathsf{Val}, s \in \mathbb{N}\} \cup \{\mathtt{FL}(x)\#s \mid x \in \mathsf{Loc}, s \in \mathbb{N}\} \\
& \cup \{\mathtt{FO}(x)\#s \mid x \in \mathsf{Loc}, s \in \mathbb{N}\} \cup \{\mathtt{SF}\#s \mid s \in \mathbb{N}\})^* \qquad S \subseteq \mathbb{N}
\end{aligned}
$$

$$
iP_{\mathsf{Init}} \triangleq \lambda x.\, \epsilon \qquad iB_{\mathsf{Init}} \triangleq \lambda \tau.\, \epsilon \qquad S_{\mathsf{Init}} = \emptyset
$$

$$
\textsc{write/flush/flush-opt/sfence}\quad
\dfrac{\begin{array}{c} S' = S \uplus \{s\} \\ \mathsf{typ}(l) \in \{\mathtt{W}, \mathtt{FL}, \mathtt{FO}, \mathtt{SF}\} \\ iB' = iB[\tau \mapsto iB(\tau)\cdot l\#s] \end{array}}{\langle m, iP, iB, S\rangle \xrightarrow{\tau, l\#s}_{i\mathsf{PTSO}_{\mathsf{syn}}} \langle m, iP, iB', S'\rangle}
$$

$$
\textsc{read}\quad
\dfrac{\begin{array}{c} S' = S \uplus \{s\} \\ l = \mathtt{R}(x,v) \\ \mathsf{get}(m, \Lambda(iP(x)), \Lambda(iB(\tau)))(x) = v \end{array}}{\langle m, iP, iB, S\rangle \xrightarrow{\tau, l\#s}_{i\mathsf{PTSO}_{\mathsf{syn}}} \langle m, iP, iB, S'\rangle}
$$

$$
\textsc{rmw}\quad
\dfrac{\begin{array}{c} S' = S \uplus \{s\} \\ l = \mathtt{RMW}(x, v_{\mathsf{R}}, v_{\mathsf{W}}) \\ \mathsf{get}(m, \Lambda(iP(x)), \epsilon)(x) = v_{\mathsf{R}} \\ iB(\tau) = \epsilon \\ \forall y.\ \mathtt{FO}(\tau)\#\_ \notin iP(y) \\ iP' = iP[x \mapsto iP(x)\cdot \mathtt{W}(v_{\mathsf{W}})\#s] \end{array}}{\langle m, iP, iB, S\rangle \xrightarrow{\tau, l\#s}_{i\mathsf{PTSO}_{\mathsf{syn}}} \langle m, iP', iB, S'\rangle}
$$

$$
\textsc{rmw-fail}\quad
\dfrac{\begin{array}{c} S' = S \uplus \{s\} \\ l = \mathtt{R\text{-}ex}(x, v) \\ \mathsf{get}(m, \Lambda(iP(x)), \epsilon)(x) = v \\ iB(\tau) = \epsilon \\ \forall y.\ \mathtt{FO}(\tau)\#\_ \notin iP(y) \end{array}}{\langle m, iP, iB, S\rangle \xrightarrow{\tau, l\#s}_{i\mathsf{PTSO}_{\mathsf{syn}}} \langle m, iP, iB, S'\rangle}
$$

$$
\textsc{mfence}\quad
\dfrac{\begin{array}{c} S' = S \uplus \{s\} \\ l = \mathtt{MF} \\ iB(\tau) = \epsilon \\ \forall y.\ \mathtt{FO}(\tau)\#\_ \notin iP(y) \end{array}}{\langle m, iP, iB, S\rangle \xrightarrow{\tau, l\#s}_{i\mathsf{PTSO}_{\mathsf{syn}}} \langle m, iP, iB, S'\rangle}
$$

$$
\textsc{prop-w}\quad
\dfrac{\begin{array}{c} L = \mathtt{PropW}(x)\#s \\ iB(\tau) = \mathtt{W}(x,v)\#s \cdot ib \\ iB' = iB[\tau \mapsto ib] \qquad iP' = iP[x \mapsto iP(x)\cdot \mathtt{W}(v)\#s] \end{array}}{\langle m, iP, iB, S\rangle \xrightarrow{\tau, L}_{i\mathsf{PTSO}_{\mathsf{syn}}} \langle m, iP', iB', S\rangle}
$$

$$
\textsc{prop-fl}\quad
\dfrac{\begin{array}{c} L = \mathtt{PropFL}(x)\#s \\ iB(\tau) = \mathtt{FL}(x)\#s \cdot ib \\ \mathtt{W}(\_,\_)\#\_, \mathtt{FL}(\_)\#\_, \mathtt{FO}(x)\#\_, \mathtt{SF}\#\_ \notin ib_1 \\ iP(x) = \epsilon \\ iB' = iB[\tau \mapsto ib] \end{array}}{\langle m, iP, iB, S\rangle \xrightarrow{\tau, L}_{i\mathsf{PTSO}_{\mathsf{syn}}} \langle m, iP, iB', S\rangle}
$$

$$
\textsc{prop-fo}\quad
\dfrac{\begin{array}{c} L = \mathtt{PropFO}(x)\#s \\ iB(\tau) = ib_1 \cdot \mathtt{FO}(x)\#s \cdot ib_2 \\ \mathtt{W}(x,\_)\#\_, \mathtt{FL}(x)\#\_, \mathtt{FO}(x)\#\_, \mathtt{SF}\#\_ \notin ib_1 \\ iB' = iB[\tau \mapsto ib_1 \cdot ib_2] \qquad iP' = iP[x \mapsto iP(x)\cdot \mathtt{FO}(\tau)\#s] \end{array}}{\langle m, iP, iB, S\rangle \xrightarrow{\tau, L}_{i\mathsf{PTSO}_{\mathsf{syn}}} \langle m, iP', iB', S\rangle}
$$

$$
\textsc{prop-sf}\quad
\dfrac{\begin{array}{c} L = \mathtt{PropSF}\#s \\ iB(\tau) = \mathtt{SF}\#s \cdot ib \\ \forall y.\ \mathtt{FO}(\tau)\#\_ \notin iP(y) \\ iB' = iB[\tau \mapsto ib] \end{array}}{\langle m, iP, iB, S\rangle \xrightarrow{\tau, L}_{i\mathsf{PTSO}_{\mathsf{syn}}} \langle m, iP, iB', S\rangle}
$$

$$
\textsc{persist-w}\quad
\dfrac{\begin{array}{c} L = \mathtt{PerW}(x)\#s \\ iP(x) = \mathtt{W}(v)\#s \cdot ip \\ iP' = iP[x \mapsto ip] \qquad m' = m[x \mapsto v] \end{array}}{\langle m, iP, iB, S\rangle \xrightarrow{L}_{i\mathsf{PTSO}_{\mathsf{syn}}} \langle m', iP', iB, S\rangle}
$$

$$
\textsc{persist-fo}\quad
\dfrac{\begin{array}{c} L = \mathtt{PerFO}(x)\#s \\ iP(x) = \mathtt{FO}(\tau)\#s \cdot ip \\ iP' = iP[x \mapsto ip] \end{array}}{\langle m, iP, iB, S\rangle \xrightarrow{L}_{i\mathsf{PTSO}_{\mathsf{syn}}} \langle m, iP', iB, S\rangle}
$$

Fig. 8. The $i$PTSO$_{\mathsf{syn}}$ Instrumented Persistent Memory Subsystem (differences w.r.t. $i$PTSO$_2$ are highlighted)

## B.5 Proof of Theorem 4.6

With the four systems above, we prove Thm. 4.6.

Utilizing Lemma 2.10, we need to show:

(A) Every $m_0$-initialized $\mathsf{PTSO_{syn}}$-observable-trace is also an $m_0$-initialized Px86-observable-trace.
(B) For every $m_0$-to-$m$ $\mathsf{PTSO_{syn}}$-observable-trace $t$, some $t' \lesssim t$ is an $m_0$-to-$m$ Px86-observable-trace.
(C) Every $m_0$-initialized Px86-observable-trace is also an $m_0$-initialized $\mathsf{PTSO_{syn}}$-observable-trace.
(D) For every $m_0$-to-$m$ Px86-observable-trace $t$, some $t' \lesssim t$ is an $m_0$-to-$m$ $\mathsf{PTSO_{syn}}$-observable-trace.

In the proof outlines below, we highlight the steps whose proofs we found more interesting. The proofs of the non-highlighted steps are easier and mostly straightforward.

### B.5.1 General Definitions for all Parts.

*Definition B.10.* Let $A$ be an LTS. We say that a pair $\langle \sigma, \sigma' \rangle \in A.\Sigma \times A.\Sigma$ of transition labels *A-commutes* if

$$\xrightarrow{\sigma}_A \; ; \; \xrightarrow{\sigma'}_A \; \subseteq \; \xrightarrow{\sigma'}_A \; ; \; \xrightarrow{\sigma}_A .$$

*Definition B.11.* A trace $it$ of one the systems $i$Px86, $i\mathsf{PTSO_2}$, or $i\mathsf{PTSO_{syn}}$ is called PropFO-*complete* if for every $i \in dom(it)$ with $it(i) = \langle \tau, \mathtt{PropFO}(x)\#s \rangle$, we have $\#(it(j)) = s$ for some $j > i$. In addition, if $it$ is a $i$Px86-trace, we also say that $it$ is

1. PropFL-*complete* if for every $i \in dom(it)$ with $it(i) = \langle \tau, \mathtt{PropFL}(x)\#s \rangle$, we have $\#(it(j)) = s$ for some $j > i$.
2. $\{\mathtt{PropFL}, \mathtt{PropFO}\}$-*complete* if $it$ is both PropFL-complete and PropFO-complete.

*Definition B.12.* Given a trace $it$ of one the systems $i\mathsf{PTSO_2}$ or $i\mathsf{PTSO_{syn}}$, the *delay function* $d_{it} : dom(it) \to \mathbb{N}$ assigns to every $i \in dom(it)$ with $\mathtt{typ}(it(i)) \in \{\mathtt{RMW}, \mathtt{PropW}, \mathtt{PropFO}\}$ the difference $j - i - 1$ where $j > i$ is the (unique) index satisfying $\#(it(j)) = \#(it(i))$. If $\mathtt{typ}(it(i)) \notin \{\mathtt{RMW}, \mathtt{PropW}, \mathtt{PropFO}\}$ or such index $j$ does not exist, the delay $d_{it}(i)$ is defined to be 0. Similarly, if $it$ is a trace of $i$Px86, the *delay function* $d_{it} : dom(it) \to \mathbb{N}$ assigns to every $i \in dom(it)$ with $\mathtt{typ}(it(i)) \in \{\mathtt{RMW}, \mathtt{PropW}, \mathtt{PropFO}, \mathtt{PropFL}\}$ the difference $j - i - 1$ where $j > i$ is the (unique) index satisfying $\#(it(j)) = \#(it(i))$. If $\mathtt{typ}(it(i)) \notin \{\mathtt{RMW}, \mathtt{PropW}, \mathtt{PropFO}, \mathtt{PropFL}\}$ or such index $j$ does not exist, the delay $d_{it}(i)$ is defined to be 0.

*Definition B.13.* A trace $it$ of one the systems $i$Px86, $i\mathsf{PTSO_2}$, or $i\mathsf{PTSO_{syn}}$ is *synchronous* if $d_{it}(i) = 0$ for every $1 \leq i \leq |it|$.

### B.5.2 Proof of (A).

The proof of (A) is structured as follows:

(A.0) Let $t$ be an $m_0$-initialized $\mathsf{PTSO_{syn}}$-observable-trace.
(A.1) By Lemmas B.3 and B.9, there exists some $m_0$-initialized $i\mathsf{PTSO_{syn}}$-trace $it$ such that $\Lambda(it) = t$.
(A.2) By Lemma B.16, there exists some $m_0$-initialized $i$Px86-trace $it'$ such that $\Lambda(it') = \Lambda(it)$.
(A.3) By Lemmas B.3 and B.6, $\Lambda(it')$ is an $m_0$-initialized Px86-observable-trace.
(A.4) Then, the claim follows observing that $\Lambda(it') = \Lambda(it) = t$.

Lemma B.14. *For every $m_0$-initialized $i\mathsf{PTSO_{syn}}$-trace $it$, there exists some PropFO-complete $m_0$-initialized $i\mathsf{PTSO_{syn}}$-trace $it'$ such that $\Lambda(it) = \Lambda(it')$.*

Proof. $it$ can be extended to some $it'$ so that every $\langle \_, \mathtt{RMW}(x, \_, v)\#s \rangle$, $\langle \_, \mathtt{PropW}(x)\#s \rangle$, and $\langle \_, \mathtt{PropFO}(x)\#s \rangle$ has a matching $\mathtt{PerW}(x)\#s$ or $\mathtt{PerFO}(x)\#s$. Indeed, since it is always possible to persist entries of persistence buffer in order, we can simply append corresponding labels in the order in which unmatched propagation events occur in $it$. □

Lemma B.15. *For every PropFO-complete $m_0$-initialized $i\mathsf{PTSO_{syn}}$-trace $it$, there exists some synchronous PropFO-complete $m_0$-initialized $i\mathsf{PTSO_{syn}}$-trace $it'$ such that $\Lambda(it) = \Lambda(it')$.*

Proof sketch. We can transform $it$ into a synchronous $\mathtt{PropFO}$-complete $m_0$-initialized $i$PTSO$_{\mathsf{syn}}$-trace $it'$ simply by moving $\mathtt{PerW}(x)\#s$ and $\mathtt{PerFO}(x)\#s$ immediately after matching $\langle\_, \mathtt{PropW}(x)\#s\rangle$, $\langle\_, \mathtt{RMW}(x, \_, v)\#s\rangle$, or $\langle\_, \mathtt{PropFO}(x)\#s\rangle$ labels in $it$. In a $\mathtt{PropFO}$-complete trace, the writes $x$ that do not persist always occur after $\mathtt{PerFO}(x)\#\_$ steps. With that observed, one can argue that considering propagation labels in order and moving their matching persist labels is possible, as relevant persistence buffers constraints are satisfied by construction. □

Lemma B.16 (Step A.2). *For every $m_0$-initialized $i$PTSO$_{\mathsf{syn}}$-trace $it$, there exists some $m_0$-initialized $i$Px86-trace $it'$ such that $\Lambda(it') = \Lambda(it)$.*

Proof sketch. By Lemma B.14 applied to $it$, there exists some $\mathtt{PropFO}$-complete $m_0$-initialized $i$PTSO$_{\mathsf{syn}}$-trace $it_1$ such that $\Lambda(it) = \Lambda(it_1)$. Moreover, by Lemma B.15 applied to $it_1$, there exists some synchronous $\mathtt{PropFO}$-complete $m_0$-initialized $i$PTSO$_{\mathsf{syn}}$-trace $it'_1$ such that $\Lambda(it_1) = \Lambda(it'_1)$. We further transform $it'_1$ into $it'$ by putting a persist step $\mathtt{PerPER}(x)\#s$ after each $\mathtt{PropFL}(x)\#s$, and by replacing $\mathtt{PerFO}(x)\#s$ after each $\mathtt{PropFO}(x)\#s$ with $\mathtt{PerPER}(x)\#s$. Note that the resulting trace is $\{\mathtt{PropFL}, \mathtt{PropFO}\}$-complete and synchronous.

We argue that $it'$ that is a $i$Px86-trace. Indeed, for all but persistence steps, whenever $i$PTSO$_{\mathsf{syn}}$ performs a step, the same step is possible in $i$Px86. The persistence steps in $it'$ are enabled by construction, since their constraints on the content of the persistence buffer are trivially satisfied in a synchronous trace. Overall, we have constructed $it'$ that is $m_0$-initialized $i$Px86-trace such that $\Lambda(it') = \Lambda(it)$. □

### B.5.3 Proof of (B).

The proof of (B) is structured as follows:

(B.0) Let $t$ be an $m_0$-to-$m$ PTSO$_{\mathsf{syn}}$-observable-trace.
(B.1) By Lemmas B.3 and B.9, there exists some $m_0$-to-$m$ $i$PTSO$_{\mathsf{syn}}$-trace $it$ such that $\Lambda(it) = t$.
(B.2) By Lemma B.17, $it$ is also an $m_0$-to-$m$ $i$PTSO$_2$-trace.
(B.3) By Lemma B.22, there exists some $m_0$-to-$m$ $i$PTSO$_1$-trace $it_1$ such that $\Lambda(it_1) \lesssim \Lambda(it)$.
(B.4) By Lemma B.23, there exists some $m_0$-to-$m$ $i$Px86-trace $it'$ such that $\Lambda(it') = \Lambda(it_1)$.
(B.5) By Lemmas B.3 and B.6, $\Lambda(it')$ is an $m_0$-to-$m$ Px86-observable-trace.
(B.6) Then, the claim follows observing that $\Lambda(it') = \Lambda(it_1) \lesssim \Lambda(it) = t$.

Lemma B.17. *Every $m_0$-to-$m$ $i$PTSO$_{\mathsf{syn}}$-trace $it$ is also an $m_0$-to-$m$ $i$PTSO$_2$-trace.*

Proof. Every transition of $i$PTSO$_{\mathsf{syn}}$ is also a transition of $i$PTSO$_2$. □

Lemma B.18. *For every $m_0$-to-$m$ $i$PTSO$_2$-trace $it$, there exists some $\mathtt{PropFO}$-complete $m_0$-to-$m$ $i$PTSO$_2$-trace $it'$ such that $\Lambda(it') \lesssim \Lambda(it)$.*

Proof. We take $it'$ to be the trace obtained from $it$ by discarding all transition labels at an index $i$ with $\mathtt{typ}(it(i)) = \mathtt{PropFO}$ but $\#(it(j)) \neq \#(it(i))$ for every $j > i$. It is straightforward to verify that $it'$ is a $\mathtt{PropFO}$-complete $m_0$-to-$m$ $i$PTSO$_2$-trace, as well as that $\Lambda(it') \lesssim \Lambda(it)$. □

Proposition B.19. *$\langle \alpha, \beta \rangle$ $i$PTSO$_2$-commutes if $\mathtt{typ}(\beta) \in \{\mathtt{PerW}, \mathtt{PerFO}\}$ and one of the following conditions holds:*

- $\mathtt{typ}(\alpha) \notin \{\mathtt{PerW}, \mathtt{PerFO}\}$ *and* $\#(\alpha) \neq \#(\beta)$.
- $\mathtt{typ}(\alpha) \in \{\mathtt{PerW}, \mathtt{PerFO}\}$ *and* $\mathtt{loc}(\alpha) \neq \mathtt{loc}(\beta)$.

Lemma B.20. *For every $\mathtt{PropFO}$-complete $m_0$-to-$m$ $i$PTSO$_2$-trace $it$, there exists some synchronous $\mathtt{PropFO}$-complete $m_0$-to-$m$ $i$PTSO$_2$-trace $it'$ such that $\Lambda(it') = \Lambda(it)$.*

Proof. By induction on the sum of delays in $it$ (i.e., $\sum_i d_{it}(i)$). If this sum is 0, then we can take $it' = it$. Otherwise, consider the minimal $1 \leq i \leq |it|$ with $d_{it}(i) > 0$. Then, we have

$\mathtt{typ}(it(i)) \in \{\mathtt{RMW}, \mathtt{PropW}, \mathtt{PropFO}\}$ and $\#(it(j)) = \#(it(i))$ for $j = i + d_{it}(i) + 1$. Following $i$PTSO$_2$'s transitions, it must be the case that $\mathtt{loc}(it(j)) = \mathtt{loc}(it(i))$, $\mathtt{typ}(it(j)) = \mathtt{PerW}$ if $\mathtt{typ}(it(i)) \in \{\mathtt{RMW}, \mathtt{PropW}\}$, and $\mathtt{typ}(it(j)) = \mathtt{PerFO}$ if $\mathtt{typ}(it(i)) = \mathtt{PropFO}$. Now, it is straightforward to verify that $\langle it(j-1), it(j)\rangle$ must satisfy one of the conditions in Prop. B.19, and so this pair $i$PTSO$_2$-commutes. The resulting PropFO-complete $m_0$-to-$m$ $i$PTSO$_2$-trace has smaller sum of delays, and the claim follows by applying the induction hypothesis. □

Lemma B.21. *For every synchronous PropFO-complete $m_0$-to-$m$ $i$PTSO$_2$-trace $it$, there exists some $m_0$-to-$m$ $i$PTSO$_1$-trace $it'$ such that $\Lambda(it') = \Lambda(it)$.*

Proof. We obtain $it'$ by merging consecutive prop-fo and persist-fo steps in $it$ into one prop-fo step of $i$PTSO$_1$, thus maintaining the persistence buffers without FO-entries. □

Lemma B.22 (Step B.3). *For every $m_0$-to-$m$ $i$PTSO$_2$-trace $it$, there exists some $m_0$-to-$m$ $i$PTSO$_1$-trace $it'$ such that $\Lambda(it') \lesssim \Lambda(it)$.*

Proof. By Lemma B.18, there exists some PropFO-complete $m_0$-to-$m$ $i$PTSO$_2$-trace $it_c$ such that $\Lambda(it_c) \lesssim \Lambda(it)$. Then, by Lemma B.20, there exists a synchronous PropFO-complete $m_0$-to-$m$ $i$PTSO$_2$-trace $it_s$ such that $\Lambda(it_s) = \Lambda(it_c)$. Then, by Lemma B.21, there exists an $m_0$-to-$m$ $i$PTSO$_1$-trace $it'$ such that $\Lambda(it') = \Lambda(it_s)$. Now, since $\Lambda(it_c) \lesssim \Lambda(it)$, $\Lambda(it_s) = \Lambda(it_c)$, and $\Lambda(it') = \Lambda(it_s)$, we have that $\Lambda(it') \lesssim \Lambda(it)$, and the claim follows. □

Lemma B.23 (Step B.4). *For every $m_0$-to-$m$ $i$PTSO$_1$-trace $it$, there exists some $m_0$-to-$m$ $i$Px86-trace $it'$ such that $\Lambda(it') = \Lambda(it)$.*

Proof sketch. We transform $it$ into $it'$ by putting a persist step $\mathtt{PerPER}(x)\#s$ after each occurrence of $\mathtt{PropFL}(x)\#s$ or $\mathtt{PropFO}(x)\#s$. All of the steps in $it'$ are trivially enabled in $i$Px86 by construction, so $it'$ is an $m_0$-to-$m$ $i$Px86-trace. □

*B.5.4 Helper Lemmas for (C) and (D).* To prove (C) and (D), we introduce several trace transformation properties for persisting synchronously.

Proposition B.24. *$\langle \alpha, \beta\rangle$ $i$Px86-commutes if $\mathtt{typ}(\beta) \in \{\mathtt{PerW}, \mathtt{PerPER}\}$ and one of the following conditions holds:*

- $\mathtt{typ}(\alpha) \notin \{\mathtt{PerW}, \mathtt{PerPER}\}$ *and* $\#(\alpha) \neq \#(\beta)$.
- $\mathtt{typ}(\alpha) = \mathtt{PerW}$ *and* $\mathtt{loc}(\alpha) \neq \mathtt{loc}(\beta)$.

Lemma B.25. *For every $\{\mathtt{PropFL}, \mathtt{PropFO}\}$-complete $m_0$-to-$m$ $i$Px86-trace $it$, there exists some synchronous $\{\mathtt{PropFL}, \mathtt{PropFO}\}$-complete $m_0$-to-$m$ $i$Px86-trace $it'$ such that $\Lambda(it') = \Lambda(it)$.*

Proof. By induction on the sum of delays in $it$ (i.e., $\sum_i d_{it}(i)$). If this sum is 0, then we can take $it' = it$. Otherwise, consider the minimal $1 \leq i \leq |it|$ with $d_{it}(i) > 0$. Then, we have $\mathtt{typ}(it(i)) \in \{\mathtt{RMW}, \mathtt{PropW}, \mathtt{PropFL}, \mathtt{PropFO}\}$ and $\#(it(j)) = \#(it(i))$ for $j = i + d_{it}(i) + 1$. Following $i$Px86's transitions, it must be the case that $\mathtt{loc}(it(j)) = \mathtt{loc}(it(i))$, $\mathtt{typ}(it(j)) = \mathtt{PerW}$ if $\mathtt{typ}(it(i)) \in \{\mathtt{RMW}, \mathtt{PropW}\}$, and $\mathtt{typ}(it(j)) = \mathtt{PerPER}$ if $\mathtt{typ}(it(i)) \in \{\mathtt{PropFL}, \mathtt{PropFO}\}$. Consider the possible cases:

1. $\mathtt{typ}(it(j-1)) \notin \{\mathtt{PerW}, \mathtt{PerPER}\}$: Then, by Prop. B.24, $\langle it(j-1), it(j)\rangle$ $i$Px86-commutes. The resulting PropFO-complete $m_0$-to-$m$ $i$Px86-trace has smaller sum of delays, and the claim follows by applying the induction hypothesis.
2. $\mathtt{typ}(it(j-1)) = \mathtt{PerW}$: The minimality of $i$ ensures that the index $i'$ with $\#(it(i')) = \#(it(j-1))$ satisfies $i' \geq i$. Following $i$Px86's transitions, we must have $\mathtt{loc}(it(j-1)) \neq \mathtt{loc}(it(j))$ (writes to the same location persist in their propagation order). Then, again, the claim follows using Prop. B.24 and the induction hypothesis.

(3) $\mathtt{typ}(it(j-1)) = \mathtt{PerPER}$: The minimality of $i$ ensures that the index $i'$ with $\#(it(i')) = \#(it(j-1))$ satisfies $i' \geq i$. Following $i$Px86's transitions, we must have $\mathtt{typ}(it(j)) = \mathtt{PerW}$ ($\mathtt{PER}$-entries to the same location are removed from the persistence buffer in their propagation order), as well as $\mathtt{loc}(it(j-1)) \neq \mathtt{loc}(it(j))$ (a $\mathtt{PER}$-entry cannot be removed from the persistence buffer if there is a preceding write entry to the same location). In this case we can swap $it(j-1)$ and $t(j)$, and, as before obtain a $\mathtt{PropFO}$-complete $m_0$-to-$m$ $i$Px86-trace, so the claim follows by the induction hypothesis. □

Lemma B.26 (Steps C.3 and D.3). *For every $m_0$-to-$m$ $i$PTSO$_2$-trace $it$, there exists some $m_0$-to-$m$ $i$PTSO$_{\mathsf{syn}}$-trace $it'$ such that $\Lambda(it') = \Lambda(it)$.*

Proof (outline). We use a standard forward simulation argument, where $i$PTSO$_{\mathsf{syn}}$ eagerly takes PROP-FO and PERSIST-FO steps whenever possible. Then, $i$PTSO$_{\mathsf{syn}}$ is always at a state in which the flush-optimals are further propagated w.r.t. the corresponding state of $i$PTSO$_2$ (e.g., a flush-optimal in $i$PTSO$_2$'s store buffer may already be in $i$PTSO$_{\mathsf{syn}}$'s persistence buffer). In this case, the flush-optimals impose only (possibly) weaker constraints on the transitions. For this argument to work we rely on the fact that a flush-optimal of a certain thread being further propagated does not impose constraints on actions of other threads.

More formally, we define a simulation relation $R$ between $i$PTSO$_2$-states and $i$PTSO$_{\mathsf{syn}}$-states. To define $R$ we use the notation $s|_T$ to restrict a sequence $s$ (which will be an instrumented per-location persistence buffer or an instrumented store buffer) to entries of type $\mathtt{X} \in T$ (yielding a possibly shorter sequence). The simulation relation $R \subseteq i\mathrm{PTSO}_2.\mathtt{Q} \times i\mathrm{PTSO}_{\mathsf{syn}}.\mathtt{Q}$ is defined as follows: $\langle\langle m_2, iP_2, iB_2, S_2\rangle, \langle m, iP, iB, S\rangle\rangle \in R$ if the following hold:

- $m_2 = m$ and $S_2 = S$.
- For every $x \in \mathsf{Loc}$, $iP_2(x)|_{\{\mathtt{W}\}} = iP(x)|_{\{\mathtt{W}\}}$.
- For every $\tau \in \mathsf{Tid}$, $iB_2(\tau)|_{\{\mathtt{W},\mathtt{FL},\mathtt{SF}\}} = iB(\tau)|_{\{\mathtt{W},\mathtt{FL},\mathtt{SF}\}}$.
- If $iB(\tau)(i) \in \{\mathtt{W}(x,\_)\#\_, \mathtt{FL}(x)\#\_, \mathtt{SF}\#\_\}$ and $iB(\tau)(j) = \mathtt{FO}(x)\#\_$ for some $i < j$, then $iB(\tau)(i_2) = iB(\tau)(i)$ and $iB(\tau)(j_2) = iB(\tau)(j)$ for some $i_2 < j_2$.
- If $iP(x)(i) = \mathtt{W}(\_)\#\_$ and $iP(x)(j) = \mathtt{FO}(\tau)\#s$ for some $i < j$, then one of the following holds:
  - $iP_2(x)(i_2) = iP(x)(i)$ and $iP_2(x)(j_2) = \mathtt{FO}(\tau)\#s$ for some $i_2 < j_2$; or
  - $iP_2(x)(i_2) = iP(x)(i)$ and $iB_2(\tau)(j_2) = \mathtt{FO}(x)\#s$ for some $i_2$ and $j_2$.
- If $iB_2(\tau)(i_2) = \mathtt{SF}\#\_$ and $iB_2(\tau)(j_2) = \mathtt{FO}(\_)\#\_$ for some $i_2 < j_2$, then $iB(\tau)(i) = iB_2(\tau)(i_2)$ and $iB(\tau)(j) = iB_2(\tau)(j_2)$ for some $i < j$.
- If $iB(\tau)(j) = \mathtt{FO}(x)\#\_$, then $iB(\tau)(i) \in \{\mathtt{W}(x,\_)\#\_, \mathtt{FL}(x)\#\_, \mathtt{SF}\#\_\}$ for some $i < j$.
- If $iP(x)(j) = \mathtt{FO}(\_)\#\_$, then $iP(x)(i) = \mathtt{W}(\_)\#\_$ for some $i < j$.

Initially, we clearly have $\langle\langle m_0, P_\epsilon, B_\epsilon, \emptyset\rangle, \langle m_0, P_\epsilon, B_\epsilon, \emptyset\rangle\rangle \in R$. Now, suppose that $\langle m, iP_2, iB_2, S\rangle \xrightarrow{\alpha}_{i\mathrm{PTSO}_2} \langle m', iP_2', iB_2', S'\rangle$, and let $\langle m_1, iP, iB, S_1\rangle \in i\mathrm{PTSO}_{\mathsf{syn}}.\mathtt{Q}$ such that $\langle\langle m, iP_2, iB_2, S\rangle, \langle m_1, iP, iB, S_1\rangle\rangle \in R$. Then, we have $m = m_1$ and $S = S_1$. We show that $\langle m, iP, iB, S\rangle \xrightarrow{t}_{i\mathrm{PTSO}_{\mathsf{syn}}} \langle m', iP', iB', S'\rangle$ for some $t$, $iP'$, and $iB'$ such that $\Lambda(t) = \Lambda(\alpha)$ and $\langle\langle m', iP_2', iB_2', S'\rangle, \langle m', iP', iB', S'\rangle\rangle \in R$.

Roughly speaking, to obtain this we will make $i$PTSO$_{\mathsf{syn}}$ take PERSIST-FO steps as eagerly as possible after every other step. (Thus, when $i$PTSO$_2$ takes a PROP-FO or PERSIST-FO step, $i$PTSO$_{\mathsf{syn}}$ remains in the same state.) The rest of the proof continues by separately considering each possible step of $i$PTSO$_2$, and establishing the simulation invariants at each step. For example, suppose that $\langle m, iP_2, iB_2, S\rangle \xrightarrow{\tau,\mathtt{PropW}(x)\#s}_{i\mathrm{PTSO}_2} \langle m', iP_2', iB_2', S'\rangle$. Then, the simulation invariants ensure that $\langle m, iP, iB, S\rangle \xrightarrow{\tau,\mathtt{PropW}(x)\#s}_{i\mathrm{PTSO}_{\mathsf{syn}}} \langle m', iP_{\mathrm{mid}}, iB_{\mathrm{mid}}, S'\rangle$ for some $iP_{\mathrm{mid}}$ and $iB_{\mathrm{mid}}$. Then, to establish the simulation invariant, we repeatedly execute PROP-FO and PERSIST-FO steps as long as it is possible and obtain the state $\langle m', iP', iB', S'\rangle$. □

*B.5.5 Proof of (C).* The proof of (C) is structured as follows:

(C.0) Let $t$ be an $m_0$-initialized Px86-observable-trace.
(C.1) By Lemmas B.3 and B.6, there exists some $m_0$-initialized $i$Px86-trace $it$ such that $\Lambda(it) = t$.
(C.2) By Lemma B.28, there exists some $m_0$-initialized $i$PTSO$_2$-trace $it_2$ such that $\Lambda(it_2) = \Lambda(it)$.
(C.3) By Lemma B.26, there exists some $m_0$-initialized $i$PTSO$_{\text{syn}}$-trace $it'_2$ such that $\Lambda(it'_2) = \Lambda(it_2)$.
(C.4) By Lemmas B.3 and B.9, $\Lambda(it'_2)$ is an $m_0$-initialized PTSO$_{\text{syn}}$-observable-trace.
(C.5) Then, the claim follows observing that $\Lambda(it'_2) = \Lambda(it_2) = \Lambda(it) = t$.

The next lemma states that every trace can be continued to empty the content of its persistence buffer.

Lemma B.27. *For every $m_0$-initialized $i$Px86-trace $it$, there exists some $\{\texttt{PropFL}, \texttt{PropFO}\}$-complete $m_0$-initialized $i$Px86-trace $it'$ such that $\Lambda(it) = \Lambda(it')$.*

Proof sketch. $it$ can be extended to some $it'$ so that every $\langle\_, \texttt{RMW}(x, \_, v)\#s\rangle$, $\langle\_, \texttt{PropW}(x)\#s\rangle$, $\langle\_, \texttt{PropFO}(x)\#s\rangle$ or $\langle\_, \texttt{PropFL}(x)\#s\rangle$ has a matching $\texttt{PerW}(x)\#s$ or $\texttt{PerPER}(x)\#s$. Indeed, since it is always possible to persist entries of persistence buffer in order, we can simply append corresponding labels in the order, in which unmatched propagation events occur in $it$. □

Lemma B.28 (Step C.2). *For every $m_0$-initialized $i$Px86-trace $it$, there exists some $m_0$-initialized $i$PTSO$_2$-trace $it_2$ such that $\Lambda(it_2) = \Lambda(it)$.*

Proof sketch. By Lemma B.27 applied to $it$, there is some $\{\texttt{PropFL}, \texttt{PropFO}\}$-complete $i$Px86-trace $it_1$ such that $\Lambda(it) = \Lambda(it_1)$. Moreover, by applying Lemma B.25 to $it_1$, there is some synchronous $\{\texttt{PropFL}, \texttt{PropFO}\}$-complete $m_0$-initialized $i$Px86-trace $it'_1$ such that $\Lambda(it_1) = \Lambda(it'_1)$. We transform $it'_1$ further into $it_2$ by removing every $\texttt{PerPER}(x)\#s$ following $\langle\_, \texttt{PropFL}(x)\#s\rangle$, and by replacing every $\texttt{PerPER}(x)\#s$ following $\langle\_, \texttt{PropFO}(x)\#s\rangle$ with $\texttt{PerFO}(x)\#s$.

We argue that $it_2$ that is an $i$PTSO$_2$-trace. Indeed, by construction of $it'$, each persistence buffer $iP(x)$ only contains $\texttt{FO}(\tau)\#s$-entries right before the step propagating them from the buffer takes place. Moreover, each persistence buffer $iP(x)$ does not contain $\texttt{W}(v)\#s$-entries upon executing $\langle\_, \texttt{PropFL}(x)\#s\rangle$ steps, since the conditions for persisting flush instructions in $it'_1$ ensure that such writes previously persisted. Hence, the constraints on the content of the persistence buffers are satisfied in $i$PTSO$_2$ by construction. □

*B.5.6 Proof of (D).* The proof of (D) is structured as follows:

(D.0) Let $t$ be an $m_0$-to-$m$ Px86-observable-trace.
(D.1) By Lemmas B.3 and B.6, there exists some $m_0$-to-$m$ $i$Px86-trace $it$ such that $\Lambda(it) = t$.
(D.2) By Lemma B.31, there exists some $m_0$-to-$m$ $i$PTSO$_1$-trace $it_1$ such that $\Lambda(it_1) \lesssim \Lambda(it)$.
(D.3) By Lemma B.32, there exists some $m_0$-to-$m$ $i$PTSO$_2$-trace $it_2$ such that $\Lambda(it_2) = \Lambda(it_1)$.
(D.4) By Lemma B.26, there exists some $m_0$-to-$m$ $i$PTSO$_{\text{syn}}$-trace $it'_2$ such that $\Lambda(it'_2) = \Lambda(it_2)$.
(D.5) By Lemmas B.3 and B.9, $\Lambda(it'_2)$ is an $m_0$-to-$m$ PTSO$_{\text{syn}}$-observable-trace.
(D.6) Then, the claim follows observing that $\Lambda(it'_2) = \Lambda(it_2) = \Lambda(it_1) \lesssim \Lambda(it) = t$.

Lemma B.29. *For every $m_0$-to-$m$ $i$Px86-trace $it$, there exists some $\{\texttt{PropFL}, \texttt{PropFO}\}$-complete $m_0$-to-$m$ $i$Px86-trace $it'$ such that $\Lambda(it') \lesssim \Lambda(it)$.*

Proof. Let $i_0$ be the minimal index for which $\texttt{typ}(it(i_0)) \in \{\texttt{PropFL}, \texttt{PropFO}\}$ but $\#(it(j)) \neq \#(it(i_0))$ for every $j > i_0$. Let $i_1, \dots, i_m$ be an enumeration of all indices $i > i_0$ with $\texttt{typ}(it(i)) \in \{\texttt{PerW}, \texttt{PerPER}\}$. We define $it' = it(1), \dots, it(i_0 - 1), it(i_1), \dots, it(i_m)$. We trivially have that $\Lambda(it') \lesssim \Lambda(it)$. To see that $it'$ is a ($\{\texttt{PropFL}, \texttt{PropFO}\}$-complete) $i$Px86-trace, it suffices to note that the transitions of $i$Px86 ensure that for every $1 \leq j \leq m$ with $\texttt{typ}(it(i_j)) = \texttt{PerW}$, we have $\texttt{typ}(it(k)) \in \{\texttt{RMW}, \texttt{PropW}\}$ and $\#(it(k)) = \#(it(i_j))$ for some $k < i_0$; and for every $1 \leq j \leq m$ with $\texttt{typ}(it(i_j)) =$

$\mathtt{PerPER}$, we have $\mathtt{typ}(it(k)) \in \{\mathtt{PropFL}, \mathtt{PropFO}\}$ and $\#(it(k)) = \#(it(i_j))$ for some $k < i_0$. Finally, since $it'$ includes all $\mathtt{PerW}$ transitions of $it$, it is an $m_0$-to-$m$ $i$Px86-trace. □

Lemma B.30. *For every synchronous $\{\mathtt{PropFL}, \mathtt{PropFO}\}$-complete $m_0$-to-$m$ $i$Px86-trace $it$, there exists some $m_0$-to-$m$ $i$PTSO$_1$-trace $it'$ such that $\Lambda(it') = \Lambda(it)$.*

Proof. We obtain $it'$ by merging consecutive prop-fl/prop-fo and persist-per steps in $it$ into one prop-fl/prop-fo step of $i$PTSO$_1$, thus maintaining the persistence buffers without PER-entries. □

Lemma B.31 (Step D.2). *For every $m_0$-to-$m$ $i$Px86-trace $it$, there exists some $m_0$-to-$m$ $i$PTSO$_1$-trace $it'$ such that $\Lambda(it') \lesssim \Lambda(it)$.*

Proof. By Lemma B.29, there exists some $\{\mathtt{PropFL}, \mathtt{PropFO}\}$-complete $m_0$-to-$m$ $i$Px86-trace $it_c$ such that $\Lambda(it_c) \lesssim \Lambda(it)$. Then, by Lemma B.25, there exists a synchronous $\{\mathtt{PropFL}, \mathtt{PropFO}\}$-complete $m_0$-to-$m$ $i$Px86-trace $it_s$ such that $\Lambda(it_s) = \Lambda(it_c)$. Then, by Lemma B.30, there exists an $m_0$-to-$m$ $i$PTSO$_1$-trace $it'$ such that $\Lambda(it') = \Lambda(it_s)$. Now, since $\Lambda(it_c) \lesssim \Lambda(it)$, $\Lambda(it_s) = \Lambda(it_c)$, and $\Lambda(it') = \Lambda(it_s)$, we have that $\Lambda(it') \lesssim \Lambda(it)$, and the claim follows. □

Lemma B.32 (Step D.3). *For every $m_0$-to-$m$ $i$PTSO$_1$-trace $it$, there exists some $m_0$-to-$m$ $i$PTSO$_2$-trace $it'$ such that $\Lambda(it') = \Lambda(it)$.*

Proof sketch. $i$PTSO$_2$ can simulate $i$PTSO$_1$ by taking a persist-fo step immediately after every prop-fo step, keeping the persistence buffers without any $\mathtt{FO(\_)\#\_}$ entries. □

## B.6 Proof of Lemma 4.5

Lemma 4.5. *Suppose that $\langle m_0, P_\epsilon, B_\epsilon\rangle \overset{t}{\Rightarrow}_{\mathrm{PTSO_{syn}}} \langle m, P, B\rangle$. Then:*

- *$\langle m_0, P_\epsilon, B_\epsilon\rangle \overset{t}{\Rightarrow}_{\mathrm{PTSO_{syn}}} \langle m', P', B_\epsilon\rangle$ for some $m'$ and $P'$.*
- *$\langle m_0, P_\epsilon, B_\epsilon\rangle \overset{t'}{\Rightarrow}_{\mathrm{PTSO_{syn}}} \langle m, P, B_\epsilon\rangle$ for some $t' \lesssim t$.*

Proof. The first item is trivial (we can simply propagate and persist whatever needed in the end of the trace). We prove the second using the instrumented system $i$PTSO$_{\mathrm{syn}}$. By Lemmas B.3 and B.9, there exist $it$, $iP$, $iB$, and $S \subseteq \mathbb{N}$, such that $\langle m_0, P_\epsilon, B_\epsilon, \emptyset\rangle \xrightarrow{it}_{i\mathrm{PTSO_{syn}}} \langle m, iP, iB, S\rangle$, $\Lambda(it) = t$, $\Lambda(iP) = P$, and $\Lambda(iB) = B$. For every $\tau \in \mathrm{Tid}$, let $i_\tau$ be the minimal index such that $\mathtt{tid}(it(i_\tau)) = \tau$, $\mathtt{typ}(it(i_\tau)) \in \{\mathtt{W}, \mathtt{FL}, \mathtt{FO}, \mathtt{SF}\}$, and $\#(it(j)) \neq \#(it(i_\tau))$ for every $j > i$ (that is, the operation in index $i_\tau$ never propagated from the store buffer). If such index does not exist, we let $i_\tau = \bot$. For every $\tau \in \mathrm{Tid}$, let $I_\tau$ be the set of all indices $i \geq i_\tau$ such that $\mathtt{tid}(it(i)) = \tau$ and $\mathtt{typ}(it(i)) \in \{\mathtt{W}, \mathtt{R}, \mathtt{RMW}, \mathtt{R\text{-}ex}, \mathtt{MF}, \mathtt{FL}, \mathtt{FO}, \mathtt{SF}\}$ (that is, the operation in index $i$ was issued after an operation that never propagated from the store buffer). If $i_\tau = \bot$, we let $I_\tau = \emptyset$. Now, let $it'$ be the sequence obtained from $t$ by omitting for every $\tau \in \mathrm{Tid}$ all transition labels in indices $I_\tau$, and further omitting $it(j)$ if $\#(it(j)) = \#(it(i))$ for some $i \in I_\tau$ (that is, we remove the operations in $I_\tau$ and their corresponding propagation operations). Note that such $j$ can only exist if $\mathtt{typ}(it(i)) = \mathtt{FO}$. It is easy to see that $\langle m_0, P_\epsilon, B_\epsilon, \emptyset\rangle \xrightarrow{it'}_{i\mathrm{PTSO_{syn}}} \langle m, iP, B_\epsilon, S'\rangle$ for some $S'$ (in particular, all operations of threads $\pi \neq \tau$, as well as all propagation operations, are oblivious to the contents of $B(\tau)$). Going back to the non-instrumented system, by Lemmas B.3 and B.9, we obtain that $\langle m_0, P_\epsilon, B_\epsilon\rangle \xRightarrow{\Lambda(it')}_{\mathrm{PTSO_{syn}}} \langle m, \Lambda(iP), B_\epsilon\rangle$. It is also easy to see that our construction ensures that $\Lambda(it') \lesssim t$. □

# C PROOFS FOR SECTION 5

Lemma 5.14. *The following conditions together ensure that $M$ observationally refines $D$:*

*(i) For every $m_0$-initialized $M$-observable-trace $t$, there exists a $D$-consistent $m_0$-initialized execution graph $G$ such that $t \in \mathsf{traces}(G)$.*
*(ii) For every $m_0$-to-$m$ $M$-observable-trace $t$, there exist $t' \lesssim t$ and $D$-consistent $m_0$-initialized execution graph such that $t' \in \mathsf{traces}(G)$ and $m(G) = m$.*

Proof. Suppose that $\overline{q} \in Pr.\mathsf{Q}$ is reachable under $M$. Then, by definition, $\langle \overline{q}, m, \tilde{m} \rangle$ is reachable in $Pr \parallel M$ for some $\langle m, \tilde{m} \rangle \in M.\mathsf{Q}$. Thus, there exist crashless observable program traces $t_0, \dots ,t_n$, initial program states $\overline{q}_0, \dots ,\overline{q}_n \in Pr.\mathsf{Q}_{\mathsf{Init}}$, initial non-volatile memories $m_1, \dots ,m_n \in \mathsf{Loc} \to \mathsf{Val}$, and initial volatile states $\tilde{m}_0, \dots ,\tilde{m}_n \in M.\tilde{\mathsf{Q}}_{\mathsf{Init}}$, such that the following hold:

- $\langle \overline{q}_0, m_{\mathsf{Init}}, \tilde{m}_0 \rangle \overset{t_0}{\Longrightarrow}_{Pr \parallel M} \langle \_, m_1, \_ \rangle$, and $\langle \overline{q}_i, m_i, \tilde{m}_i \rangle \overset{t_i}{\Longrightarrow}_{Pr \parallel M} \langle \_, m_{i+1}, \_ \rangle$ for every $1 \leq i \leq n-1$.
- $\langle \overline{q}_n, m_n, \tilde{m}_n \rangle \overset{t_n}{\Longrightarrow}_{Pr \parallel M} \langle \overline{q}, \_, \_ \rangle$.

By Prop. A.1, it follows that:

- $\overline{q}_i \overset{t_i}{\Longrightarrow}_{Pr} \_$ for every $0 \leq i \leq n-1$, and $\overline{q}_n \overset{t_n}{\Longrightarrow}_{Pr} \overline{q}$.
- $t_0$ is an $m_{\mathsf{Init}}$-to-$m$ $M$-observable-trace, and $t_i$ is an $m_i$-to-$m_{i+1}$ $M$-observable-trace for every $1 \leq i \leq n-1$.
- $t_n$ is an $m_n$-initialized $M$-observable-trace.

Then, assumption (ii) entails that there exist $t'_0, \dots ,t'_{n-1}$ and $D$-consistent execution graphs $G_0, \dots ,G_{n-1}$ such that the following hold:

- $t'_i \lesssim t_i$ for every $0 \leq i \leq n-1$.
- $t'_i \in \mathsf{traces}(G_i)$ for every $0 \leq i \leq n-1$.
- $G_0$ is $m_{\mathsf{Init}}$-initialized and $m(G_0) = m_1$.
- For every $1 \leq i \leq n-1$, $G_i$ is $m_i$-initialized and $m(G_i) = m_{i+1}$.

Now, since $\overline{q}_i \overset{t_i}{\Longrightarrow}_{Pr} \_$ and $t'_i \lesssim t_i$ for every $0 \leq i \leq n-1$, by Prop. 2.4, we have $\overline{q}_i \overset{t'_i}{\Longrightarrow}_{Pr} \_$ for every $0 \leq i \leq n-1$. Since $t'_i \in \mathsf{traces}(G_i)$ for every $0 \leq i \leq n-1$, by Prop. 5.12, it follows that $G_i$ is generated by $Pr$ for every $0 \leq i \leq n-1$.

In addition, assumption (i) entails that there exists a $D$-consistent $m_n$-initialized execution graph $G_n$ such that $t_n \in \mathsf{traces}(G_n)$. Since $\overline{q}_n \overset{t_n}{\Longrightarrow}_{Pr} \overline{q}$, by Prop. 5.12, it follows that $G_n$ is generated by $Pr$ with final state $\overline{q}$.

It follows that $G_0, \dots ,G_n$ are $D$-consistent execution graphs that satisfy the conditions of Def. 5.13, so that $\overline{q}$ is reachable under $D$. □

Lemma 5.15. *If for every $D$-consistent initialized execution graph $G$, some $t \in \mathsf{traces}(G)$ is an $m_{\mathsf{Init}}(G)$-to-$m(G)$ $M$-observable-trace, then $D$ observationally refines $M$.*

Proof. Suppose that $\overline{q} \in Pr.\mathsf{Q}$ is reachable under $D$. Let $G_0, \dots ,G_n$ be $D$-consistent execution graphs that satisfy the conditions of Def. 5.13. Our assumption entails that there exist $t_0, \dots ,t_n$ such that for every $1 \leq i \leq n$, $t_i \in \mathsf{traces}(G_i)$ and $t_i$ is an $m_{\mathsf{Init}}(G_i)$-to-$m(G_i)$ $M$-observable-trace. Let $\tilde{m}_0, \dots ,\tilde{m}_n \in M.\tilde{\mathsf{Q}}_{\mathsf{Init}}$ such that $\langle m_{\mathsf{Init}}(G_i), \tilde{m}_i \rangle \overset{t_i}{\Longrightarrow}_{M} \langle m(G_i), \_ \rangle$ for every $1 \leq i \leq n$.

By Prop. 5.11, since $G_i$ is generated by $Pr$ for every $0 \leq i \leq n-1$, there exist initial program states $\overline{q}_0, \dots ,\overline{q}_{n-1} \in Pr.\mathsf{Q}_{\mathsf{Init}}$, such that $\overline{q}_i \overset{t_i}{\Longrightarrow}_{Pr} \_$ for every $0 \leq i \leq n-1$. Using Prop. A.1, it follows that $\langle \overline{q}_i, m_{\mathsf{Init}}(G_i), \tilde{m}_i \rangle \overset{t_i}{\Longrightarrow}_{Pr \parallel M} \langle \_, m(G_i), \_ \rangle$ for every $0 \leq i \leq n-1$.

In addition, since $G_n$ is generated by $Pr$ with final state $\overline{q}$, there exists initial program state $\overline{q}_n \in Pr.\mathsf{Q}_{\mathsf{Init}}$, such that $\overline{q}_n \overset{t_n}{\Longrightarrow}_{Pr} \overline{q}$. Using Prop. A.1, it follows that $\langle \overline{q}_n, m_{\mathsf{Init}}(G_n), \tilde{m}_n \rangle \overset{t_n}{\Longrightarrow}_{Pr \parallel M} \langle \overline{q}, m(G_n), \_ \rangle$.

Now, since $m_{\mathsf{Init}}(G_0) = m_{\mathsf{Init}}$ and $m_{\mathsf{Init}}(G_i) = m(G_{i-1})$ for every $1 \leq i \leq n$, it follows that $\langle \overline{q}, m(G_n), \tilde{m} \rangle$ is reachable in $Pr \parallel M$ for some $\tilde{m} \in M.\tilde{\mathsf{Q}}$. □

The following property of $\mathsf{ppo}$ is useful below:

Lemma C.1. $G.\mathsf{ppo}\,;[\mathsf{R}]\,;G.\mathsf{po} \subseteq G.\mathsf{ppo}$.

Lemma 5.24. *Let $\mathit{tpo}$ be a propagation order for an execution graph $G$ for which the conditions of Def. 5.19 hold. Then, $G.\mathsf{ppo} \cup G.\mathtt{rfe} \cup \mathit{tpo} \cup G.\mathtt{fr}(\mathit{tpo})$ is acyclic.*

Proof. In this proof we consider a single graph $G$, and thus omit the "$G.$" prefix from all notations.

Consider a cycle in $\mathsf{ppo} \cup \mathtt{rfe} \cup \mathit{tpo} \cup \mathtt{fr}(\mathit{tpo})$ of minimal length. The fact that $\mathit{tpo}$ is total on $\mathsf{P}$ and the minimality of the cycle imply that this cycle may contain at most two events in $\mathsf{P}$.

If the cycle contains no events in $\mathsf{P}$, then it must consist solely of $\mathsf{ppo}$-edges, which contradict the fact that $\mathsf{po}$ is irreflexive.

If the cycle contains one event in $\mathsf{P}$, then we must have $\langle e, e\rangle \in (\mathsf{ppo} \cup \mathtt{rfe})\,;\mathsf{ppo}^{+}\,;(\mathsf{ppo} \cup \mathtt{fr}(\mathit{tpo}))$ for some $e \in \mathsf{E}$, which implies that one of the following holds:

(*i*) $\langle e, e\rangle \in \mathsf{ppo}^{+} \subseteq \mathsf{po}$,
(*ii*) $\langle e, e\rangle \in \mathsf{ppo}^{+}\,;\mathtt{fr}(\mathit{tpo}) \subseteq \mathsf{po}\,;\mathtt{fr}(\mathit{tpo})$,
(*iii*) $\langle e, e\rangle \in \mathtt{rfe}\,;\mathsf{ppo}^{+} \subseteq \mathtt{rfe}\,;\mathsf{po}$, or
(*iv*) $\langle e, e\rangle \in \mathtt{rfe}\,;\mathsf{ppo}^{+}\,;\mathtt{fr}(\mathit{tpo}) \subseteq \mathtt{rfe}\,;\mathsf{po}\,;\mathtt{fr}(\mathit{tpo})$.

Each of these options contradicts one of the conditions of Def. 5.19.

Finally, suppose that the cycle contains two events in $\mathsf{P}$. Then, from the fact that $\mathit{tpo}$ is total on $\mathsf{P}$, there must exist some $\langle e_1, e_2\rangle \in \mathit{tpo}$, such that $\langle e_2, e_1\rangle \in \mathsf{ppo} \cup \mathtt{rfe} \cup \mathtt{fr}(\mathit{tpo})$ or $\langle e_2, e_1\rangle \in (\mathsf{ppo} \cup \mathtt{rfe})\,;[\mathsf{R}]\,;\mathsf{ppo}^{*}\,;(\mathsf{ppo} \cup \mathtt{fr}(\mathit{tpo}))$. The first case leads to a contradiction since the conditions of Def. 5.19 ensure that $\mathit{tpo}\,;\mathsf{ppo}$, $\mathit{tpo}\,;\mathtt{rfe}$, and $\mathit{tpo}\,;\mathtt{fr}(\mathit{tpo})$ are all irreflexive. It follows that one of the following holds:

(*i*) $\langle e_2, e_2\rangle \in \mathsf{ppo}\,;[\mathsf{R}]\,;\mathsf{ppo}^{+}\,;\mathit{tpo} \subseteq \mathsf{ppo}\,;[\mathsf{R}]\,;\mathsf{po}\,;\mathit{tpo} \subseteq \mathsf{ppo}\,;\mathit{tpo}$ (by Lemma C.1),
(*ii*) $\langle e_2, e_2\rangle \in \mathsf{ppo}\,;[\mathsf{R}]\,;\mathsf{ppo}^{*}\,;\mathtt{fr}(\mathit{tpo})\,;\mathit{tpo}$,
(*iii*) $\langle e_2, e_2\rangle \in \mathtt{rfe}\,;\mathsf{ppo}^{+}\,;\mathit{tpo} \subseteq \mathtt{rfe}\,;\mathsf{po}\,;\mathit{tpo}$, or
(*iv*) $\langle e_2, e_2\rangle \in \mathtt{rfe}\,;\mathsf{ppo}^{*}\,;\mathtt{fr}(\mathit{tpo})\,;\mathit{tpo} \subseteq \mathit{tpo} \cup \mathtt{rfe}\,;\mathsf{po}\,;\mathtt{fr}(\mathit{tpo})\,;\mathit{tpo}$.

As before, each of these options contradicts one of the conditions of Def. 5.19. The least trivial case is (*ii*): suppose that $\langle e_2, e_2\rangle \in \mathsf{ppo}\,;[\mathsf{R}]\,;\mathsf{ppo}^{*}\,;\mathtt{fr}(\mathit{tpo})\,;\mathit{tpo}$. Then, it must be the case that $e_2 \in \mathsf{RMW} \cup \mathsf{R\text{-}ex} \cup \mathsf{MF}$, and so $\langle e_2, e_2\rangle \in \mathsf{po}\,;\mathtt{fr}(\mathit{tpo})\,;\mathit{tpo}\,;[\mathsf{RMW} \cup \mathsf{R\text{-}ex} \cup \mathsf{MF}]$, which contradicts Def. 5.19. □

Theorem 5.29 is obtained from the following two theorems (one for each direction):

Theorem C.2. $\mathsf{PTSO_{syn}}$ *observationally refines* $\mathsf{DPTSO_{syn}}$.

Proof (outline). Using Lemma 5.14, it suffices to show that:

- For every $m_0$-initialized $\mathsf{PTSO_{syn}}$-observable-trace $t$, there exists a $\mathsf{DPTSO_{syn}}$-consistent $m_0$-initialized execution graph $G$ such that $t \in \mathsf{traces}(G)$.
- For every $m_0$-to-$m$ $\mathsf{PTSO_{syn}}$-observable-trace $t$, there exist $t' \lesssim t$ and $m_0$-initialized $\mathsf{DPTSO_{syn}}$-consistent execution graph such that $t' \in \mathsf{traces}(G)$ and $m(G) = m$.

Using Lemma 4.5, it suffices to prove that $\langle m_0, P_\epsilon, B_\epsilon\rangle \xRightarrow{t}_{\mathsf{PTSO_{syn}}} \langle m, P, B_\epsilon\rangle$ implies that there exists a $\mathsf{DPTSO_{syn}}$-consistent $m_0$-initialized execution graph $G$ such that $t \in \mathsf{traces}(G)$ and $m(G) = m$. Suppose that $\langle m_0, P_\epsilon, B_\epsilon\rangle \xRightarrow{t}_{\mathsf{PTSO_{syn}}} \langle m, P, B_\epsilon\rangle$. We construct a $\mathsf{DPTSO_{syn}}$-consistent $m_0$-initialized execution graph $G$ such that $t \in \mathsf{traces}(G)$ and $m(G) = m$.

We use the instrumented semantics ($i\mathsf{PTSO_{syn}}$). By Lemmas B.3 and B.9, we have $\langle m_0, P_\epsilon, B_\epsilon, \emptyset\rangle \xRightarrow{it}_{i\mathsf{PTSO_{syn}}} \langle m, iP, B_\epsilon, S\rangle$ for some $it$ such that $\Lambda(it) = t$, $iP$, and $S \subseteq \mathbb{N}$. We use the (instrumented) trace $it$ to construct $G$:

- Events: For every $1 \leq i \leq |it|$ with $it(i) = \langle \tau, l\#\_ \rangle$ and $\mathtt{typ}(l) \in \{\mathtt{W}, \mathtt{R}, \mathtt{RMW}, \mathtt{R\text{-}ex}, \mathtt{MF}, \mathtt{FL}, \mathtt{FO}, \mathtt{SF}\}$, we include the event $e_i \triangleq \langle \tau, i, l \rangle$ in $G.\mathsf{E}$. In addition, we include the initialization events $e_x \triangleq \langle \bot, 0, \mathtt{W}(x, m_0(x)) \rangle$ for every $x \in \mathsf{Loc}$. It is easy to see that we have $t \in \mathrm{traces}(G)$ and that $G$ is $m_0$-initialized.
- Reads-from: $G.\mathtt{rf}$ is constructed as follows: for every $1 \leq i \leq |it|$ with $\mathtt{typ}(e_i) \in \{\mathtt{R}, \mathtt{RMW}, \mathtt{R\text{-}ex}\}$ and $\mathtt{loc}(e_i) = x$, we locate the last index $1 \leq j < i$ such that $\mathtt{typ}(e_j) = \mathtt{W}$, $\mathtt{loc}(e_j) = x$, $\mathtt{tid}(e_j) = \mathtt{tid}(e_i)$ and there does not exist an index $j < k < i$ such that $\#(it(k)) = \#(it(j))$ (namely, the write that corresponds to $e_j$ was not propagated from the store buffer when the read that corresponds to $e_i$ was executed), and include an edge $\langle e_j, e_i \rangle$ in $G.\mathtt{rf}$. If such an index $j$ does not exist, we further locate the last index $1 \leq k < i$ such that such that $\mathtt{typ}(e_j) \in \{\mathtt{RMW}, \mathtt{PropW}\}$ and $\mathtt{loc}(e_j) = x$, and include an edge $\langle e_j, e_i \rangle$ in $G.\mathtt{rf}$, where $j$ is the unique index satisfying $j < k$ and $\#(it(j)) = \#(it(k))$, or $j = k$ in case $\mathtt{typ}(e_j) = \mathtt{RMW}$. Finally, if such index $k$ does not exist as well, we include the edge $\langle e_x, e_i \rangle$ in $G.\mathtt{rf}$ (reading from the initialization event). Using $i$PTSO$_{\mathsf{syn}}$'s operational semantics, it is easy to verify that $G.\mathtt{rf}$ is indeed a reads-from relation for $G.\mathsf{E}$.
- Memory assignment: To define $G.\mathsf{M}$, for every $x \in \mathsf{Loc}$, let $i(x)$ be the maximal index such that $\mathtt{typ}(it(i(x))) = \mathtt{PerW}$ and $\mathtt{loc}(it(i(x))) = x$ (that is, $i(x)$ is the index of the last propagation to the persistent memory of a write to $x$). In addition, let $w(i(x))$ be the (unique) index $k$ such that $\mathtt{typ}(it(k)) \in \{\mathtt{W}, \mathtt{RMW}\}$ and $\#(it(k)) = \#(it(i(x)))$ (that is, $w(i(x))$ is the index of the write operation that persists in index $i(x)$). Now, we define $G.\mathsf{M}(x) \triangleq e_{w(i(x))}$ for every $x \in \mathsf{Loc}$ for which $i(x)$ is defined. If $i(x)$ is undefined ($\mathtt{typ}(it(i) = \mathtt{PerW}$ and $\mathtt{loc}(it(i)) = x$ never hold), we set $G.\mathsf{M}(x) \triangleq e_x$ (the initialization event of $x$). Then, we clearly have $m(G) = m$.

To show that $G$ is DPTSO$_{\mathsf{syn}}$-consistent, we construct a propagation order $tpo$ for $G$. First, for every $1 \leq i \leq |it|$ with $\mathtt{typ}(e_i) \in \{\mathtt{W}, \mathtt{FL}, \mathtt{FO}, \mathtt{SF}\}$, let $tp(i)$ denote the (unique) index $k$ such that $\mathtt{typ}(it(k)) \in \{\mathtt{PropW}/\mathtt{PropFL}/\mathtt{PropFO}/\mathtt{PropSF}\}$ and $\#(it(k)) = \#(it(i))$ (that is, $tp(i)$ is the index of the propagation from the store buffer of the operation in index $i$). In addition, for every $1 \leq i \leq |it|$ with $\mathtt{typ}(e_i) \in \{\mathtt{RMW}, \mathtt{R\text{-}ex}, \mathtt{MF}\}$, we let $tp(i) \triangleq i$. Now, $tpo$ is constructed as follows: for every $e_i, e_j \in G.\mathsf{P}$, we include $\langle e_i, e_j \rangle \in tpo$ iff $tp(i) < tp(j)$. In addition, we include in $tpo$ some arbitrary total order on $G.\mathsf{E} \cap \mathsf{Init}$, as well as pairs ordering all initialization events before all non-initialization events. It is straightforward to verify that this construction satisfies the (local) properties of Def. 5.19 yielding a DPTSO$_{\mathsf{syn}}$-consistent graph:

(1) For every $a, b \in \mathsf{P}$, except for the case that $a \in \mathsf{W} \cup \mathsf{FL} \cup \mathsf{FO}$, $b \in \mathsf{FO}$, and $\mathtt{loc}(a) \neq \mathtt{loc}(b)$, if $\langle a, b \rangle \in G.\mathsf{po}$, then $\langle a, b \rangle \in tpo$: Let $a, b \in \mathsf{P}$ such that $\langle a, b \rangle \in G.\mathsf{po}$. Suppose that it is not the case that $a \in \mathsf{W} \cup \mathsf{FL} \cup \mathsf{FO}$, $b \in \mathsf{FO}$, and $\mathtt{loc}(a) \neq \mathtt{loc}(b)$. First, if $a$ is an initialization event, then by definition we have $\langle a, b \rangle \in tpo$ ($b$ cannot be an initialization event in this case). Otherwise, we have that $a = e_i$ and $b = e_j$ for some $1 \leq i < j \leq |it|$ such that $\mathtt{tid}(e_i) = \mathtt{tid}(e_j)$. Since $i$PTSO$_{\mathsf{syn}}$ propagates the entries from the persistent buffer in the same order they were issued, except for the case of an $\mathtt{FO}$-entry that may propagate before previously-issued $\mathtt{W}/\mathtt{FL}/\mathtt{FO}$-entries to a different location, it must be the case that $tp(i) < tp(j)$, and so we have $\langle a, b \rangle = \langle e_i, e_j \rangle \in tpo$.

(2) $tpo^? \mathbin{;} G.\mathtt{rfe} \mathbin{;} G.\mathsf{po}^?$ is irreflexive: First, we show that $G.\mathtt{rfe} \mathbin{;} G.\mathsf{po}^?$ is irreflexive. Suppose that $\langle a, b \rangle \in G.\mathtt{rfe}$ and $\langle b, a \rangle \in G.\mathsf{po}^?$. Then, we have that $a = e_j$ and $b = e_i$ for some $1 \leq i \leq j \leq |it|$ such that $\mathtt{tid}(e_i) = \mathtt{tid}(e_j)$ (note that initialization events do not have incoming $\mathsf{po}$ or $\mathtt{rf}$-edges). However, $\langle e_j, e_i \rangle \in G.\mathtt{rf}$ implies that $j < i$. Now, suppose that $\langle a, b \rangle \in tpo$, $\langle b, c \rangle \in G.\mathtt{rfe}$, and $\langle c, a \rangle \in G.\mathsf{po}^?$. Then, it follows that $a = e_i$, $b = e_j$, and $c = e_k$ for some $1 \leq i, j, k \leq |it|$ such that $\mathtt{tid}(e_k) = \mathtt{tid}(e_i)$, $k \leq i$, and $tp(i) < tp(j)$. Then, since we do not have $\langle e_j, e_k \rangle \in G.\mathsf{po} \cup G.\mathsf{po}^{-1}$, we cannot have $\tau(e_j) = \tau(e_k)$. Then, the

construction of $G.\mathtt{rf}$ ensures that $tp(j) < k$. It follows that $tp(i) < k$. Since $i \leq tp(i)$, this contradicts the fact that $k \leq i$.

(3) $G.\mathtt{fr}(tpo)\ ;G.\mathtt{rfe}^?\ ;G.\mathsf{po}$ is irreflexive: From the construction of $G.\mathtt{rf}$, it is easy to verify that $\langle e_i, e_j\rangle \in G.\mathtt{fr}(tpo)$ implies that $i < tp(j)$. Now, suppose that $\langle a, b\rangle \in G.\mathtt{fr}(tpo)$ and $\langle b, a\rangle \in G.\mathsf{po}$. Then, $a = e_j$ and $b = e_i$ for some $1 \leq i \leq j \leq |it|$ such that $\mathtt{tid}(e_i) = \mathtt{tid}(e_j)$ and $j < tp(i)$. It follows that $i < tp(i)$ which contradicts our construction. Finally, suppose that $\langle a, b\rangle \in G.\mathtt{fr}(tpo)$, $\langle b, c\rangle \in G.\mathtt{rfe}$, and $\langle c, a\rangle \in G.\mathsf{po}$. Then, it follows that $a = e_i$, $b = e_j$, and $c = e_k$ for some $1 \leq i, j, k \leq |it|$ such that $\mathtt{tid}(e_k) = \mathtt{tid}(e_i)$, $k \leq i$, and $i < tp(j)$. As in the previous item, we have that $tp(j) < k$, which leads to a contradiction.

(4) $G.\mathtt{fr}(tpo)\ ;tpo$ is irreflexive: Suppose that $\langle a, b\rangle \in G.\mathtt{fr}(tpo)$ and $\langle b, a\rangle \in tpo$. Then, $a = e_j$ and $b = e_i$ for some $1 \leq i, j \leq |it|$ such that $i < tp(j)$ and $tp(j) \leq tp(i)$. It follows that $i < tp(i)$, which contradicts our construction.

(5) $G.\mathtt{fr}(tpo)\ ;tpo\ ;G.\mathtt{rfe}\ ;G.\mathsf{po}$ is irreflexive: Suppose that $\langle a, b\rangle \in G.\mathtt{fr}(tpo)$, $\langle b, c\rangle \in tpo$, $\langle c, d\rangle \in G.\mathtt{rfe}$, and $\langle d, a\rangle \in G.\mathsf{po}$. Then, it follows that $a = e_i$, $b = e_j$, $c = e_k$, and $d = e_m$ for some $1 \leq i, j, k, m \leq |it|$ such that $\mathtt{tid}(e_m) = \mathtt{tid}(e_i)$, $m < i$, $i < tp(j)$, $tp(j) < tp(k)$, and $tp(k) < m$. Clearly, these inequalities lead to a contradiction.

(6) $G.\mathtt{fr}(tpo)\ ;tpo\ ;[\mathsf{RMW} \cup \mathsf{R\text{-}ex} \cup \mathsf{MF}]\ ;G.\mathsf{po}$ is irreflexive: Suppose that $\langle a, b\rangle \in G.\mathtt{fr}(tpo)$, $\langle b, c\rangle \in tpo$, $c \in \mathsf{RMW} \cup \mathsf{R\text{-}ex} \cup \mathsf{MF}$, and $\langle c, a\rangle \in G.\mathsf{po}$. Then, it follows that $a = e_i$, $b = e_j$, $c = e_k$ for some $1 \leq i, j, k \leq |it|$ such that $\mathtt{tid}(e_k) = \mathtt{tid}(e_i)$, $k < i$, $i < tp(j)$, $tp(j) < tp(k)$. However, since $c \in \mathsf{RMW} \cup \mathsf{R\text{-}ex} \cup \mathsf{MF}$, we have $tp(k) = k$, and, as before, these inequalities lead to a contradiction.

(7) $G.\mathsf{dtpo}(tpo)\ ;tpo$ is irreflexive: Suppose that $\langle a, b\rangle \in G.\mathsf{dtpo}(tpo)$ and $\langle b, a\rangle \in tpo$. By definition, there is a location $x \in \mathsf{Loc}$ such that

$$a \in G.\mathsf{FLO}_x = G.\mathsf{FL}_x \cup (\mathsf{FO}_x \cap dom(G.\mathsf{po}\ ;[\mathsf{RMW} \cup \mathsf{R\text{-}ex} \cup \mathsf{MF} \cup \mathsf{SF}])),$$

$b \in \mathsf{W}_x \cup \mathsf{RMW}_x$, and $\langle G.\mathsf{M}(x), b\rangle \in tpo$. Then, $a = e_j$ and $b = e_i$ for some $1 \leq i, j \leq |it|$ such that $tp(i) < tp(j)$. Now, if $a$ is a flush event, the flush step in index $j$ can only exist if the write entry that corresponds to $b$ has persisted. Hence, $i(x)$ is defined, and we have $G.\mathsf{M}(x) = e_{w(i(x))}$. In addition, $\langle G.\mathsf{M}(x), b\rangle \in tpo$ implies that $tp(w(i(x))) \leq tp(i)$. However, since the persistence order (on each location) must follow the order in which the write propagated from the store buffer, the write entry that corresponds to $b$ must persist after the write entry that corresponds to $G.\mathsf{M}(x)$, which contradicts the construction of $G.\mathsf{M}$. The case that $a$ is a flush-optimal event followed by an $\mathsf{RMW} \cup \mathsf{R\text{-}ex} \cup \mathsf{MF} \cup \mathsf{SF}$-event of the same thread is handled similarly. □

Theorem C.3. $\mathsf{DPTSO}_{\mathsf{syn}}$ *observationally refines* $\mathsf{PTSO}_{\mathsf{syn}}$.

Proof (outline). By Lemma 5.15, is suffices to show that for every $\mathsf{DPTSO}_{\mathsf{syn}}$-consistent initialized execution graph $G$, some $t \in \mathsf{traces}(G)$ is an $m_{\mathsf{Init}}(G)$-to-$m(G)$ $\mathsf{PTSO}_{\mathsf{syn}}$-observable-trace. By Lemmas B.3 and B.9, we may use the instrumented system $i\mathsf{PTSO}_{\mathsf{syn}}$ and show that there exists an $m_{\mathsf{Init}}(G)$-to-$m(G)$ $i\mathsf{PTSO}_{\mathsf{syn}}$-trace $it$ such that $\Lambda(it) \in \mathsf{traces}(G)$.

Let $G$ be a $\mathsf{DPTSO}_{\mathsf{syn}}$-consistent execution graph, and let $tpo$ be a propagation order for $G$ that satisfies the conditions of Def. 5.19. Let $F$ be some injective function from events to $\mathbb{N}$ (we will use it to assign identifiers to the different operations). For every event $e \in \mathsf{E}$, we associate three transition labels $\alpha(e), \beta(e), \gamma(e)$:

- Issue of $e$: $\alpha(e) = \langle \mathtt{tid}(e), \mathtt{lab}(e)\#F(e)\rangle$.

- Propagation of $e$ from store buffer to persistence buffer (only defined for $e \in \mathsf{W} \cup \mathsf{FL} \cup \mathsf{FO} \cup \mathsf{SF}$):
$$\beta(e) = \begin{cases} \langle \mathtt{tid}(e), \mathtt{PropW}(\mathtt{loc}(e))\#F(e)\rangle & e \in \mathsf{W} \\ \langle \mathtt{tid}(e), \mathtt{PropFL}(\mathtt{loc}(e))\#F(e)\rangle & e \in \mathsf{FL} \\ \langle \mathtt{tid}(e), \mathtt{PropFO}(\mathtt{loc}(e))\#F(e)\rangle & e \in \mathsf{FO} \\ \langle \mathtt{tid}(e), \mathtt{PropSF}\#F(e)\rangle & e \in \mathsf{SF} \end{cases}$$
- Propagation of $e$ from persistence buffer to persistent memory (only defined for $e \in \mathsf{W} \cup \mathsf{RMW} \cup \mathsf{FO}$): $\gamma(e) = \begin{cases} \mathtt{PerW}(\mathtt{loc}(e))\#F(e) & e \in \mathsf{W} \cup \mathsf{RMW} \\ \mathtt{PerFO}(\mathtt{loc}(e))\#F(e) & e \in \mathsf{FO} \end{cases}$

Using these definition, we construct a set $A$ of transition labels of $i\mathrm{PTSO}_{\mathsf{syn}}$. Let:

- $E_\alpha = G.\mathsf{E} \setminus \mathsf{Init}$.
- $E_\beta = (G.\mathsf{W} \setminus \mathsf{Init}) \cup G.\mathsf{FL} \cup G.\mathsf{FO} \cup G.\mathsf{SF}$.
- $E_\gamma^{\mathsf{W}_x} = \{w \in (\mathsf{W}_x \setminus \mathsf{Init}) \cup \mathsf{RMW}_x \mid \langle w, G.\mathsf{M}(x)\rangle \in tpo^?\}$.
- $E_\gamma^{\mathsf{W}} = \bigcup_{x \in \mathsf{Loc}} E_\gamma^{\mathsf{W}_x}$.
- $E_\gamma^{\mathsf{FO}_x} = \mathsf{FO}_x \cap (dom(tpo^? ; G.\mathsf{po} ; [\mathsf{RMW} \cup \mathsf{R\text{-}ex} \cup \mathsf{MF} \cup \mathsf{SF}]) \cup dom(tpo ; [\mathsf{FL}_x \cup \{G.\mathsf{M}(x)\}])$.
- $E_\gamma^{\mathsf{FO}} = \bigcup_{x \in \mathsf{Loc}} E_\gamma^{\mathsf{FO}_x}$.
- $E_\gamma = E_\gamma^{\mathsf{W}} \cup E_\gamma^{\mathsf{FO}}$.

We define
$$A = \{\alpha(e) \mid e \in E_\alpha\} \cup \{\beta(e) \mid e \in E_\beta\} \cup \{\gamma(e) \mid e \in E_\gamma\}.$$

Next, we construct an enumeration of $A$ which will serve as $it$. Let $R$ be the union of the following relations on $A$:

- $R_1 = \{\langle \alpha(e), \beta(e)\rangle \mid e \in E_\beta\}$
- $R_2 = \{\langle \beta(e), \gamma(e)\rangle \mid e \in E_\gamma\}$
- $R_3 = \{\langle \alpha(e_1), \alpha(e_2)\rangle \mid \langle e_1, e_2\rangle \in [E_\alpha] ; G.\mathsf{po}\}$
- $R_4 = \{\langle \beta(e_1), \beta(e_2)\rangle \mid \langle e_1, e_2\rangle \in [E_\beta] ; tpo ; [E_\beta]\}$
- $R_5 = \{\langle \alpha(e_1), \beta(e_2)\rangle \mid \langle e_1, e_2\rangle \in [\mathsf{RMW} \cup \mathsf{R\text{-}ex} \cup \mathsf{MF}] ; tpo ; [E_\beta]\}$
- $R_6 = \{\langle \beta(e_1), \alpha(e_2)\rangle \mid \langle e_1, e_2\rangle \in [E_\beta] ; tpo ; [\mathsf{RMW} \cup \mathsf{R\text{-}ex} \cup \mathsf{MF}]\}$
- $R_7 = \{\langle \beta(e_1), \alpha(e_2)\rangle \mid \langle e_1, e_2\rangle \in [E_\beta] ; G.\mathtt{rfe}\}$
- $R_8 = \{\langle \alpha(e_1), \alpha(e_2)\rangle \mid \langle e_1, e_2\rangle \in [\mathsf{RMW}] ; G.\mathtt{rfe}\}$
- $R_9 = \{\langle \alpha(e_1), \beta(e_2)\rangle \mid \langle e_1, e_2\rangle \in G.\mathtt{fr}(tpo) ; [E_\beta]\}$
- $R_{10} = \{\langle \alpha(e_1), \alpha(e_2)\rangle \mid \langle e_1, e_2\rangle \in G.\mathtt{fr}(tpo) ; [\mathsf{RMW}]\}$
- $R_{11} = \{\langle \gamma(e_1), \beta(e_2)\rangle \mid \langle e_1, e_2\rangle \in [E_\gamma] ; tpo ; [\mathsf{FL}]\}$
- $R_{12} = \{\langle \gamma(e_1), \beta(e_2)\rangle \mid \langle e_1, e_2\rangle \in [E_\gamma^{\mathsf{FO}}] ; G.\mathsf{po} ; [\mathsf{SF}]\}$
- $R_{13} = \{\langle \gamma(e_1), \alpha(e_2)\rangle \mid \langle e_1, e_2\rangle \in [E_\gamma^{\mathsf{FO}}] ; G.\mathsf{po} ; [\mathsf{RMW} \cup \mathsf{R\text{-}ex} \cup \mathsf{MF}]\}$
- $R_{14} = \{\langle \gamma(e_1), \gamma(e_2)\rangle \mid \langle e_1, e_2\rangle \in [E_\gamma] ; tpo ; [E_\gamma]\}$

It is standard to verify that for any enumeration $it$ of $R$, we have $\Lambda(it) \in \mathrm{traces}(G)$ and that $it$ is an $m_{\mathsf{Init}}(G)$-to-$m(G)$ $i\mathrm{PTSO}_{\mathsf{syn}}$-trace. In particular, let $x \in \mathsf{Loc}$ and suppose that for the last transition label of the form $\mathtt{PerW}(x)\#\_$ in $it$ is not $\mathtt{PerW}(x)\#F(G.\mathsf{M}(x))$, but rather $\mathtt{PerW}(x)\#F(w)$ for some $w \in E_\gamma^{\mathsf{W}} \setminus \{G.\mathsf{M}(x)\}$. Then, since $w \in E_\gamma^{\mathsf{W}}$ we have $\langle w, G.\mathsf{M}(x)\rangle \in tpo^?$, which contradicts the fact that $R_{14} \subseteq R$. The proof that $it$ is indeed an $i\mathrm{PTSO}_{\mathsf{syn}}$-trace is performed by induction: assume that a prefix $it'$ of $it$ is an $i\mathrm{PTSO}_{\mathsf{syn}}$-trace, show that it can be extended with one more label from $it$. For that matter, the claim has to be strengthened to relate the prefix $it'$ with the state that $i\mathrm{PTSO}_{\mathsf{syn}}$ reaches. This state, denoted by $\langle m_{it'}, iP_{it'}, iB_{it'}, S_{it'}\rangle$, is constructed as follows:

- Persistent memory: For every $x \in \mathsf{Loc}$, let $e_x \in E^{\mathsf{W}}_{\gamma} \cap \mathsf{E}_x$ such that $\gamma(e_x)$ is the last occurrence in $it'$ of a transition label of the form $\mathtt{PerW}(x)\#\_$. If no transition of the form $\mathtt{PerW}(x)\#\_$ occurs in $it'$, let $e_x$ be the initialization write to $x$ in $G$ (i.e., $m_{\mathsf{Init}}(G)(x)$). Then, $m_{it'} = \lambda x.\ \mathtt{val}_{\mathtt{W}}(e_x)$.
- Instrumented persistent buffers: For every location $x$, we include in $iP_{it'}(x)$ all entries of the following forms:
  - $\mathtt{W}(\mathtt{loc}(e), \mathtt{val}_{\mathtt{W}}(e))\#\#(e)$ for some $e \in G.\mathsf{W}_x$ such that $\beta(e) \in it'$ and $\gamma(e) \notin it'$.
  - $\mathtt{W}(\mathtt{loc}(e), \mathtt{val}_{\mathtt{W}}(e))\#\#(e)$ for some $e \in G.\mathsf{RMW}_x$ such that $\alpha(e) \in it'$ and $\gamma(e) \notin it'$.
  - $\mathtt{FO}(\mathtt{tid}(e))\#\#(e)$ for some $e \in G.\mathsf{FO}_x$ such that $\beta(e) \in it'$ and $\gamma(e) \notin it'$.

  Denote the instrumented entry related to event $e$ by $entry(e)$. Then, $entry(e_1)$ appears before $entry(e_2)$ in $iP_{it'}(x)$ iff one of the following hold:
  - If $e_1, e_2 \notin G.\mathsf{RMW}_x$ and $\beta(e_1)$ appears before $\beta(e_2)$ in $it'$.
  - If $e_1 \notin G.\mathsf{RMW}_x$, $e_2 \in G.\mathsf{RMW}_x$, and $\beta(e_1)$ appears before $\alpha(e_2)$ in $it'$.
  - If $e_1 \in G.\mathsf{RMW}_x$, $e_2 \notin G.\mathsf{RMW}_x$, and $\alpha(e_1)$ appears before $\beta(e_2)$ in $it'$.
  - If $e_1, e_2 \in G.\mathsf{RMW}_x$ and $\alpha(e_1)$ appears before $\alpha(e_2)$ in $it'$.
- Instrumented store buffers: For every thread identifier $\tau$, we include in $iB_{it'}(\tau)$ all entries of the following forms:
  - $\mathtt{W}(\mathtt{loc}(e), \mathtt{val}_{\mathtt{W}}(e))\#\#(e)$ for some $e \in G.\mathsf{W}^{\tau}$ such that $\alpha(e) \in it'$ and $\beta(e) \notin it'$.
  - $\mathtt{FL}(\mathtt{loc}(e))\#\#(e)$ for some $e \in G.\mathsf{FL}^{\tau}$ such that $\alpha(e) \in it'$ and $\beta(e) \notin it'$.
  - $\mathtt{FO}(\mathtt{loc}(e))\#\#(e)$ for some $e \in G.\mathsf{FO}^{\tau}$ such that $\alpha(e) \in it'$ and $\beta(e) \notin it'$.
  - $\mathtt{SF}\#\#(e)$ for some $e \in G.\mathsf{SF}^{\tau}$ such that $\alpha(e) \in it'$ and $\beta(e) \notin it'$.

  Denote the instrumented entry related to event $e$ by $entry(e)$. Then, $entry(e_1)$ appears before $entry(e_2)$ in $iB_{it'}(\tau)$ iff $\alpha(e_1)$ appears before $\alpha(e_2)$ in $it'$.
- $S_{it'}$ is the set of all identifiers used in $it'$.

It remains to show that $R$ is acyclic. Clearly, a cycle in $R_3$ induces a $G.\mathsf{po}$-cycle, and so $R_3$ is acyclic. Now, since $R_3$ is transitive, we can assume that any use of $R_3$ in an $R$-cycle follows an $R_i$-step with $i \neq 3$. It follows that any use of $R_3$ in an $R$-cycle must start in a transition label $\alpha(e)$ for some $e \in G.\mathsf{R} \cup G.\mathsf{RMW} \cup G.\mathsf{R}\text{-}\mathsf{ex} \cup G.\mathsf{MF}$. Hence, any $R$-cycle induces cycle in $G.\mathtt{ppo} \cup G.\mathtt{rfe} \cup tpo \cup G.\mathtt{fr}(tpo)$, which is acyclic by Lemma 5.24. □

## D PROOFS FOR SECTION 6

For the proofs in this section, we use the instrumented persistent memory subsystem (see Appendix B.1) $i$PSC, presented in Fig. 9. The functions $\mathtt{tid}$, $\mathtt{typ}$, $\mathtt{loc}$ are extended to $i\mathrm{PSC}.i\Sigma$ in the obvious way (in particular, for $\alpha \in i\mathrm{PSC}.i\Sigma$, we have $\mathtt{typ}(\alpha) \in \{\mathtt{PerW}/\mathtt{PerFO}\}$).

It is easy to see that $i$PSC is an instrumentation of PSC (see Def. B.8 for the definition of an erasure of an instrumented per-location persistence buffer).

Lemma D.1. *$i$PSC is a $\Lambda$-instrumentation of* PSC *for* $\Lambda \triangleq \lambda\langle iP, S\rangle.\ \Lambda(iP)$.

## E PROOFS FOR SECTION 6.1

The next lemmas are used to prove Thm. 6.2.

Lemma E.1. *Every $m_0$-to-$m$* $\mathrm{PSC}_{\mathtt{fin}}$*-observable-trace $t$ is also an $m_0$-to-$m$* PSC*-observable-trace.*

Proof (outline). We use a standard forward simulation argument. A simulation relation $R \subseteq \mathrm{PSC}_{\mathtt{fin}}.\mathtt{Q} \times \mathrm{PSC}.\mathtt{Q}$ is defined as follows: $\langle\langle m_f, \tilde{m}, L, T\rangle, \langle m, P\rangle\rangle \in R$ if the following hold:

- $m_f = m$.
- For every $x \in \mathsf{Loc}$, $\tilde{m}(x) = \mathsf{get}(m, P(x))(x)$.
- $x \in L$ iff $P(x) = \epsilon$.
- $\tau \in T$ iff $\forall y.\ \mathtt{FO}(\tau) \notin P(y)$.

$$i\mathsf{PSC}.i\Sigma \triangleq \{\mathtt{PerW}(x)\#s \mid x \in \mathsf{Loc}, s \in \mathbb{N}\} \cup \{\mathtt{PerFO}(x)\#s \mid x \in \mathsf{Loc}, s \in \mathbb{N}\}$$

$$m \in \mathsf{Loc} \to \mathsf{Val} \qquad iP \in \mathsf{Loc} \to (\{\mathtt{W}(x,v)\#s \mid x \in \mathsf{Loc}, v \in \mathsf{Val}, s \in \mathbb{N}\} \cup \{\mathtt{FO}(\tau)\#s \mid \tau \in \mathsf{Tid}, s \in \mathbb{N}\})^*$$

$$iP_{\mathsf{Init}} \triangleq \lambda x.\ \epsilon \qquad S_{\mathsf{Init}} = \emptyset$$

$$\textsc{write}\quad \frac{\begin{array}{c} S' = S \uplus \{s\} \\ l = \mathtt{W}(x,v) \\ iP' = iP[x \mapsto iP(x) \cdot \mathtt{W}(v)\#s] \end{array}}{\langle m, iP, S\rangle \xrightarrow{\tau, l\#s}_{i\mathsf{PSC}} \langle m, iP', S'\rangle} \qquad \textsc{read}\quad \frac{\begin{array}{c} S' = S \uplus \{s\} \\ l = \mathtt{R}(x,v) \\ \mathsf{get}(m, \Lambda(iP(x)))(x) = v \end{array}}{\langle m, iP, S\rangle \xrightarrow{\tau, l\#s}_{i\mathsf{PSC}} \langle m, iP, S'\rangle}$$

$$\textsc{rmw}\quad \frac{\begin{array}{c} S' = S \uplus \{s\} \\ l = \mathtt{RMW}(x, v_{\mathsf{R}}, v_{\mathsf{W}}) \\ \mathsf{get}(m, \Lambda(iP(x)))(x) = v_{\mathsf{R}} \\ \forall y.\ \mathtt{FO}(\tau)\#\_ \notin iP(y) \\ iP' = iP[x \mapsto iP(x) \cdot \mathtt{W}(v_{\mathsf{W}})\#s] \end{array}}{\langle m, iP, S\rangle \xrightarrow{\tau, l\#s}_{i\mathsf{PSC}} \langle m, iP', S'\rangle} \qquad \textsc{rmw-fail}\quad \frac{\begin{array}{c} S' = S \uplus \{s\} \\ l = \mathtt{R\text{-}ex}(x,v) \\ \mathsf{get}(m, \Lambda(iP(x)))(x) = v \\ \forall y.\ \mathtt{FO}(\tau)\#\_ \notin iP(y) \end{array}}{\langle m, iP, S\rangle \xrightarrow{\tau, l\#s}_{i\mathsf{PSC}} \langle m, iP, S'\rangle} \qquad \textsc{mfence/sfence}\quad \frac{\begin{array}{c} S' = S \uplus \{s\} \\ l \in \{\mathtt{MF}, \mathtt{SF}\} \\ \forall y.\ \mathtt{FO}(\tau)\#\_ \notin iP(y) \end{array}}{\langle m, iP, S\rangle \xrightarrow{\tau, l\#s}_{i\mathsf{PSC}} \langle m, iP, S'\rangle}$$

$$\textsc{flush}\quad \frac{\begin{array}{c} S' = S \uplus \{s\} \\ l = \mathtt{FL}(x) \\ iP(x) = \emptyset \end{array}}{\langle m, iP, S\rangle \xrightarrow{\tau, l\#s}_{i\mathsf{PSC}} \langle m, iP, S'\rangle} \qquad \textsc{flush-opt}\quad \frac{\begin{array}{c} S' = S \uplus \{s\} \\ l = \mathtt{FO}(x) \\ iP' = iP[x \mapsto iP(x) \cdot \mathtt{FO}(\tau)\#s] \end{array}}{\langle m, iP, S\rangle \xrightarrow{\tau, l\#s}_{i\mathsf{PSC}} \langle m, iP', S'\rangle}$$

$$\textsc{persist-w}\quad \frac{\begin{array}{c} L = \mathtt{PerW}(x)\#s \\ iP(x) = \mathtt{W}(v)\#s \cdot ip \\ iP' = iP[x \mapsto ip] \qquad m' = m[x \mapsto v] \end{array}}{\langle m, iP, S\rangle \xrightarrow{L}_{i\mathsf{PSC}} \langle m', iP', S\rangle} \qquad \textsc{persist-fo}\quad \frac{\begin{array}{c} L = \mathtt{PerFO}(x)\#s \\ iP(x) = \mathtt{FO}(\_)\#s \cdot ip \\ iP' = iP[x \mapsto ip] \end{array}}{\langle m, iP, S\rangle \xrightarrow{L}_{i\mathsf{PSC}} \langle m, iP', S\rangle}$$

Fig. 9. The $i$PSC Instrumented Persistent Memory Subsystem (the instrumentation is colored).

Initially, we clearly have $\langle\langle m_0, \tilde{m}_{\mathsf{Init}}, L_{\mathsf{Init}}, T_{\mathsf{Init}}\rangle, \langle m_0, P_\epsilon\rangle\rangle \in R$. Now, suppose that $\langle m_f, \tilde{m}, L, T\rangle \xrightarrow{\tau, l}_{\mathsf{PSC}_{\mathtt{fin}}} \langle m'_f, \tilde{m}', L', T'\rangle$, and let $\langle m, P\rangle \in \mathsf{PSC.Q}$ such that $\langle\langle m_f, \tilde{m}, L, T\rangle, \langle m, P\rangle\rangle \in R$. Then, we have $m_f = m$. We show that $\langle m, P\rangle \xRightarrow{\tau, l}_l \langle m'_f, P'\rangle$ for some $P'$ such that $\langle\langle m'_f, \tilde{m}', L', T'\rangle, \langle m'_f, P'\rangle\rangle \in R$. The rest of the proof continues by separately considering each possible step of $\mathsf{PSC}_{\mathtt{fin}}$, and establishing the simulation invariants at each step. Below, we present the mapping of $\mathsf{PSC}_{\mathtt{fin}}$-steps to PSC-steps:

- write-persist-step is mapped to a write-step immediately followed by a persist-w-step.
- write-no-persist is mapped to a write-step.
- rmw-persist is mapped to an rmw-step immediately followed by a persist-w-step.
- rmw-no-persist is mapped to an rmw-step.
- flush-opt-persist is mapped to an flush-opt-step immediately followed by a persist-fo-step.
- flush-opt-no-persist is mapped to an flush-opt-step.

- All other steps (READ, RMW-FAIL, MFENCE ,SFENCE, and FLUSH) are mapped to the PSC-step of the same name.

It is straightforward to verify that this mapping induces possible sequences of steps, and preserves the simulation invariants. □

For the converse, we use the following additional proposition (see Def. B.10 for the definition of "commutes").

Proposition E.2. $\langle \alpha, \beta \rangle$ *i*PSC*-commutes if* $\mathtt{typ}(\beta) \in \{\mathtt{PerW}, \mathtt{PerFO}\}$ *and one of the following conditions holds:*

- $\mathtt{typ}(\alpha) \notin \{\mathtt{PerW}, \mathtt{PerFO}\}$ *and* $\#(\alpha) \neq \#(\beta)$.
- $\mathtt{typ}(\alpha) \in \{\mathtt{PerW}, \mathtt{PerFO}\}$ *and* $\mathtt{loc}(\alpha) \neq \mathtt{loc}(\beta)$.

Lemma E.3. *Every $m_0$-to-$m$* PSC*-observable-trace $t$ is also an $m_0$-to-$m$* $\mathrm{PSC}_{\mathtt{fin}}$*-observable-trace.*

Proof (outline). Let $t$ be an $m_0$-to-$m$ PSC-observable-trace. By Lemmas D.1 and B.3, there exists an $m_0$-to-$m$ *i*PSC-trace *it* such that $\Lambda(it) = t$. Using Prop. E.2, we can move all PerW-steps and PerFO-steps to immediately follow their corresponding W/RMW-step and FO-step, thus obtaining a "synchronized" instrumented trace in which every write/rmw/flush-optimal either persists immediately after it is issued or never persists. This instrumented trace easily induces an $m_0$-to-$m$ $\mathrm{PSC}_{\mathtt{fin}}$-observable-trace: we take a *-PERSIST-step for steps that are followed by a PerW-steps or PerFO-steps, and otherwise we take the *-NO-PERSIST or other steps of $\mathrm{PSC}_{\mathtt{fin}}$. □

Theorem 6.2. PSC *and* $\mathrm{PSC}_{\mathtt{fin}}$ *are observationally equivalent.*

Proof. Follows from Lemmas 2.10, E.1 and E.3. □

## F PROOFS FOR SECTION 6.2

The following lemma is used to show that DPSC observationally refines PSC.

Lemma F.1. *Let $G$ be a* DPSC*-consistent initialized execution graph. Then, some $t \in \mathsf{traces}(G)$ is an $m_{\mathsf{Init}}(G)$-to-$m(G)$* PSC*-observable-trace.*

Proof (outline). By Lemmas D.1 and B.3, we may use the instrumented system *i*PSC and show that some *it* with $\Lambda(it) \in \mathsf{traces}(G)$ is an $m_{\mathsf{Init}}(G)$-to-$m(G)$ *i*PSC-trace.

Let *mo* be a modification order for $G$ that satisfies the condition of Def. 6.4. Let $F$ be some injective function from events to $\mathbb{N}$ (we will use it to assign identifiers to the different operations). For every event $e \in \mathsf{E}$, we associate two transition labels $\alpha(e), \gamma(e)$:

- Issue of $e$: $\alpha(e) = \langle \mathtt{tid}(e), \mathtt{lab}(e)\#F(e) \rangle$.
- Propagation of $e$ from persistence buffer to persistent memory (only defined for $e \in \mathsf{W} \cup \mathsf{RMW} \cup \mathsf{FO}$): $\gamma(e) = \begin{cases} \mathtt{PerW}(\mathtt{loc}(e))\#F(e) & e \in \mathsf{W} \cup \mathsf{RMW} \\ \mathtt{PerFO}(\mathtt{loc}(e))\#F(e) & e \in \mathsf{FO} \end{cases}$

Let $T$ be any total order on $G.\mathsf{E}$ extending $G.\mathsf{hb}_{\mathsf{PSC}}(mo)$. We construct a set $A$ of transition labels of *i*PSC and an enumeration of $A$ which will serve as *it*.

Let:

- $E_\alpha = G.\mathsf{E} \setminus \mathsf{Init}$.
- $E_\gamma^{\mathsf{W}_x} = \{w \in (\mathsf{W}_x \setminus \mathsf{Init}) \cup \mathsf{RMW}_x \mid \langle w, G.\mathsf{M}(x) \rangle \in mo^? \}$.
- $E_\gamma^{\mathsf{W}} = \bigcup_{x \in \mathsf{Loc}} E_\gamma^{\mathsf{W}_x}$.
- $E_\gamma^{\mathsf{FO}_x} = \mathsf{FO}_x \cap (dom(T^? ; [\mathsf{FO}_x] ; G.\mathsf{po} ; [\mathsf{RMW} \cup \mathsf{R\text{-}ex} \cup \mathsf{MF} \cup \mathsf{SF}]) \cup dom(T ; [\mathsf{FL}_x \cup E_\gamma^{\mathsf{W}_x}]))$.
- $E_\gamma^{\mathsf{FO}} = \bigcup_{x \in \mathsf{Loc}} E_\gamma^{\mathsf{FO}_x}$.

- $E_\gamma = E^{\mathsf{W}}_\gamma \cup E^{\mathsf{FO}}_\gamma$.

We define

$$A = \{\alpha(e) \mid e \in E_\alpha\} \cup \{\gamma(e) \mid e \in E^{\mathsf{W}}_\gamma \cup E^{\mathsf{FO}}_\gamma\}.$$

Let $R$ be the union of the following relations on $A$:

- $R_1 = \{\langle \alpha(e), \gamma(e)\rangle \mid e \in E_\gamma\}$
- $R_2 = \{\langle \alpha(e_1), \alpha(e_2)\rangle \mid \langle e_1, e_2\rangle \in [E_\alpha] \mathbin{;} T\}$
- $R_3 = \{\langle \gamma(e_1), \alpha(e_2)\rangle \mid \langle e_1, e_2\rangle \in [E_\gamma] \mathbin{;} T \mathbin{;} [\mathsf{FL}]\}$
- $R_4 = \{\langle \gamma(e_1), \alpha(e_2)\rangle \mid \langle e_1, e_2\rangle \in [E^{\mathsf{FO}}_\gamma] \mathbin{;} G.\mathsf{po} \mathbin{;} [\mathsf{RMW} \cup \mathsf{R\text{-}ex} \cup \mathsf{MF} \cup \mathsf{SF}]\}$
- $R_5 = \{\langle \gamma(e_1), \gamma(e_2)\rangle \mid \langle e_1, e_2\rangle \in [E_\gamma] \mathbin{;} T \mathbin{;} [E_\gamma]\}$

It is easy to see that $R$ is acyclic (an $R$-cycle would entail a $T$-cycle). It is standard to verify that for any enumeration $it$ of $R$, we have $\Lambda(it) \in \mathsf{traces}(G)$ and that $it$ is an $m_{\mathsf{Init}}(G)$-to-$m(G)$ $i$PSC-trace. In particular, let $x \in \mathsf{Loc}$ and suppose that for the last transition label of the form $\mathtt{PerW}(x)\#\_$ in $it$ is not $\mathtt{PerW}(x)\#F(G.\mathsf{M}(x))$, but rather $\mathtt{PerW}(x)\#F(w)$ for some $w \in E^{\mathsf{W}}_\gamma \setminus \{G.\mathsf{M}(x)\}$. Then, since $w \in E^{\mathsf{W}}_\gamma$ we have $\langle w, G.\mathsf{M}(x)\rangle \in mo^? \subseteq T^?$, which contradicts the fact that $R_5 \subseteq R$. □

Theorem 6.5. PSC *and* DPSC *are observationally equivalent.*

Proof (outline). The fact that DPSC observationally refines PSC immediately follows from Lemmas 5.15 and F.1. Next, we first show that PSC observationally refines DPSC. Let $t$ be an $m_0$-to-$m$ PSC-observable-trace. We construct a DPSC-consistent $m_0$-initialized execution graph $G$ such that $t \in \mathsf{traces}(G)$ and $m(G) = m$. Then, the claim follows using Lemma 5.14.

We use the instrumented semantics ($i$PSC). By Lemmas D.1 and B.3, there exists a $m_0$-to-$m$ $i$PSC-trace $it$ such that $\Lambda(it) = t$. We use $it$ to construct $G$:

- Events: For every $1 \le i \le |it|$ with $it(i)$ of the form $\langle \tau, l\#s\rangle$, we include the event $e_i \triangleq \langle \tau, i, l\rangle$ in $G.\mathsf{E}$. In addition, we include the initialization events $e_x \triangleq \langle \bot, 0, \mathtt{W}(x, m_0(x))\rangle$ for every $x \in \mathsf{Loc}$. It is easy to see that we have $t \in \mathsf{traces}(G)$ and that $G$ is $m_0$-initialized.
- Reads-from: $G.\mathtt{rf}$ is constructed as follows: for every $1 \le i \le |it|$ with $\mathtt{typ}(e_i) \in \{\mathtt{R}, \mathtt{RMW}, \mathtt{R\text{-}ex}\}$ and $\mathtt{loc}(e_i) = x$, we locate the maximal index $1 \le j < i$ such that $\mathtt{typ}(e_j) \in \{\mathtt{W}, \mathtt{RMW}\}$ and $\mathtt{loc}(e_j) = x$ (namely, the write that corresponds to $e_j$ was the last write executed before the read that corresponds to $e_i$ was executed), and include an edge $\langle e_j, e_i\rangle$ in $G.\mathtt{rf}$. If such index $j$ does not exist, we include the edge $\langle e_x, e_i\rangle$ in $G.\mathtt{rf}$ (reading from the initialization event). Using $i$PSC's operational semantics, it is easy to verify that $G.\mathtt{rf}$ is indeed a reads-from relation for $G.\mathsf{E}$.
- Memory assignment: To define $G.\mathsf{M}$, for every $x \in \mathsf{Loc}$, let $i(x)$ be the maximal index such that $\mathtt{typ}(it(i(x))) = \mathtt{PerW}$ and $\mathtt{loc}(it(i(x))) = x$ (that is, $i(x)$ is the index of the last propagation to the persistent memory of a write to $x$). In addition, let $w(i(x))$ be the (unique) index $k$ such that $\mathtt{typ}(it(k)) \in \{\mathtt{W}, \mathtt{RMW}\}$ and $\#(it(k)) = \#(it(i(x)))$ (that is, $w(i(x))$ is the index of the write operation that persists in index $i(x)$). Now, we define $G.\mathsf{M}(x) \triangleq e_{w(i(x))}$ for every $x \in \mathsf{Loc}$ for which $i(x)$ is defined. If $i(x)$ is undefined ($\mathtt{typ}(it(i) = \mathtt{PerW}$ and $\mathtt{loc}(it(i)) = x$ never hold), we set $G.\mathsf{M}(x) \triangleq e_x$ (the initialization event of $x$). Then, we clearly have $m(G) = m$.

To show that $G$ is DPSC-consistent, we construct a modification $mo$ for $G$. For every two events $e_i, e_j \in G.\mathsf{E} \cap (\mathsf{W} \cup \mathsf{RMW})$ with $\mathtt{loc}(e_i) = \mathtt{loc}(e_j)$, we include $\langle e_i, e_j\rangle$ in $mo$ if either $e_i \in \mathsf{Init}$ or $i < j$ (that is, the write the corresponds to $e_i$ was executed before the write that corresponds to $e_j$). It is to verify that $\langle e_i, e_j\rangle \in G.\mathsf{po} \cup G.\mathtt{rf} \cup mo \cup G.\mathtt{fr}(mo) \cup G.\mathtt{dtpo}(mo)$ implies that $e_i \in \mathsf{Init}$ or $i < j$. It follows that $G.\mathtt{hb}_{\mathsf{PSC}}(mo)$ is acyclic and so $G$ is DPSC-consistent. □

## G PROOFS FOR SECTION 7

Lemma 7.11. *Let $G$ be a* $\mathsf{DPTSO_{syn}}$*-consistent execution graph. Suppose that for every* $w \in G.\mathsf{W} \cup G.\mathsf{RMW}$ *and $G$-unprotected event* $e \in \mathsf{R}_{\mathtt{loc}(w)} \cup \mathsf{FO}_{\mathtt{loc}(w)}$*, we have either* $\langle w, e\rangle \in (G.\mathsf{po} \cup G.\mathtt{rf})^+$ *or* $\langle e, w\rangle \in (G.\mathsf{po} \cup G.\mathtt{rf})^+$*. Then, $G$ is* $\mathsf{DPSC}$*-consistent.*

Proof. By Thm. 5.28, there exists a modification order $mo$ for $G$ such that $G.\mathsf{hb}(mo)$ and $G.\mathtt{fr}(mo)\,;G.\mathsf{po}$ are irreflexive. We show that $G.\mathsf{hb}_{\mathsf{PSC}}(mo)$ is irreflexive. Suppose otherwise. Let $po = G.\mathsf{po}$, $rf = G.\mathtt{rf}$, $fr = G.\mathtt{fr}(mo)$, $dtpo = G.\mathtt{dtpo}(mo)$, $ppo = G.\mathsf{ppo}$, and $hb = G.\mathsf{hb}(mo)$.

Since $po$ is transitive, $(rf \cup mo \cup fr \cup dtpo)\,;dtpo = \emptyset$ (because of the domains and codomains of the different relations), $rf\,;fr \subseteq mo$, $mo\,;fr \subseteq mo$, $fr\,;fr \subseteq fr$, $dtpo\,;fr \subseteq dtpo$ (all these easily follow from the fact that $hb$ is irreflexive), and $dom(rf \cup mo) \subseteq \mathsf{W} \cup \mathsf{RMW}$, it suffices to show that $po\,;[\mathsf{W} \cup \mathsf{RMW}] \cup rf \cup mo \cup po\,;fr \cup po\,;dtpo$ is acyclic.

For this matter, we show that

$$[(\mathsf{R} \cup \mathsf{W} \cup \mathsf{RMW} \cup \mathsf{R\text{-}ex}) \setminus \mathsf{Init}];(po;fr \cup po;dtpo) \setminus (po;[\mathsf{W} \cup \mathsf{RMW}] \cup rf \cup mo)^+ \subseteq ppo^+;fr \cup ppo^+;dtpo.$$

Given the latter inclusion, since $po\,;[\mathsf{W} \cup \mathsf{RMW}] \subseteq ppo$, the acyclicity of $po\,;[\mathsf{W} \cup \mathsf{RMW}] \cup rf \cup mo \cup po\,;fr \cup po\,;dtpo$ will follow from the fact that $hb$ is irreflexive.

Let $\langle a, c\rangle \in [(\mathsf{R} \cup \mathsf{W} \cup \mathsf{RMW} \cup \mathsf{R\text{-}ex}) \setminus \mathsf{Init}]\,;(po\,;fr \cup po\,;dtpo) \setminus (po\,;[\mathsf{W} \cup \mathsf{RMW}] \cup rf \cup mo)^+$. Let $b \in \mathsf{E}$ such that $\langle a, b\rangle \in po$ and $\langle b, c\rangle \in fr \cup dtpo$. Let $x = \mathtt{loc}(b)$. Consider the possible cases:

- $a \in \mathsf{W}$, $\mathtt{loc}(a) \neq x$, $b \in \mathsf{R}$, and $b$ is $G$-protected: Then, we obtain that $\langle a, b\rangle \in po\,;[\mathsf{W}_x \cup \mathsf{RMW} \cup \mathsf{R\text{-}ex} \cup \mathsf{MF}]\,;po$. If $\langle a, b\rangle \in po\,;[\mathsf{RMW} \cup \mathsf{R\text{-}ex} \cup \mathsf{MF}]\,;po$, then we have $\langle a, b\rangle \in ppo^+$. Otherwise, there is some $b' \in \mathsf{W}_x$ such that $\langle a, b'\rangle \in po$ and $\langle b', b\rangle \in po$. In this case it follows that $\langle b', c\rangle \in mo$, which contradicts the assumption that $\langle a, c\rangle \notin (po;[\mathsf{W} \cup \mathsf{RMW}] \cup rf \cup mo)^+$.
- $a \in \mathsf{W}$, $\mathtt{loc}(a) \neq x$, $b \in \mathsf{R}$, and $b$ is not $G$-protected: Then, we must have either $\langle c, b\rangle \in (po \cup rf)^+$ or $\langle b, c\rangle \in (po \cup rf)^+$. In the first case we obtain that $\langle b, b\rangle \in fr\,;(po \cup rf)^+$, which contradicts the fact that $hb$ and $fr\,;po$ are irreflexive. In turn, the second case contradicts the assumption that $\langle a, c\rangle \notin (po\,;[\mathsf{W} \cup \mathsf{RMW}] \cup rf \cup mo)^+$.
- $a \in \mathsf{W}$, $\mathtt{loc}(a) = x$, and $b \in \mathsf{R}$: In this case, we must have $\langle a, b\rangle \in mo^?\,;rf$ and so $\langle a, c\rangle \in mo$, which contradicts the assumption that $\langle a, c\rangle \notin (po\,;[\mathsf{W} \cup \mathsf{RMW}] \cup rf \cup mo)^+$.
- $a \in \mathsf{W}$, $\mathtt{loc}(a) \neq x$, and $b \in \mathsf{FO}$: Then, if $b$ is $G$-protected, we obtain that $\langle a, b\rangle \in po\,;[\mathsf{W}_x \cup \mathsf{RMW} \cup \mathsf{R\text{-}ex} \cup \mathsf{MF} \cup \mathsf{SF}]\,;po \subseteq ppo^+$. Otherwise, we must have either $\langle c, b\rangle \in (po \cup rf)^+$ or $\langle b, c\rangle \in (po \cup rf)^+$. In the first case we obtain that $\langle b, b\rangle \in dtpo\,;(po \cup rf)^+$, which contradicts the fact that $hb$ is irreflexive. In turn, the second case contradicts the assumption that $\langle a, c\rangle \notin (po\,;[\mathsf{W} \cup \mathsf{RMW}] \cup rf \cup mo)^+$.
- Otherwise, the fact that $\langle a, b\rangle \in po$ directly implies that $\langle a, b\rangle \in ppo$. □

Theorem 7.8. *For a program $Pr$ that is not strongly racy, a program state $\overline{q} \in Pr.\mathsf{Q}$ is reachable under* $\mathsf{PTSO_{syn}}$ *iff it is reachable under* $\mathsf{PSC}$*.*

Proof. The right-to-left direction is trivial. For the left-to-right direction, suppose that $\overline{q} \in Pr.\mathsf{Q}$ is reachable under $\mathsf{PTSO_{syn}}$. By Theorems 5.28 and C.2, $\overline{q}$ is reachable under $\mathsf{DPTSO^{mo}_{syn}}$. Let $G_0, \dots, G_n$ be $\mathsf{DPTSO^{mo}_{syn}}$-consistent execution graphs that satisfy the conditions of Def. 5.13 (for the program $Pr$ and the state $\overline{q}$). If all $G_i$'s are DPSC-consistent, then $\overline{q}$ is reachable under DPSC, and the claim follows using Thm. 6.5.

Suppose otherwise. We show that $Pr$ is strongly racy, which contradicts our assumption. Let $0 \leq i \leq n-1$ be the minimal index such that $G_i$ is not DPSC-consistent. Let $G = G_i$. The minimality of $i$ ensures that $G_0, \dots, G_{i-1}$ are all DPSC-consistent as well. Hence, using the sequence $G_0, \dots, G_{i-1}$, by repeatedly applying Lemma F.1 and Prop. 5.11, we obtain that for $m_0 \triangleq m(G_{i-1})$ or $m_0 \triangleq m_{\mathsf{Init}}$ if $i = 0$, we have that $\langle \overline{q}_{\mathsf{Init}}, m_0, P_\epsilon\rangle$ is reachable in $Pr \parallel \mathsf{PSC}$ for some $\overline{q}_{\mathsf{Init}} \in Pr.\mathsf{Q}_{\mathsf{Init}}$.

Let $G.\mathsf{hb} = (G.\mathsf{po} \cup G.\mathsf{rf})^+$ and let

$$W = \left\{ w \in \mathsf{W} \cup \mathsf{RMW} \;\middle|\; \exists e.\ \begin{array}{l} e \text{ is } G\text{-unprotected } \land e \in \mathsf{R}_{\mathtt{loc}(w)} \cup \mathsf{FO}_{\mathtt{loc}(w)} \land \\ \langle w, e\rangle \notin (G.\mathsf{po} \cup G.\mathsf{rf})^+ \;\land\; \langle e, w\rangle \notin (G.\mathsf{po} \cup G.\mathsf{rf})^+ \end{array} \right\}.$$

By Lemma 7.11, $W$ is not empty. Let $w$ be a $G.\mathsf{po} \cup G.\mathsf{rf}$-minimal event in $W$, and let $e$ be a $G.\mathsf{po} \cup G.\mathsf{rf}$-minimal $G$-unprotected event in $\mathsf{R}_{\mathtt{loc}(w)} \cup \mathsf{FO}_{\mathtt{loc}(w)}$ such that $\langle w, e\rangle \notin (G.\mathsf{po} \cup G.\mathsf{rf})^+$ and $\langle e, w\rangle \notin (G.\mathsf{po} \cup G.\mathsf{rf})^+$.

Let $E' = \{e' \mid \langle e', w\rangle \in (G.\mathsf{po} \cup G.\mathsf{rf})^+ \lor \langle e', e\rangle \in (G.\mathsf{po} \cup G.\mathsf{rf})^+\}$ and $G'$ be the execution graph given by $G'.\mathsf{E} = E'$, $G'.\mathsf{rf} = [G'.\mathsf{E}];G.\mathsf{rf};[G'.\mathsf{E}]$, and $G'.\mathsf{M} = \lambda x.\ \max_{mo} G'.\mathsf{E} \cap (\mathsf{W}_x \cup \mathsf{RMW}_x)$, where $mo$ is some modification order for $G$ that satisfies the conditions of Def. 6.4. It is easy to see that $G'$ is $\mathsf{DPTSO}_{\mathsf{syn}}$-consistent (since $G$ is $\mathsf{DPTSO}_{\mathsf{syn}}$-consistent). The minimality of $w$ and $e$ ensures that for every $w' \in G'.\mathsf{W} \cup G'.\mathsf{RMW}$ and $G'$-unprotected event $e' \in \mathsf{R}_{\mathtt{loc}(w)} \cup \mathsf{FO}_{\mathtt{loc}(w)}$, we have either have $\langle w', e'\rangle \in (G'.\mathsf{po} \cup G'.\mathsf{rf})^+$ or $\langle e', w'\rangle \in (G'.\mathsf{po} \cup G'.\mathsf{rf})^+$. Hence, by Lemma 7.11, $G'$ is DPSC-consistent.

Now, since $G$ is generated by $Pr$, we clearly also have that $G'$ is generated by $Pr$ with some final state $\overline{q}'$. Hence, by Prop. 5.11, for every $t \in \mathsf{traces}(G')$, we have $\overline{q}_{\mathsf{Init}} \overset{t}{\Rightarrow}_{Pr} \overline{q}'$ for some $\overline{q}_{\mathsf{Init}} \in Pr.\mathsf{Q}_{\mathsf{Init}}$. By Lemma F.1, some $t \in \mathsf{traces}(G')$ is an $m_0$-to-$m(G')$ PSC-observable-trace. It follows that $\langle \overline{q}_{\mathsf{Init}}, m_0, P_\epsilon\rangle \overset{t}{\Rightarrow}_{Pr \| \mathsf{PSC}} \langle \overline{q}', m(G'), P\rangle$ for some $P$.

Furthermore, the construction of $G'$ ensures that for $\tau_{\mathsf{W}} = \mathtt{tid}(w)$ and $\tau = \mathtt{tid}(e)$, we have that $\overline{q}'(\tau_{\mathsf{W}})$ enables $\mathtt{lab}(w)$ and $\overline{q}'(\tau_{\mathsf{R}})$ enables $\mathtt{lab}(e)$. To show that $Pr$ is strongly racy, it remains to show that $\mathtt{lab}(e)$ is unprotected in $\mathsf{suffix}_{\mathtt{tid}}(e)(t)$. Let $G'_e$ be the execution graph given by $G'_e.\mathsf{E} = G'.\mathsf{E} \cup \{e\}$, $G'_e.\mathsf{rf} = [G'_e.\mathsf{E}]\ ;G.\mathsf{rf}\ ;[G'_e.\mathsf{E}]$, and $G'_e.\mathsf{M} = \lambda x.\ \max_{mo} G'_e.\mathsf{E} \cap (\mathsf{W}_x \cup \mathsf{RMW}_x)$. Using Prop. 7.10, it suffices to show that $e$ is $G'_e$-unprotected. The latter easily follows from the fact that $e$ is $G$-unprotected. □

$$\begin{array}{lll}
e ::= & & r \mid v \mid e+e \mid e=e \mid e\neq e \mid \ldots \\
\mathsf{Inst} \ni \mathit{inst} ::= & & r := e \mid \texttt{if}\ e\ \texttt{goto}\ n \mid x := e \mid r := x \mid \\
& & r := \texttt{FADD}(x,e) \mid r := \texttt{CAS}(x,e,e) \mid \\
& & \texttt{mfence} \mid \texttt{fl}(x) \mid \texttt{fo}(x) \mid \texttt{sfence}
\end{array}$$

Fig. 10. Programming language syntax.

$$\frac{\begin{array}{c} S(pc) = r := e \\ \phi' = \phi[r \mapsto \phi(e)] \end{array}}{\langle pc, \phi\rangle \xrightarrow{\epsilon}_S \langle pc+1, \phi'\rangle}
\qquad
\frac{\begin{array}{c} S(pc) = \texttt{if}\ e\ \texttt{goto}\ n \\ \phi(e) \neq 0 \end{array}}{\langle pc, \phi\rangle \xrightarrow{\epsilon}_S \langle n, \phi\rangle}
\qquad
\frac{\begin{array}{c} S(pc) = \texttt{if}\ e\ \texttt{goto}\ n \\ \phi(e) = 0 \end{array}}{\langle pc, \phi\rangle \xrightarrow{\epsilon}_S \langle pc+1, \phi\rangle}$$

$$\frac{\begin{array}{c} S(pc) = x := e \\ l = \mathtt{W}(x, \phi(e)) \end{array}}{\langle pc, \phi\rangle \xrightarrow{l}_S \langle pc+1, \phi\rangle}
\qquad
\frac{\begin{array}{c} S(pc) = r := x \\ l = \mathsf{R}(x, v) \quad \phi' = \phi[r \mapsto v] \end{array}}{\langle pc, \phi\rangle \xrightarrow{l}_S \langle pc+1, \phi'\rangle}
\qquad
\frac{\begin{array}{c} S(pc) = r := \texttt{FADD}(x, e) \\ l = \mathsf{RMW}(x, v, v + \phi(e)) \\ \phi' = \phi[r \mapsto v] \end{array}}{\langle pc, \phi\rangle \xrightarrow{l}_S \langle pc+1, \phi'\rangle}$$

$$\frac{\begin{array}{c} S(pc) = r := \texttt{CAS}(x, e_{\mathsf{R}}, e_{\mathsf{W}}) \\ l = \mathsf{RMW}(x, \phi(e_{\mathsf{R}}), \phi(e_{\mathsf{W}})) \\ \phi' = \phi[r \mapsto \phi(e_{\mathsf{R}})] \end{array}}{\langle pc, \phi\rangle \xrightarrow{l}_S \langle pc+1, \phi'\rangle}
\qquad
\frac{\begin{array}{c} S(pc) = r := \texttt{CAS}(x, e_{\mathsf{R}}, e_{\mathsf{W}}) \\ l = \mathsf{R\text{-}ex}(x, v) \qquad v \neq \phi(e_{\mathsf{R}}) \\ \phi' = \phi[r \mapsto v] \end{array}}{\langle pc, \phi\rangle \xrightarrow{l}_S \langle pc+1, \phi'\rangle}
\qquad
\frac{\begin{array}{c} S(pc) = \texttt{mfence} \\ l = \mathsf{MF} \end{array}}{\langle pc, \phi\rangle \xrightarrow{l}_S \langle pc+1, \phi\rangle}$$

$$\frac{\begin{array}{c} S(pc) = \texttt{fl}(x) \\ l = \mathsf{FL}(x) \end{array}}{\langle pc, \phi\rangle \xrightarrow{l}_S \langle pc+1, \phi\rangle}
\qquad
\frac{\begin{array}{c} S(pc) = \texttt{fo}(x) \\ l = \mathsf{FO}(x) \end{array}}{\langle pc, \phi\rangle \xrightarrow{l}_S \langle pc+1, \phi\rangle}
\qquad
\frac{\begin{array}{c} S(pc) = \texttt{sfence} \\ l = \mathsf{SF} \end{array}}{\langle pc, \phi\rangle \xrightarrow{l}_S \langle pc+1, \phi\rangle}$$

Fig. 11. Transitions of LTS induced by a sequential program $S \in \mathsf{SProg}$.

## H FROM PROGRAMS TO LABELED TRANSITION SYSTEMS

We present a concrete programming language syntax for (sequential) programs, and show how programs in this language are interpreted as LTSs in the form assumed assumed in §2.1.

Let $\mathsf{Reg} \subseteq \{\mathsf{a}, \mathsf{b}, \ldots\}$ be a finite set of register names. Figure 10 presents our toy language. Its expressions are constructed from registers (local variables) and values. Instructions include assignments and conditional branching, as well as memory operations.

A sequential program $S$ is a function from a set of the form $\{0, 1, \ldots, N\}$ (the possible values of the program counter) to instructions. It induces an LTS over $\mathsf{Lab} \cup \{\epsilon\}$. Its states are pairs $q = \langle pc, \phi\rangle$ where $pc \in \mathbb{N}$ (called *program counter*) and $\phi : \mathsf{Reg} \to \mathsf{Val}$ (called *local store*, and extended to expressions in the obvious way). Its initial state is $\langle 0, \lambda r \in \mathsf{Reg}.\, 0\rangle$ and its transitions are given in Fig. 11 (In particular, a read instruction in $S$ induces $|\mathsf{Val}|$ transitions with different labels.)