Revamping Hardware Persistency Models

View-Based and Axiomatic Persistency Models for Intel-x86 and Armv8

Kyeongmin Cho
kyeongmin.cho@kaist.ac.kr
KAIST
Daejeon, Korea

Sung-Hwan Lee
sunghwan.lee@sf.snu.ac.kr
Seoul National University
Seoul, Korea

Azalea Raad
azalea@imperial.ac.uk
Imperial College London
London, United Kingdom

Jeehoon Kang
jeehoon.kang@kaist.ac.kr
KAIST
Daejeon, Korea

Abstract

Non-volatile memory (NVM) is a cutting-edge storage technology that promises the performance of DRAM with the durability of SSD. Recent work has proposed several *persistency models* for mainstream architectures such as Intel-x86 and Armv8, describing the order in which writes are propagated to NVM. However, these models have several limitations; most notably, they either lack operational models or do not support persistent synchronization patterns.

We close this gap by revamping the existing persistency models. First, inspired by the recent work on promising semantics, we propose a *unified operational style* for describing persistency using *views*, and develop view-based operational persistency models for Intel-x86 and Armv8, thus presenting the *first* operational model for Armv8 persistency. Next, we propose a *unified axiomatic style* for describing hardware persistency, allowing us to recast and repair the existing axiomatic models of Intel-x86 and Armv8 persistency. We prove that our axiomatic models are equivalent to the authoritative semantics reviewed by Intel and Arm engineers. We further prove that each axiomatic hardware persistency model is equivalent to its operational counterpart. Finally, we develop a persistent model checking algorithm and tool, and use it to verify several representative examples.

CCS Concepts: • **Theory of computation** → **Concurrency**; **Semantics and reasoning**; *Verification by model checking*.

PLDI '21, June 20–25, 2021, Virtual, Canada

ACM ISBN 978-1-4503-8391-2/21/06.
https://doi.org/10.1145/3453483.3454027

Keywords: persistent memory, non-volatile random-access memory, NVRAM, persistency semantics, x86, Armv8

ACM Reference Format:
Kyeongmin Cho, Sung-Hwan Lee, Azalea Raad, and Jeehoon Kang. 2021. Revamping Hardware Persistency Models: View-Based and Axiomatic Persistency Models for Intel-x86 and Armv8. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3453483.3454027

1 Introduction

Non-volatile memory (NVM) is an emerging class of storage technology that simultaneously provides (1) byte addressability, low latency, and high throughput as DRAM does; and (2) durability (data persistency across system crashes) and high capacity as SSD does. It is widely believed that NVM (a.k.a. persistent memory) will eventually supplant volatile memory [42], allowing efficient access to persistent data. This belief is backed by industrial support. Specifically, the two major architectures, Intel-x86 and Armv8 which together account for almost 100% of the desktop and mobile market, have extended their official specifications to support persistent programming [4, 20]. Intel has further released open-source NVM libraries such as PMDK [19], and manufactured its own line of NVM, Optane DC persistent memory [21], with an extended academic study evaluating its performance [24]. NVM is therefore expected to innovate high performance transactional systems [7, 18, 31, 35, 38, 53] and large-scale memory systems [37, 40, 50].

However, building correct transactional systems over persistent memory is difficult in part due to *relaxed* persistency: writes to NVM locations may not be persisted to memory in the program order due to micro-architectural optimizations such as out-of-order execution, store buffering, or caching protocols. For instance, consider the programs below:

(*a*) data := 42 (*b*) commit := 1	(CommitWeak)	(*a*) data := 42 (*b*) `flush data` (*c*) commit := 1 (Commit1)

Hereafter we assume all program variables in our examples are locations in NVM[1] initialized to 0; variable reads and writes are architecture-level load and store instructions, e.g., `mov` on Intel-x86 and `ldr`, `str` in Armv8; and that `flush` represents a persistency fence, e.g., `clflush` on Intel-x86 and `dc.cvap`; `dsb.sy` on Armv8.[2]

In both examples we aim to establish the invariant $I \triangleq$ commit=1 $\Rightarrow$ data=42 even in case of an unexpected crash. In the case of CommitWeak without a persistency fence, we fail to establish I over mainstream architectures such as Intel-x86 and Armv8: the two stores may persist to NVM out of order, thereby allowing commit=1, data=0 upon crash recovery. By contrast, in the case of Commit1 the persistency fence at b ensures that the two stores persist in the intended (program) order, thereby establishing the invariant I. Micro-architecturally, `flush` data blocks until the previous store on data at a is persisted to NVM, thus ensuring that the store at c always persists after that of a. As such, persistency fences are expensive and should be used sparingly.

Relaxed persistency is further complicated in multi-threaded settings. Consider the following program with two threads:

```
(a) data := 42   ||  (b) if (data != 0) {
                 ||  (c)     flush data          (Commit2)
                 ||  (d)     commit := 1 }
```

This example differs from Commit1 in that data and commit are written to by different threads. Once again, if the fence at c were removed, the desired invariant I would no longer hold: although the store on data at a may be propagated (made visible) to the the right thread through cache coherence protocols, it may not be persisted to NVM prior to the crash. As before, the fence at c ensures that the store at a (which was propagated to the right thread before c) persists to NVM before the store at d, thus establishing I.

Note that during normal (non-crashing) executions, under both Intel-x86 and Armv8 no thread can observe the undesirable behavior commit=1, data=0 even without the fence at c, underlining the difference between the *consistency order* (the order in which stores are propagated across threads) and the *persistency order* (the order in which stores are persisted to NVM). In general, relaxed concurrency models constrain the consistency order, while relaxed persistency models additionally constrain the persistency order, further compounding the complexity of relaxed concurrency.

In order to facilitate correct persistent programming with efficient use of persistency fences, existing work includes several persistency models [8, 17, 28, 30, 42, 46–48]. However, as we discuss below, these models have several shortcomings.

[1] As in [47], we assume all locations are durable locations in NVM.

[2] Armv8 recently introduced the `dc.cvadp` instruction that, unlike `dc.cvap`, guarantees persistence even in case of battery/hardware failures [4]. We focus on `dc.cvap` in this paper, but most discussions also apply to `dc.cvadp`.

Problem To our knowledge, no existing persistency model (except for PTSO_{syn} [28], discussed shortly below) satisfies all of the following properties simultaneously:

(A) **Describing mainstream architectures or languages**: For a persistency model to be widely used and applied, it should describe the persistency behavior of mainstream hardware/software platforms such as readily available architectures, e.g., Intel-x86 [20] and Armv8 [4], and ubiquitous languages, e.g., C/C++, over which several persistent libraries are implemented [18, 19]. Moreover, the model should be sufficiently relaxed that the behaviors observable on existing platforms are also allowed by the model. Otherwise, invariants that hold according to the model would be invalidated by executions on such platforms, rendering the model unsound for reasoning.

(B) **Supporting persistent synchronization patterns**: A persistency model should support common synchronization patterns used in practical implementations of persistent objects, e.g., transactions or file systems. For instance, a model should prohibit undesirable behaviors, e.g., commit=1, data=0 in Commit1 and Commit2 that capture the essence of practical implementations of transactional systems. In particular, the model should be sufficiently strict that unobservable behaviors on existing platforms are also forbidden by the model. Otherwise, admitting unobservable behaviors in the model makes it impossible to reason about such patterns. Moreover, a model should serve as an objective correctness criteria for new, more efficient designs of persistent objects, and in doing so, guide such new designs.

(C) **Operational**: An *operational* persistency model is desirable in that it enables stepping through an execution for debugging purposes. Moreover, operational models are more suitable for building high-level reasoning techniques such as program logics. By contrast, axiomatic models constrain the admitted behaviors through a set of axioms over full executions, making them undesirable for step-by-step reasoning (e.g., as in program logics).

The models of [8, 17, 30, 42, 46] do not satisfy (A). Specifically, [8, 17, 30, 42] present language-level persistency models put forward as academic *proposals*, and are not supported by mainstream programming languages. Similarly, [46] proposes a hardware persistency model, PTSO, by integrating buffered epoch persistency [42] with the TSO architecture of x86/SPARC [49]. However, PTSO is not supported by mainstream architectures of Intel-x86 and Armv8.

The PArmv8 model [48, "PARMv8"] describes the persistency semantics of the Armv8 architecture, but is not operational (C). Moreover, as we discuss shortly in §2, the PArmv8 model is too weak in that it violates multi-copy atomicity. Similarly, the Px86 model [47] describes the persistency semantics of the Intel-x86 architecture operationally

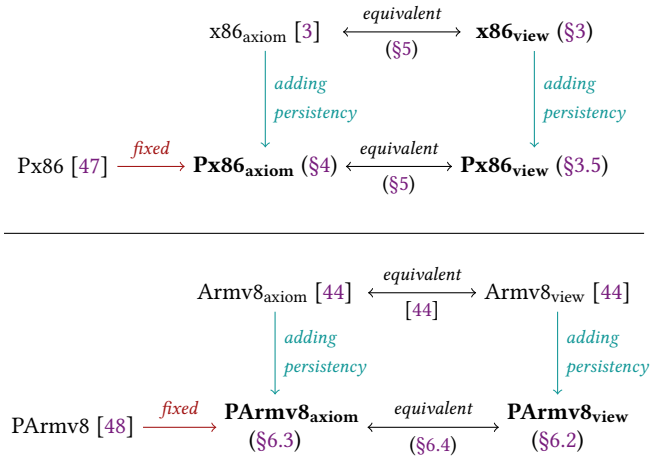

Figure 1. Relationship among Intel-x86 and Armv8 models

and axiomatically.[3] However, as we discuss shortly, Px86 is too relaxed and does not always support persistent synchronization patterns in the presence of I/O (B).

Khyzha and Lahav [28] recently developed the PTSO_{syn} model for Intel-x86 that fixes the Px86 problem regarding I/O and satisfies (A)–(C). However, they do not discuss this problem as they have a different motivation, i.e., presenting a model that better matches the developers' intuition [28, §1]. We discuss PTSO_{syn} in more detail later in §8.

Our Solution, Contributions and Outline We propose a *unified operational style* for describing relaxed persistency using *views*, and develop view-based persistency models of Intel-x86/Armv8 that satisfy all three (A)–(C) properties. In doing so, we develop the *first* operational model for Armv8 persistency. Our operational models highlight 2 flaws in the existing (axiomatic) persistency models of Intel-x86 and Armv8. To remedy this, we develop a *unified axiomatic style* for persistency, adapt the the existing Intel-x86/Armv8 persistency models to our unified style, and repair their flaws.

The remainder of this paper is organized as follows:

- We discuss the shortcomings of the existing persistency models of Intel-x86/Armv8 and present an intuitive account of our solution as view-based models (§2).
- We develop x86_{view}, a new view-based model for Intel-x86 concurrency (§3).
- We develop $\text{Px86}_{\text{view}}$ (§3.5) and $\text{PArmv8}_{\text{view}}$ (§6.2), respectively extending the x86_{view} and $\text{Armv8}_{\text{view}}$ [44] models to account for persistency.
- We present $\text{Px86}_{\text{axiom}}$ (§4) and $\text{PArmv8}_{\text{axiom}}$ (§6.3), our axiomatic models of Intel-x86 and Armv8 persistency that simplify and repair the state-of-the-art models of the respective architectures [47, 48]. We prove that our axiomatic models are equivalent to the authoritative semantics reviewed by Intel and Arm engineers, modulo our proposed fixes (§4.4 and §6.3). Our proposed fix in $\text{PArmv8}_{\text{axiom}}$ has been reviewed by Arm engineers.
- We prove that $\text{Px86}_{\text{view}}$ and $\text{PArmv8}_{\text{view}}$ are equivalent to $\text{Px86}_{\text{axiom}}$ and $\text{PArmv8}_{\text{axiom}}$, respectively. The equivalence proof is mechanized in Coq (§5 and §6.4).
- We develop a stateless model checker for persistency and use it to verify several representative examples under $\text{PArmv8}_{\text{view}}$ (§7). We conclude with related and future work (§8).

We present an overview of the concurrency and persistency models we present in this paper in Fig. 1, summarizing their relationship with existing models in the literature.

2 Overview

We discuss the shortcomings of Px86 and PArmv8 as regards to (B) (§2.1 and 2.2). We then present an intuitive account of our key idea to provide a persistency model that satisfies all three desired properties in (A)–(C) simultaneously (§2.3).

2.1 The Px86 Model and Synchronous Flushes

The Px86 model [47] is too weak in that its instruction for propagating stores to NVM behaves *asynchronously*: executing `clflush` under Px86 does not block execution, and merely guarantees that the pending stores on the given location will be persisted to NVM at some future point. For instance, if the generic `flush data` instruction in Commit1 is replaced with its Intel-x86 analogue, `clflush data`, then once `clflush data` is executed, there is no guarantee under Px86 that the earlier stores on data (including that at *a*) are persisted to NVM; rather (1) these stores will be persisted to NVM at some future point; and (2) they will be persisted to NVM *before all future stores* (including that at *c*). In other words, the persistency ordering guarantees of `clflush` in (2) allows us to establish the desired invariant *I*, even though the effect of `clflush data` may not immediately take place.

The asynchronous behavior of `clflush` is observable in the presence of *external operations* as they narrow down possible crash points through additional observations. For instance, consider the variant of Commit1 below where we replace the store to commit with an analogous I/O operation that writes "commit" to `file` on disk:

```
(a) data := 42          ||  (d) if (flag != 0) {
(b) flush data          ||  (e)   log(file,"commit")}   (CommitE)
(c) flag := 1           ||
```

Let us write *C* to denote that `file` contains "`commit`". Under Px86 it is possible to observe the post-crash state S : data=0$\wedge$ *C*; i.e., when the I/O operation is executed, the asynchronous effect of `clflush` may not have taken place yet.

[3] In [47] the authors introduce two persistency models for Intel-x86: Px86_{man} which formalizes the ambiguous and under-specified behavior described in the Intel reference manual [20], and Px86_{sim} which simplifies and strengthens Px86_{man} to capture the architectural intent envisaged by Intel engineers. In this paper we focus on the Px86_{sim} model and simply refer to it as Px86.

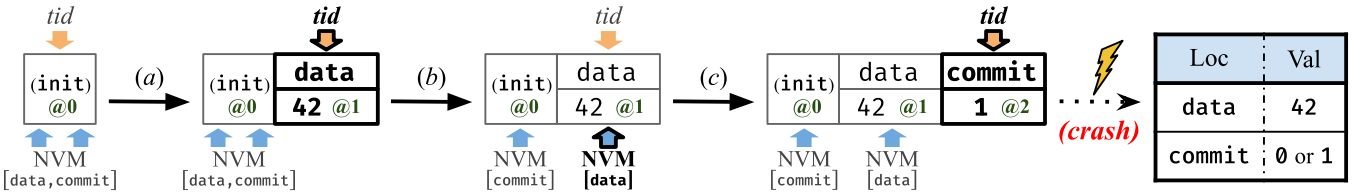

Figure 2. A view-based execution of COMMIT1

As such, to support persistency synchronization patterns such as COMMITE in the presence of external operations under Px86, we must strengthen Px86 by modeling the behavior of `clflush` *synchronously*. Let us write SPx86 for a strengthening of Px86 in which `clflush` instructions are executed synchronously, i.e., they block until all pending stores on the location are persisted to NVM. In the absence of external operations such as I/O or network messages, the asynchronous behavior of `clflush` cannot be observed, i.e., SPx86 is indistinguishable from Px86. By contrast, in the presence of external operations, only SPx86 satisfies an invariant analogous to that of COMMIT1: $C \Rightarrow$ data=42; i.e., once the I/O operation is executed, the synchronous effect of `clflush` must have taken place and S cannot be observed.

2.2 The PArmv8 Model and Multi-Copy Atomicity

The PArmv8 model [48] is too weak in that it violates the principles of *multi-copy atomicity* (MCA) which ensures that a write by one thread is made visible to all other threads simultaneously. Although Armv8 was originally non-MCA, it was recently simplified to observe MCA [43]. However, the persistency extension of Armv8 in PArmv8 violates MCA by allowing the following behavior regarding persistency:

(*a*) y := 1	(*f*) x := 1	
(*b*) dsb.sy	(*g*) dsb.sy	
(*c*) `flushopt x`	(*h*) `flushopt y`	(FLUSHMCA)
(*d*) dsb.sy	(*i*) dsb.sy	
(*e*) z := 1	(*j*) w := 1	

Executing `flushopt x` persists all pending stores on the same cache line as x *asynchronously*.[4] Moreover, if `flushopt` x is followed by a data synchronization barrier, `dsb.sy`, its effects take place *synchronously*; i.e., executing `dsb.sy` awaits the completion of all earlier `flushopt` by the same thread.

We argue that MCA should preclude the post-crash state S : z = w = 1 $\wedge$ x = y = 0. First, to observe z = w = 1 after a crash, the two threads should have fully executed to the end. Second, to observe y = 0 after a crash, (*a*) should not have been made visible to (*h*) prior to the crash, and thus (*h*) must be ordered before (*a*). Third, (*g*) must be ordered before (*h*) and (*a*) before (*b*) because (*g*) and (*b*) are fences. Transitively, (*g*) must be ordered before (*h*), (*a*), and then (*b*). As such, (*f*) should be visible to (*c*), thus ensuring x = 1 after the crash and precluding the behavior in S.[5]

To ensure MCA for persistency, we must thus strengthen PArmv8 by enforcing an order between a flush (e.g., *c*) and a write on the same location that is not persisted by the flush (e.g., *f*). Let us write SPArmv8 for such a strengthening of PArmv8. Under SPArmv8, if x = 0 after a crash, then (*c*) is ordered before (*f*); (*a*) is ordered before (*h*); y = 1 is persisted to the NVM; and thus S cannot be observed.

Upon discussing FLUSHMCA with engineers at Arm, they confirmed that this non-MCA behavior is indeed prohibited and our proposal in §6.3 is the correct interpretation of Arm architecture reference manual [4].

2.3 Our Solution: View-Based Operational Models

We present view-based operational models for the relaxed persistency behavior of Intel-x86/Armv8 architectures that satisfy all three properties in (A)–(C). We build our model over the view-based model of Armv8/RISC-V relaxed-memory *concurrency* [44]. Intuitively, view-based models [27, 34, 44] combine two key ideas: (1) recording the entire *store history* in the memory and allowing threads to read old values; and (2) imposing ordering constraints with per-thread *views* representing the set of stores propagated to each thread and thus constraining the outcomes of future loads and stores by a thread. Here, we further introduce the notion of *persistency views* for each location l, denoting the set of stores on l that have persisted to NVM and thus will survive a crash.

We next illustrate these ideas through a view-based execution of COMMIT1 in Fig. 2, comprising a single thread *tid*. At each execution stage, the store history is recorded in memory as an indexed (timestamped, e.g., @1) list of stores; the view of *tid* records (the timestamp of) the latest store propagated to *tid* (the *tid*-labelled arrow); and the persistency view of each location l records (the timestamp of) the latest store on l that has persisted to NVM (the NVM[l]-labelled arrows).

The initial memory is $M = [\,]$, denoting the empty history (no stores have executed), depicted as `init` at timestamp 0 (@0); the *tid* view is v = @0 (no stores have propagated to *tid*); and the persistency view of each location l is $v_{\text{NVM}}[l]$ = @0 (no stores on l have persisted to NVM). Subsequently:

[4] For the sake of uniformity with our respective Intel-x86 models, we write `flushopt x` in lieu of the Armv8 instruction `dc cvap x`.

[5] The reader may have noted that this behavior is forbidden even if the `dsb.sy` at (*b*) and (*g*) are replaced with the weaker `dmb.sy`. We opt for `dsb.sy` to simplify the example by using only one kind of fence.

(*a*) Executing data := 42 appends its store to memory (M = [⟨data := 42⟩@1]), and advances the *tid* view (ν = @1): the store is executed by and thus propagated to *tid*.
(*b*) Executing `flush` data joins the persistency view of data with the *tid* view (ν_{NVM}[data] = @1), ensuring that the latest data store propagated to *tid* is persisted to NVM.
(*c*) Analogously, executing commit := 1 yields ν = @2 and M = [⟨data := 42⟩@1, ⟨commit := 1⟩@2].

The post-crash outcomes (NVM contents) are then determined by the persistency views. Concretely, after a crash each NVM location l may contain a value written by a store whose timestamp is at least $\nu_{\text{NVM}}[l]$. For instance, if a crash occurs after executing `flush` data, then in the post-crash state ν_{NVM}[data] = @1 and thus data = 42@1; i.e., data := 42 must have persisted to NVM, establishing invariant I.

We next describe an execution of Commit2, where tid_1 and tid_2 denote the left and right threads, respectively. Initially, the memory is M = []; the persistency view is $\nu_{\text{NVM}} = \lambda l.@0$; and the tid_i view is ν_i = @0 (for $i \in \{1, 2\}$). Then:

(*a*) Executing data := 42 yields M=[⟨data := 42⟩@1], ν_1=@1.
(*b*) Thread tid_2 may then load data = 42 as its view timestamp (ν_2 = @0) is less than @1 of data := 42. After loading data = 42, the tid_2 view is joined with @1: ν_2 = @1.
(*c*) Executing `flush` data yields ν_{NVM}[data] = @1.
(*d*) Executing commit := 1 results in M = [⟨data := 42⟩@1, ⟨commit := 1⟩@2] and ν_2 = @2.

As with Commit1, the invariant I holds in case of a crash.

Our models indeed satisfy all desired properties. (A) Our models capture the persistency behavior of the mainstream Armv8 and Intel-x86 architectures. Specifically, we prove that our models are equivalent (modulo fixes) to the axiomatic models of [47, 48] reviewed by Intel/Arm engineers. Our equivalence proof is mechanized in Coq [1] and is publicly available [10]. (B) Our models support persistent synchronization patterns such as those of Commit1 and Commit2. (C) Our models are operational as with the existing family of view-based models [27, 44]. Furthermore, to support reasoning about programs over our models, we develop a stateless model checking algorithm and tool for persistency verification, and use it to verify several representative examples under PArmv8$_{\text{view}}$.[6] Our model checking tool and verified examples are open-source and publicly available [10].

[6] As a proof of concept, we focus on model checking only Armv8 persistency. This is sufficient to showcase the feasibility of model checking for hardware persistency since Armv8 is more complex than Intel-x86 with a bigger search space. We believe it is straightforward to adapt our approach to Intel-x86 persistency, especially given our unified semantic style for persistency.

$$
\begin{aligned}
p &::= s_1 \,||\, \ldots \,||\, s_n \\
s \in \text{St} &::= \texttt{skip} \mid s_1; s_2 \mid \texttt{if}\ (e)\ s_1\ s_2 \mid \texttt{while}\ (e)\ s \mid r := e \\
&\quad \mid r := \texttt{load}\ [e] \mid \texttt{store}\ [e_1]\ e_2 \mid r := \texttt{rmw}\ rop\ [e] \\
&\quad \mid \texttt{fence}_f \mid \texttt{flush}\ e \mid \texttt{flushopt}\ e \\
rop \in \text{Rmw} &::= \texttt{fetch-op}\ op\ e \mid \texttt{cas}\ e_1\ e_2 \\
f \in \text{F} &::= \texttt{sfence} \mid \texttt{mfence} \\
e \in \text{Expr} &::= v \mid r \mid (e_1\ op\ e_2) \qquad op \in \text{O} ::= + \mid - \mid \ldots \\
v \in \text{Val} &= \mathbb{Z} \qquad r \in \text{Reg} = \mathbb{N} \qquad l \in \text{Loc} = \text{Val}
\end{aligned}
$$

Figure 3. Intel-x86 concurrency and persistency language

3 Px86$_{\text{view}}$: A View-Based Model for Intel-x86 Persistency

We develop Px86$_{\text{view}}$, a view-based model for Intel-x86 persistency. We present a simple language for Intel-x86 concurrency and persistency (§3.1) used throughout this section. We develop x86$_{\text{view}}$, a new Intel-x86 concurrency model we use as a baseline (without persistency) and its two key ideas: store histories (§3.2) and views (§3.3); we describe how we support read-modify-writes (§3.4). We then extend x86$_{\text{view}}$ with persistency and develop Px86$_{\text{view}}$ (§3.5).

3.1 Language for Intel-x86 Persistency

To keep our presentation concrete, we use the language in Fig. 3 for Intel-x86 concurrency and persistency. A program p consists of concurrent statements run by distinct threads. A statement s is given by the standard 'while' language over register machines with concurrent memory instructions. The instruction r := `load` [e] reads from the (NVM) location denoted by expression e and returns it in register r. The `store` [e_1] e_2 reads the value denoted by e_2 and stores it at the location denoted by e_1. Analogously, r := `rmw` rop [e] evaluates the expressions rop, performs an RMW ('read-modify-write') operation (e.g., 'compare-and-swap') on the location denoted by e, and returns its old value in r. Finally, `fence`$_f$ issues a memory 'fence' such as `sfence` or `mfence`; and `flush` e and `flushopt` e persist to NVM the pending stores on the cache line containing the location given by e.

3.2 The x86$_{\text{view}}$ Model

We present x86$_{\text{view}}$, our view-based operational model for Intel-x86 concurrency, in Fig. 4. Our design of x86$_{\text{view}}$ is inspired by Armv8$_{\text{view}}$, a view-based concurrency model of Armv8 [44]. As Intel-x86 concurrency is simpler than that of Armv8, we develop x86$_{\text{view}}$ by removing certain Armv8$_{\text{view}}$ features. Here, we highlight the interesting aspects of Intel-x86 and refer the reader to [9, Appendix A] for the full details.

States We represent a machine as a pair $\langle \vec{T}, M \rangle$, comprising a thread map $\vec{T}$ and a memory M. A thread map associates each thread with a statement and a thread state. A thread state

$$\langle \vec{T}, M \rangle \in \text{Machine} \triangleq (\text{TId} \to \text{Thread}) \times \text{Memory}$$
$$tid \in \text{TId} \triangleq \mathbb{N} \quad T \in \text{Thread} \triangleq \text{St} \times \text{TState}$$
$$M \in \text{Memory} \triangleq \text{list Msg} \quad w \in \text{Msg} \triangleq \langle \text{loc}:\text{Loc}; \text{val}:\text{Val}; \text{tid}:\text{TId} \rangle$$
$$\langle l := v \rangle_{tid} \triangleq \langle \text{loc}=l; \text{val}=v; \text{tid}=tid \rangle \quad t \in \mathbb{T} \triangleq \mathbb{N} \quad v \in \mathbb{V} \triangleq \mathbb{T}$$
$$ts \in \text{TState} \triangleq \left\langle \begin{array}{l} \text{regs} : \text{Reg} \to \text{Val}; \\ \text{coh} : \text{Loc} \to \mathbb{V}; \quad \text{v}_{\text{rNew}} : \mathbb{V}; \end{array} \right\rangle$$

(INIT)

$$\frac{p = s_1 \mid\mid \ldots \mid\mid s_n}{\text{init}(p, \langle \lambda tid.\ \langle s_{tid}, \langle \text{regs} = \lambda\_.\ 0; \text{coh} = \lambda\_.\ @0; \text{v}_{\text{rNew}} = @0 \rangle \rangle, [] \rangle)}$$

(MACHINE)

$$\frac{\vec{T}[tid], M \to_{tid} T', M'}{\langle \vec{T}, M \rangle \to \langle \vec{T}[tid \mapsto T'], M' \rangle}$$

(NOT-OVERWRITTEN)

$$\frac{\forall t \in (v_2, v_1].\ M[t].\text{loc} \neq l}{v_1 \sqsubseteq_{M,l} v_2}$$

(STORE)

$$\frac{\begin{array}{l} l = [\![e_1]\!]_{ts.\text{regs}} \quad v = [\![e_2]\!]_{ts.\text{regs}} \quad ts.\text{coh}[l] \sqsubseteq t \quad \sqcup_l ts.\text{coh}[l] \sqsubseteq t \\ t = |M| + 1 \quad M' = M +\!\!+ [\langle l := v \rangle_{tid} @ t] \quad ts' = ts[\text{coh}[l] \mapsto t] \end{array}}{(\texttt{store}\ [e_1]\ e_2, ts), M \to_{tid} (\texttt{skip}, ts'), M'}$$

(LOAD)

$$\frac{\begin{array}{l} l = [\![e]\!]_{ts.\text{regs}} \quad M[t] = \langle l := v \rangle \quad ts.\text{coh}[l] \sqsubseteq t \quad ts.\text{v}_{\text{rNew}} \sqsubseteq_{M,l} t \\ ts' = ts[\text{regs}[r] \mapsto v, \text{coh}[l] \mapsto t, \text{v}_{\text{rNew}} \mapsto_{\sqcup} t \neq ts.\text{coh}[l]\ ?\ t] \end{array}}{(r := \texttt{load}\ [e], ts), M \to_{tid} (\texttt{skip}, ts'), M}$$

(MFENCE)

$$\frac{ts' = ts[\text{v}_{\text{rNew}} \mapsto_{\sqcup} \sqcup_l ts.\text{coh}[l]]}{(\texttt{mfence}, ts), M \to_{tid} (\texttt{skip}, ts'), M}$$

Figure 4. States and transitions of x86_{view} (excerpt)

$ts \in$ TState consists of a register map, regs, assigning values to registers, and per-thread 'views' (described in §3.3). A memory is a list of messages; a message is a triple $\langle l := v \rangle_{tid}$ comprising a memory location (l), a value stored (v), and the id (tid) of the thread storing it. We write $\langle l := v \rangle_{tid} @ t$ to denote that $\langle l := v \rangle_{tid}$ is issued at timestamp (index) t, starting from index @1. For simplicity, we assume a memory contains the initial message $\langle l := 0 \rangle @0$ for each l.

Transitions of x86$_{view}$ In the initial state for a program p **(INIT)**, thread statements are those in p; the register maps are $\lambda\_.0$; the views are @0; and the memory is empty ([]).

The transitions for control flow and assignment are standard (omitted). The **(MACHINE)** transition of x86_{view} models thread interleaving as in sequential consistency (SC) [32].

Nevertheless, x86_{view} allows relaxed (weaker than SC) behaviors since it records the entire history of stores in its memory as a list of messages, and allows threads to read stale values. Ignoring the colored premises (described later), when executing a store **(STORE)**, a thread determines the location l and the value v, and appends a new message $\langle l := v \rangle$ to the memory. Analogously, when executing $r := \texttt{load}\ [e]$ **(LOAD)**, a thread determines the location l, chooses a message $\langle l := v \rangle @ t$ from the memory, and assigns v to r in the register map. Crucially, the chosen message need not be the latest one, thus allowing a stale value to be read. However, the chosen message should not have been overwritten **(NOT-OVERWRITTEN)** from the thread's point of view. We describe the remaining transitions shortly.

Table 1. Informal description of concurrency views

View	*Past*	*Future*
coh[l]	Upper bound of past reads and writes on l	Lower bound of future reads and writes on l
v_{rNew}	Upper bound of past updates; upper bound of external reads (from other threads)	Lower bound of future reads

Store Buffering Recording stores as messages allows store buffering, a representative relaxed behavior of Intel-x86:

$$\begin{array}{l} (a)\ \text{x} := 1 \\ (b)\ r_1 := \text{y}\ \ //= 0 \end{array} \Big\| \begin{array}{l} (c)\ \text{y} := 1 \\ (d)\ r_2 := \text{x}\ \ //= 0 \end{array} \qquad \text{(SB)}$$

While the relaxed outcome $r_1 = r_2 = 0$ is prohibited under SC, it is allowed under Intel-x86 and may arise in x86_{view} by: (a) writing $\langle \text{x} := 1 \rangle_{tid_1} @1$; ($b$) reading $\langle \text{y} := 0 \rangle @0$; ($c$) writing $\langle \text{y} := 1 \rangle_{tid_2} @2$; and most importantly, (d) reading the old value $\langle \text{x} := 0 \rangle @0$ that is overwritten by (a).

3.3 Concurrency Views

The model described thus far is too weak in that it allows behaviors prohibited under Intel-x86. We next describe how we strengthen the model to forbid such behaviors through *views*, as summarized in Table 1.

Coherence Intel-x86 orders loads and stores on the same location in a single thread as illustrated below:

$$\begin{array}{lll} (a)\ \text{x} := 20 & //= @2 \\ (b)\ \text{x} := 10 & //\neq @1 \\ (c)\ r_1 := \text{x} & //\neq @1 \end{array} \Big\| \begin{array}{lll} (d)\ r_2 := \text{y} & //= @4 \\ (e)\ \text{y} := 30 & //\neq @3 \\ (f)\ r_3 := \text{y} & //\neq @3 \end{array} \Big\| \cdots \qquad \text{(COH)}$$

The first thread issues $\langle \text{x} := 20 \rangle @2$, and then writes to x again and reads from it. Coherence orders (a) before (b) and (c) since they access the same location, thus forbidding them from accessing earlier timestamps, e.g., @1. Similarly, the second thread reads the message $\langle \text{y} := 40 \rangle @4$, and then writes to y and reads from it again. Coherence orders (d) before (e) and (f) as they access the same location, thus forbidding them from accessing earlier timestamps, e.g., @3.[7]

[7] Indeed, coherence between an access and a write is already enforced through **(STORE)** : stores always append messages to the end of memory. Nevertheless, we explicitly order them with views to achieve (i) uniformity with other coherence orders and (ii) correspondence with $\text{Armv8}_{\text{view}}$, where stores may add messages in places other than the end of memory.

To enforce coherence, we introduce *coherence views* that record past thread behaviors and simultaneously constrain future thread behaviors. Specifically, for each location l, a thread state ts records a coherence view in $ts.\mathsf{coh}[l]$ as a timestamps (initialised to @0) representing an index in memory. The $ts.\mathsf{coh}[l]$ represents the maximum (latest) timestamp observed for l by the thread; moreover, it forbids the thread from accessing messages of l with earlier timestamps than $ts.\mathsf{coh}[l]$. Put formally, in **(load)** and **(store)** we additionally require $ts.\mathsf{coh}[l] \sqsubseteq t$ in the premise and update $ts'.\mathsf{coh}[l] \mapsto t$ in the conclusion.

These changes indeed forbid the undesirable behavior in Coh: (a) updates $ts.\mathsf{coh}[\mathsf{x}]$ to @2, forbidding (b) and (c) from accessing @1. Similarly, (d) updates $ts.\mathsf{coh}[\mathsf{y}]$ to @4, forbidding (e) and (f) from accessing @3.

Message Passing In addition to coherence, Intel-x86 orders certain accesses on different locations via 'message passing':

$$\begin{array}{l|l} (a)\ \mathsf{data} := 42 & (c)\ r_1 := \mathsf{flag} \quad //= 1 \\ (b)\ \mathsf{flag} := 1 & (d)\ r_2 := \mathtt{data} \quad //\neq 0 \end{array} \tag{MP}$$

If the right thread reads 1 is from flag, then it should read 42 from data as (a) is ordered before (d) as follows:

- (a) **before** (b)**:** A load or store is ordered before later stores. To enforce this, in **(store)** we additionally require $\bigsqcup_l ts.\mathsf{coh}[l] \sqsubseteq t$ in the premise.[8]
- (b) **before** (c)**:** A store is ordered before loads that read from it ("message passing"). This is already enforced as the store message read by the load is issued before it.
- (c) **before** (d)**:** A load is ordered before a later load. To enforce this, we introduce the *new-read view*. Specifically, a thread state ts includes a 'new-read' view, $ts.\mathsf{v_{rNew}}$, recording the maximum (latest) view previously read by the thread. Moreover, it forbids the thread's future loads (on any location) from reading messages that are overwritten by $ts.\mathsf{v_{rNew}}$. Put formally, in **(load)** we require $ts.\mathsf{v_{rNew}} \sqsubseteq_{M,l} t$ (i.e., t is not overwritten by $ts.\mathsf{v_{rNew}}$ in M as far as l is concerned; see **(not-overwritten)** for details) in the premise and $ts'.\mathsf{v_{rNew}} \mapsto_\sqcup t$ (shorthand for $ts'.\mathsf{v_{rNew}} = ts.\mathsf{v_{rNew}} \sqcup t$) in the conclusion.

These changes ensure 'message passing' in MP: (a) the left thread issues $\langle \mathsf{data} := 42 \rangle @1$, updating $ts_1.\mathsf{coh}[\mathsf{data}]$ to @1; and (b) issues $\langle \mathsf{flag} := 1 \rangle @2$, updating $ts_1.\mathsf{coh}[\mathsf{flag}]$ to @2; (c) the right thread reads $\langle \mathsf{flag} := 1 \rangle @2$, updating $ts_2.\mathsf{coh}[\mathsf{flag}]$ and $ts_2.\mathsf{v_{rNew}}$ to @2; (d) it then cannot read $\langle \mathsf{data} := 0 \rangle @0$ as $ts_2.\mathsf{v_{rNew}} = @2 \not\sqsubseteq_{M,\mathsf{data}} @0$.

Store Buffering with Fences As shown in SB, Intel-x86 may reorder a store and a later load on different locations. If necessary, one can prevent this by inserting fences:

$$\begin{array}{l|l} (a)\ \mathsf{x} := 1 & (d)\ \mathsf{y} := 1 \\ (b)\ \mathtt{mfence} & (e)\ \mathtt{mfence} \\ (c)\ r_1 := \mathsf{y}\ //= 0 & (f)\ r_2 := \mathsf{x}\ //\neq 0 \end{array} \tag{SBFence}$$

To model this, in the conclusion of **(mfence)** we join $ts.\mathsf{v_{rNew}}$ with $\bigsqcup_l ts.\mathsf{coh}[l]$, thus forbidding store buffering. Without loss of generality, assume M=[$\langle \mathsf{x} := 1 \rangle @1$, $\langle \mathsf{y} := 1 \rangle @2$]. The right thread then (d) issues $\langle \mathsf{y} := 1 \rangle @2$, updating $ts_2.\mathsf{coh}[\mathsf{y}]$ to @2; (e) executes mfence, updating $ts_2.\mathsf{v_{rNew}}$ to @2; and (f) cannot read $\langle \mathsf{x} := 0 \rangle @0$ as $ts_2.\mathsf{v_{rNew}}$=@2 $\not\sqsubseteq_{M,\mathsf{x}} @0$.

Forwarding By strengthening $\mathrm{x86_{view}}$ we have precluded forbidden Intel-x86 behaviors. However, $\mathrm{x86_{view}}$ is now too strong and must be weakened to allow store forwarding:

$$\begin{array}{l|l} (a)\ \mathsf{x} := 1 & (d)\ \mathsf{y} := 1 \\ (b)\ r_1 := \mathsf{x}\ //= 1 & (e)\ r_3 := \mathsf{y}\ //= 1 \\ (c)\ r_2 := \mathsf{y}\ //= 0 & (f)\ r_4 := \mathsf{x}\ //= 0 \end{array} \tag{SBFwd}$$

While (b) and (c) are ordered, (a) and (c) are not because (b) is forwarded from (a) in the same thread, thus allowing the reordering of (a) after (b) and (c). To model this, in **(load)** the new-read view is joined with the read message's timestamp only if it is written by a different thread. This is denoted by the conditional notation $ts'.\mathsf{v_{rNew}} \mapsto_\sqcup t \neq ts.\mathsf{coh}[l]\ ?\ t$, stating that if $t \neq ts.\mathsf{coh}[l]$, then $ts'.\mathsf{v_{rNew}} \mapsto_\sqcup t$; otherwise, $ts'.\mathsf{v_{rNew}}$ is left unchanged. These changes then admit the behavior in SBFwd. Without loss of generality, assume M = [$\langle \mathsf{x} := 1 \rangle @1$, $\langle \mathsf{y} := 1 \rangle @2$]. The right thread (d) writes $\langle \mathsf{y} := 1 \rangle @2$, updating $ts_2.\mathsf{coh}[\mathsf{y}]$ to @2; (e) reads $\langle \mathsf{y} := 1 \rangle @2$, without updating $ts_2.\mathsf{v_{rNew}}$ thanks to forwarding; and (f) reads $\langle \mathsf{x} := 0 \rangle @0$ as $ts_2.\mathsf{v_{rNew}} = @0 \sqsubseteq_{M,\mathsf{x}} @0$.

(rmw-fail)

$$\frac{l = [\![e]\!]_{ts.\mathsf{regs}} \quad M[t] = \langle l := v \rangle \quad [\![rop]\!]_{ts.\mathsf{regs}}(v, \bot) \quad ts.\mathsf{coh}[l] \sqsubseteq t \quad ts.\mathsf{v_{rNew}} \sqsubseteq_{M,l} t \quad ts' = ts[\mathsf{regs}[r] \mapsto v, \mathsf{coh}[l] \mapsto t, \mathsf{v_{rNew}} \mapsto_\sqcup t \neq ts.\mathsf{coh}[l]\ ?\ t]}{(r := \mathtt{rmw}\ rop\ [e], ts), M \rightarrow_{tid} (\mathtt{skip}, ts'), M}$$

(rmw)

$$\frac{l = [\![e]\!]_{ts.\mathsf{regs}} \quad M[t_1] = \langle l := v_1 \rangle \quad [\![rop]\!]_{ts.\mathsf{regs}}(v_1, v_2) \quad t_2 = |M| + 1 \quad M' = M +\!\!+ [\langle l := v_2 \rangle_{tid} @ t_2] \quad t_2 - 1 \sqsubseteq_{M,l} t_1 \quad ts.\mathsf{coh}[l] \sqsubseteq t_1, t_2 \quad ts.\mathsf{v_{rNew}} \sqsubseteq_{M,l} t_1 \quad \sqcup_l ts.\mathsf{coh}[l] \sqsubseteq t_2 \quad ts' = ts[\mathsf{regs}[r] \mapsto v_1, \mathsf{coh}[l] \mapsto t_2, \mathsf{v_{rNew}} \mapsto_\sqcup \sqcup_l ts.\mathsf{coh}[l] \sqcup t_2]}{(r := \mathtt{rmw}\ rop\ [e], ts), M \rightarrow_{tid} (\mathtt{skip}, ts'), M'}$$

Figure 5. RMW transitions of $\mathrm{x86_{view}}$

3.4 Supporting Read-Modify-Writes (RMW)

The RMW transitions (Fig. 5) are obtained by combining the transitions of loads, stores and mfences. A failed RMW **(rmw-fail)** degenerates to a load;[9]; if an RMW fails, then

[8] The astute reader may have noticed that this condition is stronger than the coherence requirement $ts.\mathsf{coh}[l] \sqsubseteq t$ and thus makes it redundant. Nevertheless, we explicit include the two conditions to emphasize the two requirements, namely coherence and ordering.

[9] The semantics of failed RMWs in Intel-x86 is not fully agreed upon in the literature. Our model assumes a failed RMW to degenerate to a load; an alternative model may additionally assume that failed RMWs execute a

$$ts \in \mathsf{TState} \triangleq \left\langle \ldots;\quad \mathsf{v_{pReady}} : \mathbb{V};\quad \mathsf{v_{pAsync}}, \mathsf{v_{pCommit}} : \mathsf{Loc} \to \mathbb{V};\right\rangle$$

(FLUSH)

$$\frac{l = [\![e]\!]_{ts.\mathsf{regs}} \quad v = \sqcup_{l'} ts.\mathsf{coh}[l'] \quad ts' = ts[\mathsf{v_{pAsync}} \mapsto_\sqcup \lambda l'.\ cl(l,l')\ ?\ v, \mathsf{v_{pCommit}} \mapsto_\sqcup \lambda l'.\ cl(l,l')\ ?\ v]}{(\mathtt{flush}\ e, ts), M \to_{tid} (\mathtt{skip}, ts'), M}$$

(FLUSHOPT)

$$\frac{l = [\![e]\!]_{ts.\mathsf{regs}} \quad v = \sqcup_{l'} cl(l,l')\ ?\ ts.\mathsf{coh}[l'] \quad ts' = ts[\mathsf{v_{pAsync}} \mapsto_\sqcup \lambda l'.\ cl(l,l')\ ?\ (v \sqcup ts.\mathsf{v_{pReady}})]}{(\mathtt{flushopt}\ e, ts), M \to_{tid} (\mathtt{skip}, ts'), M}$$

(SFENCE)

$$\frac{ts' = ts[\mathsf{v_{pReady}} \mapsto_\sqcup \sqcup_l ts.\mathsf{coh}[l], \mathsf{v_{pCommit}} \mapsto_\sqcup ts.\mathsf{v_{pAsync}}]}{(\mathtt{sfence}, ts), M \to_{tid} (\mathtt{skip}, ts'), M}$$

(LOAD)

$$\frac{\cdots\ ts' = ts[\cdots, \mathsf{v_{pReady}} \mapsto_\sqcup t \neq ts.\mathsf{coh}[l]\ ?\ t]}{(r := \mathtt{load}\ [e], ts), M \to_{tid} (\mathtt{skip}, ts'), M}$$

(CRASH)

$$\frac{\forall l.\ \exists t.\ M[t] = \langle l := SM[l]\rangle\ \wedge\ \forall (\_, ts) \in \vec{T}.\ ts.\mathsf{v_{pCommit}}[l] \sqsubseteq_{M,l} t}{\langle \vec{T}, M\rangle \to_{\mathsf{crash}} SM}$$

Figure 6. States and transitions of $\mathrm{Px86}_{\mathrm{view}}$ where the highlighted rule denotes the *extension* of **LOAD** transition from Fig. 4 as shown; the premises of **MFENCE**, **RMW** and **RMW-FAIL** are analogously extended and omitted here.

$[\![rop]\!]_{ts.\mathsf{regs}}(v_1, \bot)$ holds (e.g., $[\![\mathtt{cas\ 4\ 5}]\!]_{rmap}(3, \bot)$ but not $[\![\mathtt{cas\ 4\ 5}]\!]_{rmap}(4, \bot)$[10]). A successful RMW **(RMW)** atomically reads from and writes to a location; if an RMW succeeds, then $[\![rop]\!]_{ts.\mathsf{regs}}(v_1, v_2)$ holds (e.g., $[\![\mathtt{cas\ 3\ 5}]\!]_{rmap}(3, 5)$ or $[\![\mathtt{fetch\text{-}add\ 1}]\!]_{rmap}(4, 5)$). Moreover, atomicity requires that there be no intervening messages on the same location between those read and written by the RMW; i.e., $t_2 - 1 \sqsubseteq_{M,l} t_1$. Lastly, as with mfences, we join $ts.\mathsf{v_{rNew}}$ with $\sqcup_l ts.\mathsf{coh}'[l]$.

As we show in §5, our $\mathrm{x86}_{\mathrm{view}}$ model is *equivalent* to the authoritative axiomatic model reviewed by Intel engineers.

3.5 Persistency Views

We next develop $\mathrm{Px86}_{\mathrm{view}}$ by extending $\mathrm{x86}_{\mathrm{view}}$ with persistency. As discussed in §2.3, the key idea is *persistency views*, determining persisted messages as summarized in Table 2.

Synchronous Flush As shown in Fig. 6, in order to model the behaviour of `flush` instructions synchronously, we extend a thread state ts with a persistency view, $ts.\mathsf{v_{pCommit}}$. For each location l, the $ts.\mathsf{v_{pCommit}}[l]$ denotes the maximum view (timestamp) of the messages on l that have persisted

memory fence. Nevertheless, we can straightforwardly adapt our model to support this by extending **(RMW-FAIL)** with the effects of **(MFENCE)**.

[10]Here we assume compare-and-swaps are strong: they do not fail spuriously. In our Coq formalization, we also support weak compare-and-swaps.

Table 2. Informal description of persistency views

View	*Past*	*Future*
$\mathsf{v_{pReady}}$	Upper bound of past external reads (from other threads)	Lower bound of messages to be asynchronously flushed by future `flushopt`
$\mathsf{v_{pAsync}}[l]$	Upper bound of past `flush`/`flushopt` on the same cache line as l	Lower bound of messages on l to be persisted by future fences/updates
$\mathsf{v_{pCommit}}[l]$	Upper bound of past (1) `flush` l'; and (2) `flushopt` l' followed by fences/updates, where l' is on the same cache line as l	Lower bound of persisted messages on l to survive a crash

to NVM. Executing a `flush` **(FLUSH)** determines the location l, and for each location l' on the same cache line as l, joins $ts.\mathsf{v_{pCommit}}[l']$ with the maximum coherence view v, thus persisting those messages of l' propagated to the thread (i.e., all earlier writes on l'). (The *asynchronous persistency view*, $ts.\mathsf{v_{pAsync}}[l']$, will be described shortly.) After a crash **(CRASH)**, the contents of NVM, SM ('sequential memory'), satisfy the following condition for each location l: $SM[l]$ holds the value of some message on l whose timestamp t is not overwritten by any thread's persistency view on l.

This indeed establishes the invariant $I \triangleq \mathsf{commit{=}1} \Rightarrow \mathsf{data{=}42}$ for COMMIT2. After executing the left thread, $M = [\langle \mathsf{data} := 42\rangle@1]$. The right thread (b) reads $\langle \mathsf{data} := 42\rangle@1$, updating $ts_2.\mathsf{coh}[\mathsf{data}]$ and $ts_2.\mathsf{v_{rNew}}$ to @1; (c) persists the message $ts_2.\mathsf{coh}[\mathsf{data}] = @1$, updating $ts_2.\mathsf{v_{pCommit}}[\mathsf{data}]$ to @1; and (d) writes $\langle \mathsf{commit} := 1\rangle@2$. After a crash, if $\langle \mathsf{commit} := 1\rangle@2$ has persisted, then (d) must have been executed; therefore $ts_2.\mathsf{v_{pCommit}}[\mathsf{data}] = @1$ and $\mathsf{data} = 42$.

Asynchronous Flush `flushopt` is a weaker variant of `flush` that may be reordered after certain instructions, and thus its execution may be delayed until a later fence/RMW. This may improve performance when persisting multiple locations:

```
(a) data1 := 42  ||  (c) if (data2 != 0) {
(b) data2 := 7   ||  (d)   flushopt data1
                 ||  (e)   flushopt data2      (COMMITOPT)
                 ||  (f)   sfence
                 ||  (g)   commit := 1 }
```

Similarly to COMMIT2, the invariant $I' \triangleq \mathsf{commit{=}1} \Rightarrow \mathsf{data1{=}42} \wedge \mathsf{data2{=}7}$ always holds. The `sfence` (f) awaits the completion of both (d) and (e), reducing I/O latency.

To model `flushopt` instructions, we extend a thread state ts with (1) $ts.\mathsf{v}_{\mathsf{pReady}}$, denoting the view to be persisted asynchronously at a subsequent `flushopt`; and (2) $ts.\mathsf{v}_{\mathsf{pAsync}}[l]$, denoting the maximum view of messages on l that have been persisted asynchronously.

The additional transitions of $\mathrm{Px86}_{\mathrm{view}}$ are given in Fig. 6. Executing `flushopt` l **(FLUSHOPT)** or `flush` l **(FLUSH)** joins, for each location l' on the same cache line as l, $ts.\mathsf{v}_{\mathsf{pAsync}}[l']$ with $ts.\mathsf{v}_{\mathsf{pReady}}$ and the maximum coherence view, v, of the cache line. Executing a fence in **(MFENCE)** and **(SFENCE)**, or a successful RMW in **(RMW)**, joins $ts.\mathsf{v}_{\mathsf{pCommit}}$ with $ts.\mathsf{v}_{\mathsf{pAsync}}$ and $ts.\mathsf{v}_{\mathsf{pReady}}$ with $\bigsqcup_l ts.\mathsf{coh}[l]$. Executing a load or a failed RMW in **(LOAD)** and **(RMW-FAIL)** joins $ts.\mathsf{v}_{\mathsf{pReady}}$ with the read message's timestamp unless forwarded.

This allows us to establish I' for CommitOpt. Without loss of generality, let $M = [\langle \mathsf{data1} := 42\rangle@1, \langle \mathsf{data2} := 7\rangle@2]$. The right thread then (c) reads $\langle \mathsf{data2} := 7\rangle@2$, updating $ts_2.\mathsf{coh}[\mathsf{data2}]$, $ts_2.\mathsf{v}_{\mathsf{rNew}}$ and $ts_2.\mathsf{v}_{\mathsf{pReady}}$ to @2; (d, e) asynchronously persists $ts_2.\mathsf{v}_{\mathsf{pReady}} = @2$ to data1 and data2, updating $ts_2.\mathsf{v}_{\mathsf{pAsync}}[\mathsf{data1}]$, $ts_2.\mathsf{v}_{\mathsf{pAsync}}[\mathsf{data2}]$ to @2; (f) awaits the completion of (d) and (e), updating $ts_2.\mathsf{v}_{\mathsf{pCommit}}[\mathsf{data1}]$ and $ts_2.\mathsf{v}_{\mathsf{pCommit}}[\mathsf{data2}]$ to @2; (g) writes $\langle \mathsf{commit} := 1\rangle@3$. After a crash, if $\langle \mathsf{commit} := 1\rangle@3$ is persisted, then (g) must have been executed; $ts_2.\mathsf{v}_{\mathsf{pCommit}} = [\mathsf{data1} \mapsto @2, \mathsf{data2} \mapsto @2]$; and thus data1 = 42 and data2 = 7.

The resulting model, $\mathrm{Px86}_{\mathrm{view}}$, is proven equivalent to the authoritative axiomatic model reviewed by Intel engineers [47] (modulo the fix discussed in §2.1 – see §5).

4 Fixing and Simplifying the Px86 Model

We present $\mathrm{Px86}_{\mathrm{axiom}}$, a new axiomatic model for Intel-x86 persistency that simplifies Px86 [47] and fixes its flaws discussed in §2.1. We present a short background on axiomatic models (§4.1); describe the baseline axiomatic model for Intel-x86 concurrency (§4.2); extend it to persistency and present $\mathrm{Px86}_{\mathrm{axiom}}$ (§4.3); and compare $\mathrm{Px86}_{\mathrm{axiom}}$ with Px86, proving their equivalence modulo our fixes in $\mathrm{Px86}_{\mathrm{axiom}}$ (§4.4).

4.1 Background on Axiomatic Models

Executions and Events In the literature of axiomatic (a.k.a. declarative) memory models, the traces of shared memory accesses of a program are represented as a set of *executions*, where each execution G is a graph comprising: (i) a set of events (graph nodes); and (ii) a number of relations on events (graph edges). We typically use a, b and e to range over events. Each event captures the execution of a primitive command (e.g., a load) and is a triple of the form $e{=}(n, \mathsf{tid}, l)$, where $n \in \mathbb{N}$ is the (unique) *event identifier*; $\mathsf{tid} \in \mathrm{TId}$ identifies the executing thread; and $l \in \mathrm{Lab}$ is the event *label*. Event labels are defined by the underlying memory model; for Intel-x86 a label l may be (1) $(\mathtt{R}, x, v)$ for reading (loading) value v from location x; (2) $(\mathtt{W}, x, v)$ for writing (storing) value v to location x; (3) $(\mathtt{U}, x, v, v')$ for a successful update (RMW) modifying x to v' when its value matches v; (4) `MF` for executing an `mfence`. The functions `loc`, `rval` and `wval` respectively project the location, the read value and the written value of a label, where applicable. For instance, $\mathtt{loc}(l){=}x$ and $\mathtt{wval}(l){=}v$ for $l{=}(\mathtt{W}, x, v)$. The functions `thrd` and `lab` respectively project the thread identifier and the label of an event.

Notation Given a relation r on a set A, we write $\mathsf{r}^?$ and r^+ for the reflexive and transitive closures of r, respectively. We write r^{-1} for the inverse of r; $[A]$ for the identity relation on A, i.e., $\{(a, a) \mid a \in A\}$; and A_x for $\{a \in A \mid \mathtt{loc}(a){=}x\}$. We write ri for the internal subset of r (on events of the same thread), i.e., $\mathsf{ri} \triangleq \{(a, b) \in \mathsf{r} \mid \mathtt{thrd}(a){=}\mathtt{thrd}(b)\}$; and re for the external subset of r (on events of different threads). Finally, we write $\mathsf{r}_1; \mathsf{r}_2$ for the relational composition of r_1 and r_2, i.e., $\{(a, b) \mid \exists c.\ (a, c) \in \mathsf{r}_1 \wedge (c, b) \in \mathsf{r}_2\}$.

Definition 4.1 (Executions)**.** An *execution*, G, is a tuple of the form $(E, \mathsf{po}, \mathsf{rf}, \mathsf{co})$, where:

- E is a set of events, including a set of *initialisation* events, $I \subseteq E$, comprising a single write event with label $(\mathtt{W}, x, 0)$ for each $x \in \mathrm{Loc}$. The set of *read* events in E is: $R \triangleq \{e \in E \mid \exists x, v.\ \mathtt{lab}(e){=}(\mathtt{R}, x, v)\}$; the sets of *writes* (W), *RMW* (U) and *memory fence* (MF) events are analogous.
- $\mathsf{po} \subseteq E \times E$ denotes the *'program-order' relation*, defined as a disjoint union of strict total orders, each ordering the events of one thread, together with $I \times (E \setminus I)$ that orders initialisation events before all others.
- $\mathsf{rf} \subseteq (W \cup U) \times (R \cup U)$ denotes the *'reads-from' relation* on events of the same location with matching values; i.e., $(a, b) \in \mathsf{rf} \Rightarrow \mathtt{loc}(a){=}\mathtt{loc}(b) \wedge \mathtt{wval}(a){=}\mathtt{rval}(b)$. Moreover, rf is total and functional on its range, i.e., every read/update is related to exactly one write/update. A read/update may be rf-related to an initialisation write.
- $\mathsf{co} \subseteq E \times E$ is the *'coherence-order'*, defined as the disjoint union of relations $\{\mathsf{co}_x\}_{x \in \mathrm{Loc}}$, such that each co_x is a strict total order on $W_x \cup U_x$ and $I_x \times ((W_x \cup U_x) \setminus I) \subseteq \mathsf{co}_x$.

In the context of an execution graph $(E, \mathsf{po}, \mathsf{rf}, \mathsf{co})$, we define the *'from-reads' relation* as $\mathsf{fr} \triangleq \mathsf{rf}^{-1}; \mathsf{co}$. Note that in this initial stage, executions are unrestricted: there are few constraints on rf and co. Such restrictions are determined by the set of model-specific *consistent* executions. We next define execution consistency for several models.

4.2 The $\mathrm{x86}_{\mathrm{axiom}}$ Model [3]

As the baseline axiomatic model for Intel-x86, we use that of Alglave et al. [3], presented in Fig. 7, which we refer to as $\mathrm{x86}_{\mathrm{axiom}}$.[11] We choose $\mathrm{x86}_{\mathrm{axiom}}$ as the baseline as it is stylistically similar with $\mathrm{Armv8}_{\mathrm{axiom}}$ [44], thus allowing a more uniform treatment of Intel-x86 and Armv8 persistency.

[11] For clarity, we rename the relations and axioms in [3] to highlight its similarity with the axiomatic model for Armv8 concurrency [44].

$$\mathsf{obs} = \mathsf{co} \cup \mathsf{rfe} \cup \mathsf{fre}$$
$$\mathsf{dob} = ([W \cup U \cup R]; \mathsf{po}; [W \cup U \cup R]) \setminus (W \times R)$$
$$\mathsf{bob} = [W \cup U \cup R]; \mathsf{po}; [MF]; \mathsf{po}; [W \cup U \cup R]$$
$$\mathsf{ob} = \mathsf{obs} \cup \mathsf{dob} \cup \mathsf{bob}$$
$(\mathsf{rf}; \mathsf{po}^?)$ irreflexive (CO-RW)
$(\mathsf{fr}; \mathsf{po})$ irreflexive (CO-WR)
ob acyclic (EXTERNAL)

Figure 7. The $\text{x86}_{\text{axiom}}$ model [3]

(axioms of $\text{x86}_{\text{axiom}}$ (Fig. 7))
$$\mathsf{fob} = [W \cup U \cup R]; \mathsf{po}; [FL] \cup ([U \cup R] \cup ([W]; \mathsf{po}; [MF \cup SF])); \mathsf{po}; [FO] \cup [W]; (\mathsf{po}; [FL])^?; (\mathsf{po} \cap \mathrm{CL}); [FO]$$
$\mathsf{ob} = \mathsf{obs} \cup \mathsf{dob} \cup \mathsf{bob} \cup \mathsf{fob} \cup \mathsf{pf} \cup \mathsf{fp}$ (*redefined*)
$\mathsf{pf} \subseteq (\mathsf{obs} \cup \mathsf{dob} \cup \mathsf{bob} \cup \mathsf{fob} \cup \mathsf{fp})^+$ (PF-MIN)
$P = dom(\mathsf{pf}; ([FL] \cup ([FO]; \mathsf{po}; [MF \cup SF \cup U])))$
$\forall l.\ \exists w.\ SM(l) = \mathtt{wval}(w) \ \wedge\ (P \times \{w\}) \cap \mathrm{Loc} \subseteq \mathsf{co}^?$ (PERSIST)

Figure 8. The $\text{Px86}_{\text{axiom}}$ model

The CO-RW ('coherence-read-write') axiom requires that loads not read from later stores; CO-WR ('coherence-write-read') ensures that loads do not read values overwritten by earlier stores; and EXTERNAL, ensures that externally visible events can be linearized with respect to the *'ordered-before'* relation (ob). The existence of such a globally-agreed order of events makes $\text{x86}_{\text{axiom}}$ multi-copy-atomic.

The ob relation enforces the order (a, b) if: (1) a is a store overwritten by b (co); (2) a is a store read by b in a different thread (rfe); (3) a reads a value overwritten by b in a different thread (fre); (4) a, b are accesses by the same thread and $(a, b) \notin W \times R$ (dob); or (5) a, b are accesses by the same thread and are separated by a fence (bob).

Coherence between two writes (resp. two reads) is derived from the axioms. Specifically, $\mathsf{co} \cup ([W \cup U]; \mathsf{po}; [W \cup U]) \subseteq \mathsf{ob}$ and acyclicity of ob ensure irreflexivity of co; po. Similarly, $(\mathsf{fre}; \mathsf{rfe}) \cup (\mathsf{fri}; \mathsf{rfi}) \cup ([U \cup R]; \mathsf{po}; [U \cup R]) \subseteq \mathsf{ob}$ and acyclicity of ob ensure irreflexivity of fr; rf; po.

4.3 The $\text{Px86}_{\text{axiom}}$ Model

We extend $\text{x86}_{\text{axiom}}$ with persistency semantics and develop the $\text{Px86}_{\text{axiom}}$ model as presented in Fig. 8. We first define:

- FL and FO: the set of synchronous flush (`flush`) and asynchronous flush (`flushopt`) events, respectively;
- SF: the set of `sfence` events;
- $\mathsf{pf} \subseteq (W \cup U) \times (FL \cup FO)$: the *'persists-from'* relation, relating each flush to the co-latest store for each location persisted by the flush. This is analogous to the rf relation; however, while rf relates a load to a *single* store, pf may relate a flush to multiple stores (one for each location) on the same cache line.
- $\mathsf{fp} \triangleq \mathsf{pf}^{-1}; \mathsf{co}$: the *'from-persists'* relation (analogous to fr), relating a flush to co-later stores (*cf.* $\mathsf{fr} \triangleq \mathsf{rf}^{-1}; \mathsf{co}$).

The ob relation is extended with fob ('flush-ordered-before'), ordering earlier events and a later flush as per the Intel manual [20]. Furthermore, pf and fp are included in ob for the same reason rf and fr are; i.e., because Intel-x86 is multi-copy atomic. P denotes the set of writes that must be persisted, i.e., those writes that are persisted by a synchronous (FL) or an asynchronous flush (FO) followed by a fence ($MF \cup SF \cup U$).[12] The PERSIST axiom states that in case of a crash, the persisted value (in NVM) of each location l in $SM[l]$ should not be coherence-before the writes in P. For simplicity, the PF-MIN axiom ensures that P is minimal, i.e., a flush persists only those writes that are strictly ordered before it. However, this minimality axiom is optional (Lemma 4.2).

Lemma 4.2. *A behavior is allowed under $\text{Px86}_{\text{axiom}}$ with axiom PF-MIN iff it is allowed under $\text{Px86}_{\text{axiom}}$ without PF-MIN.*

Proof. The proof is given in [9, Appendix C].

4.4 Comparing $\text{Px86}_{\text{axiom}}$ to Px86 in [47]

Fix Our $\text{Px86}_{\text{axiom}}$ model indeed fixes the Px86 shortcomings described in §2.1. In particular, as discussed in §2.1, we first strengthen Px86 to SPx86 by additionally requiring that flush instructions behave synchronously – see [9, Figs. 18 and 19] for the definitions of Px86 and SPx86.[13] In Theorem 4.3 below we then prove that $\text{Px86}_{\text{axiom}}$ and SPx86 are equivalent.

Theorem 4.3. *A behavior is allowed under SPx86 iff it is allowed under $\text{Px86}_{\text{axiom}}$.*

Proof. The proof is given in [9, Appendix D].

The Px86 and SPx86 models are based on the axiomatic Intel-x86 model known as TSO [41, 49], henceforth referred to as x86_{man} (given in [9, Fig. 18]). As such, in order to prove Theorem 4.3 we first show that x86_{man} and $\text{x86}_{\text{axiom}}$ are equivalent. In particular, existing equivalence results between x86_{man} and $\text{x86}_{\text{axiom}}$ cover loads and stores only and not RMWs and fences [2]. We extend this result for the first time to cover RMWs and fences in Theorem 4.4 below.

Theorem 4.4. *A behavior is allowed under x86_{man} iff it is allowed under x86_{axiom}.*

Proof. The proof is given in [9, Appendix D.2].

[12] One may expect an asynchronous flush to complete also when the thread terminates. But this is defined neither in the Intel manual [20] nor in its libraries [19]. We thus assume an asynchronous flush *not* to be completed when a thread terminates. However, we can easily change this by appending *TERM* to $MF \cup SF \cup U$, where *TERM* denotes thread termination. Analogously, we can adapt $\text{Px86}_{\text{view}}$ in §3 to account for terminated threads.
[13] For clarity, we adapted Px86 from [47] to match our style.

Simplification Our $\text{Px86}_{\text{axiom}}$ model is simpler than Px86 in [47] in the following aspects:

- While tso ('total store order'), nvo ('non-volatile order'), and P ('persisted stores') components of Px86 are *existentially quantified*, thus increasing non-determinism, the analogous ob and P in $\text{Px86}_{\text{axiom}}$ are *constructed*.
- While the conditions for intra-thread, inter-thread, and CPU-NVM communications are intertwined in Px86, they are separated and constrained by distinct axioms in $\text{Px86}_{\text{axiom}}$: intra-thread ones by CO-RW and CO-WR, inter-thread ones by EXTERNAL and CPU-NVM ones by PERSIST. To achieve this, $\text{Px86}_{\text{axiom}}$ orders fewer flush events than the Intel reference manual [20] does; e.g., unlike the manual, $\text{Px86}_{\text{axiom}}$ does not order FL before R.
- $\text{Px86}_{\text{axiom}}$ may optionally require the minimality of pf, which is beneficial for e.g., reducing the search space significantly for stateless model checking. By contrast, Px86 does not require a similar minimality in tso.

As we show in §5, the constructive and succinct nature of $\text{Px86}_{\text{axiom}}$ and its stylistic similarity to the axiomatic Armv8 model [44] make it easier to prove its equivalence to $\text{Px86}_{\text{view}}$.

5 Equivalence of $\text{Px86}_{\text{view}}$ and $\text{Px86}_{\text{axiom}}$

To evaluate the fidelity of $\text{Px86}_{\text{view}}$, we show that it is equivalent to $\text{Px86}_{\text{axiom}}$. To do this, we first prove the equivalence of x86_{view} and $\text{x86}_{\text{axiom}}$ by adapting the equivalence proof of the view-based and axiomatic models for Armv8 concurrency [44], and then generalize it to Intel-x86 persistency. All theorems in this section are mechanized in Coq [10].

Equivalence of x86_{view} and $\text{x86}_{\text{axiom}}$ In order to reuse the existing equivalence proof of the view-based and axiomatic models for Armv8 concurrency [44] maximally, we appeal to a new model, x86_{prom}, the *promising* view-based model for Intel-x86 concurrency, as the bridge between x86_{view} and $\text{x86}_{\text{axiom}}$. Compared to x86_{view}, x86_{prom} additionally allows 'promises', modeling speculative writes (see §6.2). Specifically, we employ the following proof strategy:

(1) We prove that x86_{view} and x86_{prom} are equivalent and that promises do not enable additional behaviors as their effect is cancelled out by concurrency views (Lemma 5.1).
(2) We prove that x86_{prom} and $\text{x86}_{\text{axiom}}$ are equivalent by adapting the analogous equivalence proof for Armv8 concurrency [44] as x86_{prom} and $\text{x86}_{\text{axiom}}$ respectively have the same style as the (view-based) $\text{Armv8}_{\text{view}}$ and (axiomatic) $\text{Armv8}_{\text{axiom}}$ models of Armv8 concurrency.

Combining the two steps we then establish the desired equivalence in Theorem 5.2.

Lemma 5.1. *A behavior is allowed under x86_{prom} iff it is allowed under x86_{view}.*

$$
\begin{array}{rcll}
 & \multicolumn{3}{l}{\ldots \text{ (the language for Intel-x86 in Fig. 3)}} \\
s \in \text{St} & ::= & \cdots & \textit{statement} \\
 & & | \; r := \texttt{load}_{xcl,rk}\ [e] & \textit{load} \\
 & & | \; r_{\text{succ}} := \texttt{store}_{xcl,wk}\ [e_1]\ e_2 & \textit{store} \\
 & & | \; \texttt{isb} \mid \texttt{dmb}.f \mid \texttt{dsb}.f & \textit{fence} \\
 & & | \; \texttt{flushopt}\ e & \textit{flush} \\
f \in \text{F} & ::= & \texttt{ld} \mid \texttt{st} \mid \texttt{sy} & \textit{order} \\
\\
xcl \in \mathbb{B} & ::= & \texttt{false} \mid \texttt{true} & \textit{exclusivity} \\
rk \in \text{RK} & ::= & \texttt{pln} \mid \texttt{wacq} \mid \texttt{acq} & \textit{read kind} \\
wk \in \text{WK} & ::= & \texttt{pln} \mid \texttt{wrel} \mid \texttt{rel} & \textit{write kind}
\end{array}
$$

Figure 9. The Armv8 concurrency/persistency language

Theorem 5.2. *A behavior is allowed under x86_{view} iff it is allowed under $\text{x86}_{\text{axiom}}$.*

Equivalence of $\text{Px86}_{\text{view}}$ and $\text{Px86}_{\text{axiom}}$ We next extend Theorem 5.2 to Intel-x86 persistency (Theorem 5.3). To do this, we relate each view of an x86_{view} execution to a set of events in the corresponding $\text{x86}_{\text{axiom}}$ execution; similarly for the persistency views in $\text{Px86}_{\text{view}}$. For example, the $\text{v}_{\text{pCommit}}$ view of a thread state is related to the set P of persisted writes in the corresponding $\text{Px86}_{\text{axiom}}$ execution. This then allows us to prove the equivalence of $\text{Px86}_{\text{view}}$ and $\text{Px86}_{\text{axiom}}$.

Theorem 5.3. *A behavior is allowed under $\text{Px86}_{\text{axiom}}$ iff it is allowed under $\text{Px86}_{\text{view}}$.*

6 View-Based and Axiomatic Models for Armv8 Persistency

In §3–5 we presented view-based and axiomatic models for Intel-x86 persistency and proved their equivalence. We next do the same for Armv8. As Intel-x86 and Armv8 persistency are highly similar, we focus on their differences (§6.1; see [9, Appendix B] for the full details). We then present the view-based Armv8 persistency model (§6.2), fix and simplify the axiomatic model for Armv8 persistency due to Raad et al. [48] as discussed in §2.2 (§6.3), and finally prove the equivalence of our view-based and axiomatic models (§6.4).

6.1 Armv8 versus Intel-x86 Persistency

We present the Armv8 language in Fig. 9, which is similar to that for Intel-x86 (Fig. 3), modulo the following:

Ordering: Armv8 ordering constraints are weaker and more elaborate than those of Intel-x86. Specifically, Armv8 loads and stores are annotated with *access ordering* constraints (rk or wk in Fig. 9). Moreover, Armv8 fences are more diverse: `isb` orders loads and later dependent accesses; $\texttt{dmb}.f$ orders accesses according to the ordering constraint f (see Fig. 9); and `dsb.sy` additionally awaits the completion of pending flush instructions.

Exclusivity: Unlike Intel-x86, Armv8 supports exclusive *load-link* and *store-conditional* instructions [25] that (if

successful) prohibit intervening stores between the load and store. Exclusive instructions are more primitive than RMWs: RMWs can be implemented via exclusive instructions but not vice versa.[14] As such, loads and stores are annotated with *exclusivity* tags (*xcl* in Fig. 9).

Flush: All Armv8 flushes are asynchronous (`flushopt`).

As we describe shortly, these differences are largely orthogonal to modeling persistency, except for the relaxed ordering of writes. Specifically, Armv8 allows (unlike Intel-x86) speculative execution of writes, interacting with NVM in an interesting way. To see this, we review the relaxed 'load buffering' behavior of Armv8 due to speculative writes:

$$\begin{array}{l|\!|l} (a)\ r_1 := \mathsf{y}\ \texttt{//=}\ 1 & (c)\ r_2 := \mathsf{x}\ \texttt{//=}\ 1 \\ (b)\ \mathsf{x} := 1 & (d)\ \mathsf{y} := 1 \end{array} \qquad \text{(LB)}$$

As Armv8 does not order a read and a subsequent write, (a) and (b) may be reordered; similarly for (c) and (d). As such, Armv8 allows an execution where (b), (d), (a), and (c) are executed in order, thus allowing the $r_1 = r_2 = 1$ behavior.

6.2 PArmv8$_{\text{view}}$: View-Based Armv8 Persistency

As with Px86$_{\text{view}}$, the view-based Armv8 persistency model, PArmv8$_{\text{view}}$, follows the same interleaving model over the history of stores. However, PArmv8$_{\text{view}}$ differs from Px86$_{\text{view}}$ in that (1) its views are more elaborate; and (2) it introduces *promises* to model speculative writes.

Views To model the ordering constraints and exclusivity of Armv8, the PArmv8$_{\text{view}}$ thread state in [9, Fig. 15] has additional view components compared to x86$_{\text{view}}$ in Fig. 4. These additional components are those of Armv8$_{\text{view}}$ [44]; i.e., the PArmv8$_{\text{view}}$ thread state is that of Armv8$_{\text{view}}$ extended with persistency views ($\mathsf{v}_{\mathsf{pReady}}$, $\mathsf{v}_{\mathsf{pAsync}}$ and $\mathsf{v}_{\mathsf{pCommit}}$ in §3.5).

Promises The additional views, however, are not sufficient to model LB: without further instrumentation, the model remains interleaving, where either (a) or (c) is executed first, reading the initial value 0.

To model speculative writes, Armv8$_{\text{view}}$ [44] introduces the notion of a *promise*: a message that may be speculatively added to the memory (or *promised*) without executing a store, provided that the promised message is later substantiated (or *fulfilled*) by executing a corresponding store. Put formally, a thread state ts contains the set $ts.\mathsf{prom}$ of the message ids that are promised by the thread but not yet fulfilled.

Using promises, we can model the LB behavior as follows, where tid_1 and tid_2 denote the left and right threads, respectively: (b-prom) tid_1 promises $\langle \mathsf{x} := 1 \rangle_{tid_1}$@1 with $ts_1.\mathsf{prom} = \{@1\}$; (c) tid_2 reads $\langle \mathsf{x} := 1 \rangle_{tid_1}$@1, updating $ts_2.\mathsf{coh}[\mathsf{x}]$ and $ts_2.\mathsf{v}_{\mathsf{rOld}}$ ('old-read view'[15]) to @1; (d) tid_2 writes $\langle \mathsf{y} := 1 \rangle_{tid_2}$ @2, updating $ts_2.\mathsf{coh}[\mathsf{y}]$ and $ts_2.\mathsf{v}_{\mathsf{wOld}}$ ('old-write view') to @2; (a) tid_1 reads $\langle \mathsf{y} := 1 \rangle_{tid_2}$@2, updating $ts_1.\mathsf{coh}[\mathsf{y}]$ and $ts_1.\mathsf{v}_{\mathsf{rOld}}$ to @2; and (b-fulfill) tid_1 fulfills $\langle \mathsf{x} := 1 \rangle_{tid_1}$@1, yielding $ts_1.\mathsf{prom} = \emptyset$ and $ts_1.\mathsf{v}_{\mathsf{wOld}} = @1$. Effectively, the write (b) is speculatively executed before the read (a) is executed.

To ensure that all speculations are substantiated, we require that a thread state's prom set be empty at the end of an execution; otherwise, the execution is deemed invalid.

Promises and Persistency The promises in PArmv8$_{\text{view}}$ similarly model speculative writes. Indeed, promises are largely orthogonal to persistency, except in the case of a crash. Specifically, in case of a crash in the presence of unfulfilled promises, we must determine the NVM contents.

On the one hand, one may argue that unfulfilled promises should persist (remain in NVM) as they have been made visible to other threads. To see this, consider Commit2 and suppose that the left thread promises $\langle \mathsf{data} := 42 \rangle$@1 which is yet unfulfilled, the right thread reads $\langle \mathsf{data} := 42 \rangle$@1 and writes $\langle \mathsf{commit} := 1 \rangle$@2, and then a crash occurs. If upon recovery $\langle \mathsf{commit} := 1 \rangle$@2 has persisted, then $\langle \mathsf{data} := 42 \rangle$@1 (which is an unfulfilled promise) should have also persisted.

On the other hand, one may argue that unfulfilled promises should not persist as they are not substantiated by a store. For example, suppose that the left thread in Commit2 promises to write $\langle \mathsf{data} := 23 \rangle$@1 without fulfilling it, and then it crashes. The promised write then should not persist as it is unsubstantiated; i.e., otherwise 23 appears *out-of-thin-air*.

To resolve this dilemma, we allow an execution to crash only if it has no unfulfilled promises. This then admits only the desired behaviors in Commit2: the execution cannot crash if either $\langle \mathsf{data} := 42 \rangle$@1 or $\langle \mathsf{data} := 23 \rangle$@1 is promised and not yet fulfilled. At first glance, this may seem restrictive as micro-architecturally an execution may crash even in the presence of uncommitted speculative writes. However, when this is the case, executing the remaining instructions to commit speculative writes does not constrain the NVM contents. Moreover, we formally justify our design by proving that PArmv8$_{\text{view}}$ and PArmv8$_{\text{axiom}}$ are equivalent (see §6.4).

6.3 PArmv8$_{\text{axiom}}$: Fixing and Simplifying PArmv8

We use the model of Pulte et al. [44] as the baseline axiomatic model for Armv8 concurrency, presented as Armv8$_{\text{axiom}}$ in [44, Appendix D].[16] The Armv8$_{\text{axiom}}$ model is equivalent to the authoritative axiomatic model in [43] which is reviewed by Arm engineers. Note that Armv8$_{\text{axiom}}$ has the same style as x86$_{\text{axiom}}$ in Fig. 7, except that: (1) all coherence constraints are captured by a single axiom (INTERNAL) since (CO-WW)

[14] While Armv8.1 also supports RMWs, they are currently missing in Armv8$_{\text{view}}$ and Armv8$_{\text{axiom}}$ [44]. Accordingly, we do not extend them to support RMWs as this is orthogonal to our objectives here.

[15] While reads update $\mathsf{v}_{\mathsf{rNew}}$ in x86$_{\text{view}}$, they update $\mathsf{v}_{\mathsf{rOld}}$ in Armv8$_{\text{view}}$. We refer the reader to [9, Appendix B] for more details.

[16] We refactor the relations in [44] to replace `dmb` with `dmb` ∪ `dsb`. The latter is a straightforward extension as `dsb` is strictly stronger than `dmb` [4].

$$
\begin{aligned}
&\text{(axioms of Armv8}_{\text{axiom}}\text{ [44] [9, Fig. 20])}\\
\mathrm{fob} &= [W \cup R]; \mathrm{po}; [\mathtt{dmb.sy} \cup \mathtt{dsb.sy}]; \mathrm{po}; [FO]\\
&\quad \cup [W \cup R]; (\mathrm{po} \cap \mathrm{CL}); [FO]\\
\mathrm{ob} &= \mathrm{obs} \cup \mathrm{dob} \cup \mathrm{aob} \cup \mathrm{bob} \cup \mathrm{fob} \cup \mathrm{pf} \cup \mathrm{fp} && \textit{(redefined)}\\
\mathrm{pf} &\subseteq (\mathrm{obs} \cup \mathrm{dob} \cup \mathrm{aob} \cup \mathrm{bob} \cup \mathrm{fob} \cup \mathrm{fp})^{+} && \textsc{(pf-min)}\\
P &= dom(\mathrm{pf}; [FO]; \mathrm{po}; [\mathtt{dsb.sy}])\\
\forall l.\ \exists w.\ SM(l) &= \mathtt{wval}(w) \ \wedge\ (P \times \{w\}) \cap \mathrm{Loc} \subseteq \mathrm{co}^{?} && \textsc{(persist)}
\end{aligned}
$$

Figure 10. The PArmv8$_{\text{axiom}}$ model

and (CO-RR) no longer follow from the other axioms;[17] (2) the ob component of Armv8$_{\text{axiom}}$ is more elaborate, modeling the weak ordering constraints of Armv8; and (3) Armv8$_{\text{axiom}}$ has an additional axiom (ATOMIC) that ensures the exclusivity of load-link/store-conditional instructions.

We next define an axiomatic model for Armv8 persistency, PArmv8$_{\text{axiom}}$ in Fig. 10, by extending Armv8$_{\text{axiom}}$ with persistency in the same style as Px86$_{\text{axiom}}$. The key differences from Px86$_{\text{axiom}}$ are that: (1) flush instructions impose different ordering constraints; (2) PArmv8$_{\text{axiom}}$ has no strong flush instructions; and (3) optimized flush instructions are guaranteed to commit only upon executing dsb.sy fences.

The PF-MIN axiom is optional as in Px86$_{\text{axiom}}$ (Lemma 4.2).

Fix Our PArmv8$_{\text{axiom}}$ model fixes the PArmv8 problem discussed in §2.2. Put formally, we prove the equivalence of PArmv8$_{\text{axiom}}$ and SPArmv8 which denotes strengthening PArmv8 by extending ob with pf and fp.

Theorem 6.1. *A behavior is allowed under SPArmv8 iff it is allowed under PArmv8$_{axiom}$.*

Proof. The proof is given in [9, Appendix E].

6.4 Equivalence of PArmv8$_{\text{view}}$ and PArmv8$_{\text{axiom}}$

Finally, we prove that PArmv8$_{\text{axiom}}$ and PArmv8$_{\text{view}}$ are equivalent by generalizing the analogous concurrency result in [44, Theorem 6.1] (showing that Armv8$_{\text{axiom}}$ and Armv8$_{\text{view}}$ are equivalent) and extending it with persistency.

Theorem 6.2. *A behavior is allowed under PArmv8$_{axiom}$ iff it is allowed under PArmv8$_{view}$.*

Proof. The proof is mechanized in [10].

7 Model Checking Persistency Patterns

We develop a stateless model checker for PArmv8$_{\text{view}}$ by generalizing and extending the Armv8$_{\text{view}}$ model checking framework in [44] to support persistency and account for crashes (§7.1). We use our model checker to verify representative persistent synchronization examples, including the AtomicPersists example [45] that emulates a persistent transaction [9, Appendix F]. Our model checking tool and verified examples are open source and publicly available [10].

[17] We could replace INTERNAL with irreflexivity of po; (co ∪ rf ∪ fr ∪ fr; rf) for uniformity with x86$_{\text{axiom}}$. We forwent this to use Armv8$_{\text{axiom}}$ [44] as is.

7.1 Model Checking Tool

Model Checking Tool for Armv8$_{view}$ We first briefly review the baseline model checking tool for Armv8$_{\text{view}}$ [44], which is a part of RMEM [5]. The tool consists of two parts: the executable model for sequential semantics of Armv8 ISA written in Sail [5]; and the executable memory model for concurrency written in Lem [39]. The former is adopted from [43], and the latter is split into two modes: the "promise-mode" which approximately enumerates the reachable final memories; and the "non-promise-mode" that checks if each potentially reachable final memory is actually reachable by thread executions to the end without promises. The two-mode execution is sound for the Armv8$_{\text{view}}$ model: a reachable state in Armv8$_{\text{view}}$ is also reachable by first promising to write all messages and then fulfilling the promises by executing the threads [44, Theorem 7.1].

Extension for PArmv8$_{view}$ We extend the model checking tool for Armv8$_{\text{view}}$ as follows: (1) we add persistency instructions to the executable model for sequential semantics in Sail; (2) we add persistency views to the executable memory model for Armv8$_{\text{view}}$ in Lem; (3) we enumerate not only final but also intermediate reachable memories in the promise–mode; and (4) we allow each thread's execution to stop amidst the non-promise-mode; and (5) we enumerate all post-crash states from the reachable states of intermediate memories and persistency views.

The performance of the resulting model checking algorithm for PArmv8$_{\text{view}}$ is similar to that for Armv8$_{\text{view}}$ because (1), (2), (4), (5) introduce only a constant-factor overhead; and the number of intermediate memories in (3) is usually dominated by that of final memories.

8 Related and Future Work

Related Work on Hardware Persistency Models Existing literature includes several works on formalising and testing hardware persistency models [11, 26, 28, 36, 42, 47, 48]. As discussed in detail in §2–6, the works of [47, 48] are closest to ours. Pelley et al. [42] propose several persistency models including epoch persistency; however, these models have not been adopted by mainstream architectures as of yet. Condit et al. [11], Joshi et al. [26] describe epoch persistency under x86-TSO [49]. Liu et al. [36] develop the PMTest testing framework for finding persistency bugs in software running over hardware models. Izraelevitz et al. [23] give a formal semantics of epoch persistency under release consistency [16]. As discussed in §1, the PTSO model of Raad and Vafeiadis [46] formalises epoch persistency under x86$_{\text{man}}$ (TSO) as a *proposal* for Intel-x86. However, PTSO is rather different from the existing Intel-x86 persistency model in Intel [20] in that it does not support the fine-grained Intel primitives for

selectively persisting cache lines (`flush` and `flushopt`), and instead proposes coarse-grained instructions (for persisting *all* locations at once) that do not exist in Intel-x86.

Khyzha and Lahav [28] recently developed the PTSO_{syn} model for Intel-x86 that satisfies the three properties of (A)–(C) discussed in §1. In particular, PTSO_{syn} supports persistent synchronization patterns even in the presence of I/O (B) as it also models `flush` instructions synchronously like $\text{Px86}_{\text{view}}$ (§3.5). However, this problem of asynchronous modeling of `flush` regarding I/O is not discussed in the paper.

Intel recently introduced Optane Persistent Memory 200 Series [22] that feature Enhanced Asynchronous DRAM Refresh (eADR), which treats processor caches as persistent (rather than volatile) by automatically flushing cache data to NVM in case of a crash. When eADR is available, a store is guaranteed to persist when made visible to other threads (e.g., after executing an `mfence`/`sfence`, but not `clflush`/`clflushopt`). Nevertheless, we believe that our contributions still stand for the following reasons. First, to ensure backwards compatibility, programs must support persistency in the absence of eADR. That is, a *correct* NVM program must defensively check whether eADR is enabled, and if not insert appropriate `clflush` or `clflushopt` instructions per our models. Second, eADR may increase runtime cost. For example, to flush cache data to NVM when a crash occurs, eADR must drain more power with higher voltage level or larger capacity, the impact of which on power consumption has not been thoroughly analyzed as of yet. The increased power consumption may affect embedded systems worse, and to our knowledge, Arm currently has no plans for supporting an eADR-like feature in Armv8.

Related Work on Software Persistency Models The literature on software persistency is more limited [8, 17, 30]. Kolli et al. [30] propose *acquire-release persistency*, an analogue to release-acquire consistency in C/C++. Gogte et al. [17] propose *synchronisation-free regions* (regions delimited by synchronisation operations or system calls). Although both approaches enjoy good performance, their semantic models are rather fine-grained, paving the way towards more coarse-grained transactional models [6, 19, 31, 48, 51, 52].

Related Work on Verification There are several works on implementing and verifying algorithms that operate on NVM. Friedman et al. [15] developed persistent queue implementations using Intel-x86 persist instructions (e.g., `flush`). Similarly, Zuriel et al. [54] developed persistent set implementations using Intel-x86 persist instructions. Derrick et al. [14] provided a formal correctness proof of the implementation in [54]. All three of [14, 15, 54] assume that the underlying concurrency model is sequential consistency [33], rather than x86_{man} (TSO). Recently, Raad et al. [45] developed a persistent program logic for verifying programs under the Px86 model. Finally, Kokologiannakis et al. [29] recently formalised the consistency and persistency semantics of the Linux ext4 filesystem, and developed a model-checking algorithm and tool for verifying the consistency and persistency behaviors of ext4 applications such as text editors

Future Work We plan to build on this work in several ways. First, we will empirically validate the proposed models w.r.t. NVM hardware using custom SoC (ASIC or FPGA) that captures the traffic between CPU and NVM, as proposed also in [47]. Second, we will explore language-level persistency by researching persistency extensions of high-level languages such as C/C++. This will liberate programmers from understanding hardware-specific persistency guarantees and make persistent programming more accessible. Third, we will first specify existing persistent libraries such as PMDK [19] and then use our model checker (§7) to verify their implementations against our specifications. Lastly, in the spirit of persistency semantics defining the order in which writes are propagated to NVM in DIMM slots, we will study the semantics in the presence of accelerators (e.g., CXL [13] and CCIX [12]), defining the order in which writes are propagated to accelerators in PCIe slots or other peripheral interconnects.

Acknowledgments

This work was supported by Institute for Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2019-0-00118, Research and Development on Memory-Centric OS Technologies of Unified Data Model for Next-Generation Shared/Hybrid Memory).

References

[1] 2020. The Coq Proof Assistant. https://coq.inria.fr/

[2] Jade Alglave. 2012. A Formal Hierarchy of Weak Memory Models. *Form. Methods Syst. Des.* 41, 2 (2012).

[3] 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 (2014).

[4] Arm. 2020. Arm architecture reference manual Armv8, for Armv8-A architecture profile (DDI 0487F.b). https://static.docs.arm.com/ddi0487/fb/DDI0487F_b_armv8_arm.pdf

[5] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. 2019. ISA Semantics for ARMv8-a, RISC-v, and CHERI-MIPS. *Proc. ACM Program. Lang.* 3, POPL (2019). https://doi.org/10.1145/3290384

[6] Hillel Avni, Eliezer Levy, and Avi Mendelson. 2015. Hardware Transactions in Nonvolatile Memory. In *Proceedings of the 29th International Symposium on Distributed Computing - Volume 9363* (Tokyo, Japan) *(DISC 2015)*. Springer-Verlag, Berlin, Heidelberg, 617–630. https://doi.org/10.1007/978-3-662-48653-5_41

[7] H. Alan Beadle, Wentao Cai, Haosen Wen, and Michael L. Scott. 2020. Nonblocking Persistent Software Transactional Memory. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Diego, California) *(PPoPP '20)*. Association for Computing Machinery, New York, NY, USA, 429–430. https://doi.org/10.1145/3332466.3374506

[8] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging Locks for Non-volatile Memory Consistency. *SIGPLAN Not.* 49, 10 (Oct. 2014), 433–452. https://doi.org/10.1145/2714064.2660224

[9] Kyeongmin Cho, Sung-Hwan Lee, Azalea Raad, and Jeehoon Kang. 2021. Appendix for Revamping Hardware Persistency Models: View-Based and Axiomatic Persistency Models for Intel-x86 and Armv8. https://cp.kaist.ac.kr/pmem

[10] Kyeongmin Cho, Sung-Hwan Lee, Azalea Raad, and Jeehoon Kang. 2021. Artifact for Revamping Hardware Persistency Models: View-Based and Axiomatic Persistency Models for Intel-x86 and Armv8. https://doi.org/10.1145/3410292

[11] 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 *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. ACM, New York, NY, USA, 133–146. https://doi.org/10.1145/1629575.1629589

[12] CCIX Consortium. [n.d.]. Cache Coherent Interconnect for Accelerators. https://www.ccixconsortium.com/

[13] CXL Consortium. [n.d.]. Compute Express Link. https://www.computeexpresslink.org/

[14] John Derrick, Simon Doherty, Brijesh Dongol, Gerhard Schellhorn, and Heike Wehrheim. 2019. Verifying Correctness of Persistent Concurrent Data Structures. In *Formal Methods – The Next 30 Years*, Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira (Eds.). Springer International Publishing, Cham, 179–195.

[15] Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. 2018. A Persistent Lock-Free Queue for Non-Volatile Memory. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Vienna, Austria) *(PPoPP '18)*. Association for Computing Machinery, New York, NY, USA, 28–40. https://doi.org/10.1145/3178487.3178490

[16] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. 1990. Memory Consistency and Event Ordering in Scalable Shared-memory Multiprocessors. *SIGARCH Comput. Archit. News* 18, 2SI (May 1990), 15–26. https://doi.org/10.1145/325096.325102

[17] Vaibhav Gogte, Stephan Diestelhorst, William Wang, Satish Narayanasamy, Peter M. Chen, and Thomas F. Wenisch. 2018. Persistency for Synchronization-free Regions. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. ACM, New York, NY, USA, 46–61. https://doi.org/10.1145/3192366.3192367

[18] Taeho Hwang, Jaemin Jung, and Youjip Won. 2015. HEAPO: Heap-Based Persistent Object Store. *ACM Trans. Storage* 11, 1, Article 3 (Dec. 2015), 21 pages. https://doi.org/10.1145/2629619

[19] Intel. 2015. Persistent Memory Programming. https://pmem.io/

[20] 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.

[21] Intel. 2019. Intel® Optane™ Persistent Memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html

[22] Intel. 2020. Intel® Optane™ Persistent Memory 200 Series Product Specifications. https://ark.intel.com/content/www/us/en/ark/products/series/203877/intel-optane-persistent-memory-200-series.html

[23] Joseph Izraelevitz, Hammurabi Mendes, and Michael L Scott. 2016. Linearizability of persistent memory objects under a full-system-crash failure model. In *International Symposium on Distributed Computing*. Springer, 313–327.

[24] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. 2019. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. arXiv:1903.05714 [cs.DC]

[25] Erik Jensen, G. W. Hagensen, and J. Broughton. 1987. A new approach to exclusive data access in shared memory multiprocessors.

[26] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. 2015. Efficient Persist Barriers for Multicores. In *Proceedings of the 48th International Symposium on Microarchitecture* (Waikiki, Hawaii) *(MICRO-48)*. ACM, New York, NY, USA, 660–671. https://doi.org/10.1145/2830772.2830805

[27] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A promising semantics for relaxed-memory concurrency. In *POPL 2017*.

[28] Artem Khyzha and Ori Lahav. 2021. Taming X86-TSO Persistency. *Proc. ACM Program. Lang.* 5, POPL (2021). https://doi.org/10.1145/3434328

[29] Michalis Kokologiannakis, Ilya Kaysin, Azalea Raad, and Viktor Vafeiadis. 2021. PerSeVerE: Persistency Semantics for Verification under Ext4. *Proc. ACM Program. Lang.* 5, POPL, Article 43 (Jan. 2021), 29 pages. https://doi.org/10.1145/3434324

[30] Aasheesh Kolli, Vaibhav Gogte, Ali Saidi, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. 2017. Language-level Persistency. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada) *(ISCA '17)*. ACM, New York, NY, USA, 481–493. https://doi.org/10.1145/3079856.3080229

[31] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. 2016. High-Performance Transactions for Persistent Memories. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, Georgia, USA) *(ASPLOS '16)*. Association for Computing Machinery, New York, NY, USA, 399–411. https://doi.org/10.1145/2872362.2872381

[32] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (July 1978).

[33] Leslie Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Computers* 28, 9 (Sept. 1979), 690–691. https://doi.org/10.1109/TC.1979.1675439

[34] Sung-Hwan Lee, Minki Cho, Anton Podkopaev, Soham Chakraborty, Chung-Kil Hur, Ori Lahav, and Viktor Vafeiadis. 2020. Promising 2.0: Global Optimizations in Relaxed Memory Concurrency *(PLDI 2020)*. https://doi.org/10.1145/3385412.3386010

[35] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. 2017. DudeTM: Building Durable Transactions with Decoupling for Persistent Memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 329–343. https://doi.org/10.1145/3037697.3037714

[36] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan. 2019. PMTest: A Fast and Flexible Testing Framework for Persistent Memory Programs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*.

[37] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. 2017. Octopus: an RDMA-enabled Distributed Persistent Memory File System. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 773–785. https://www.usenix.org/conference/atc17/technical-sessions/presentation/lu

[38] Youyou Lu, Jiwu Shu, and Long Sun. 2016. Blurred Persistence: Efficient Transactions in Persistent Memory. *ACM Trans. Storage* 12, 1, Article 3 (Jan. 2016), 29 pages. https://doi.org/10.1145/2851504

[39] Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell. 2014. Lem: Reusable Engineering of Real-World Semantics *(ICFP '14)*. https://doi.org/10.1145/2628136.2628143

[40] Ismail Oukid, Daniel Booss, Adrien Lespinasse, Wolfgang Lehner, Thomas Willhalm, and Grégoire Gomes. 2017. Memory Management Techniques for Large-Scale Persistent-Main-Memory Systems. *Proc. VLDB Endow.* 10, 11 (Aug. 2017), 1166–1177. https://doi.org/10.14778/3137628.3137629

[41] Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A Better X86 Memory Model: X86-TSO. In *TPHOL*.

[42] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. 2014. Memory Persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture* (Minneapolis, Minnesota, USA) *(ISCA '14)*. IEEE Press, 265–276.

[43] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2017. Simplifying ARM Concurrency: Multicopy-Atomic Axiomatic and Operational Models for ARMv8. 2, POPL (Dec. 2017).

[44] Christopher Pulte, Jean Pichon-Pharabod, Jeehoon Kang, Sung-Hwan Lee, and Chung-Kil Hur. 2019. Promising-ARM/RISC-V: A Simpler and Faster Operational Concurrency Model. In *PLDI 2019*.

[45] Azalea Raad, Ori Lahav, and Viktor Vafeiadis. 2020. Persistent Owicki-Gries Reasoning. *Proc. ACM Program. Lang.* 3, OOPSLA (2020).

[46] 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

[47] Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis. 2020. Persistency Semantics of the Intel-X86 Architecture. *Proc. ACM Program. Lang.* 4, POPL (2020).

[48] 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 (2019).

[49] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. X86-TSO: A Rigorous and Usable Programmer's Model for X86 Multiprocessors. *Commun. ACM* 53, 7 (2010).

[50] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. 2017. Distributed Shared Persistent Memory. In *Proceedings of the 2017 Symposium on Cloud Computing* (Santa Clara, California) *(SoCC '17)*. Association for Computing Machinery, New York, NY, USA, 323–337. https://doi.org/10.1145/3127479.3128610

[51] Hongping Shu, Hongyu Chen, Hao Liu, Youyou Lu, Qingda Hu, and Jiwu Shu. 2018. Empirical Study of Transactional Management for Persistent Memory. 61–66. https://doi.org/10.1109/NVMSA.2018.00015

[52] Arash Tavakkol, Aasheesh Kolli, Stanko Novakovic, Kaveh Razavi, Juan Gómez-Luna, Hasan Hassan, Claude Barthels, Yaohua Wang, Mohammad Sadrosadati, Saugata Ghose, Ankit Singla, Pratap Subrahmanyam, and Onur Mutlu. 2018. Enabling Efficient RDMA-based Synchronous Mirroring of Persistent Memory Transactions. *CoRR* abs/1810.09360 (2018). arXiv:1810.09360 http://arxiv.org/abs/1810.09360

[53] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Newport Beach, California, USA) *(ASPLOS XVI)*. Association for Computing Machinery, New York, NY, USA, 91–104. https://doi.org/10.1145/1950365.1950379

[54] 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