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

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

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

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.

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

ACM Reference Format:

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 *f*\_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 \(I \triangleq \text{commit}=1 \Rightarrow \text{data}=42\) 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 \(\text{commit}=1, \text{data}=0\) 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, \(f\_\text{flush}\) `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:

\[
\begin{align*}
(a) \quad \text{data} & \equiv 42 \\
(b) & \text{if } \text{(data} \not\equiv 0) \{ \\
(c) & \text{flush data} \\
(d) & \text{commit} \equiv 1 \}\quad \text{(Commit2)}
\end{align*}
\]

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 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 \(\text{commit}=1, \text{data}=0\) 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\(_\text{syn}\) [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., \(\text{commit}=1, \text{data}=0\) 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.
We develop a unified axiomatic style which formalizes the ambiguous and under-specified behavior described in the existing (axiomatic) persistency models of Intel-x86 and Armv8. Our operational models highlight 2 flaws in the existing (axiomatic) persistency models of Intel-x86 and Armv8. To remedy this, we develop a unified operational style for describing relaxed persistency and axiomatically.

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 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 Px86_view (§3.5) and PArmv8_view (§6.2), respectively extending the x86_view and Armv8_view [44] models to account for persistency.
- We present Px86axiom (§4) and PArmv8axiom (§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 PArmv8axiom has been reviewed by Arm engineers.
- We prove that Px86view and PArmv8view are equivalent to Px86axiom and PArmv8axiom, 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 PArmv8view (§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.

![Diagram of relationship among Intel-x86 and Armv8 models](image)

This diagram illustrates the 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 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 Px86_view (§3.5) and PArmv8_view (§6.2), respectively extending the x86_view and Armv8_view [44] models to account for persistency.
- We present Px86axiom (§4) and PArmv8axiom (§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 PArmv8axiom has been reviewed by Arm engineers.
- We prove that Px86view and PArmv8view are equivalent to Px86axiom and PArmv8axiom, 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 PArmv8view (§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:

\[
\begin{align*}
(a) \text{ data } & := 42 \\
(b) \text{ flush data } & \{ \\
& (d) \text{ if } (\text{flag } \neq 0) \{ \\
& (e) \log(\text{file}, "\text{commit}") \} \} \text{ (COMMIT1)} \\
& (c) \text{ flag } := 1
\end{align*}
\]

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

³In [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.
As such, to support persistency synchronization patterns such as COMMIT in the presence of external operations under Ptx86, we must strengthen Ptx86 by modeling the behavior of clflush synchronously. Let us write SPx86 for a strengthening of Ptx86 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 Ptx86. By contrast, in the presence of external operations, only SPx86 satisfies an invariant analogous to that of COMMIT: C ⇒ 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 PAvm8 Model and Multi-Copy Atomicity

The PAvm8 model [48] is too weak in that it violates the principles of multi-copy atomicity (MCA) that 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 PAvm8 violates MCA by allowing the following behavior regarding persistency:

(a) \( y := 1 \)
(b) \( \text{dsb} \_\text{sy} \)
(c) \( \text{flushopt} \_x \)
(d) \( \text{dsb} \_\text{sy} \)
(e) \( z := 1 \)
(f) \( x := 1 \)
(g) \( \text{dsb} \_\text{sy} \)
(h) \( \text{flushopt} \_y \) (FlushMCA)

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, \( \text{dsb} \_\text{sy} \), its effects take place synchronously; i.e., executing dmb \_sy awaits the completion of all earlier flushopt by the same thread.

We argue that MCA should preclude the post-crash state \( S: z = w = 1 \land x = y = 0 \). 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. Transitivity, \( g \) 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 PAvm8 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 SPAvm8 for such a strengthening of PAvm8. Under SPAvm8, if \( x = 0 \) after a crash, then \( c \) is ordered before \( f \); \( a \) is ordered before \( h \); and \( y = 1 \) is persist 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 \( \text{init} \) 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 \( \text{PVM}[l] = @0 \) (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 dmb \_sy at \( b \) and \( g \) are replaced with the weaker dmb \_sy. We opt for dmb \_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 fflush data joins the persistency view of data with the tid view (ν_NV = @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 I may contain a value written by a store whose timestamp is at least ν_NV[I]. For instance, if a crash occurs after executing fflush data, then in the post-crash state ν_NV[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 tid1 and tid2 denote the left and right threads, respectively. Initially, the memory is M = []; the persistency view is ν_NV = λL.@0; and the tid1 view is ν1 = @0 (for i ∈ {1, 2}). Then:

(a) Executing data := 42 yields M = [(data := 42)@1], ν1 = @1.

(b) Thread tid2 may then load data = 42 as its view timestamp (ν2 = @0) is less than @1 of data := 42. After loading data = 42, the tid2 view is joined with @1: ν2 = @1.

(c) Executing fflush data yields ν_NV[data] = @1.

(d) Executing commit := 1 results in M = [(data := 42)@1, ⟨commit = 1⟩@2] and ν = @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_view.6 Our model checking tool and verified examples are open-source and publicly available [10].

---

6As 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.
{(T, M) ∈ Machine \(\triangleleft\) (Tid \rightarrow Thread) \times Memory \\
tid ∈ Tid \triangleleft \mathbb{N} \ T ∈ Thread \triangleleft \mathbb{N} \times \mathcal{TState} \\
M ∈ Memory \triangleleft \text{list Msg} \ \ w ∈ \text{Msg} \triangleleft \langle \text{loc}: \text{Loc}; \text{val}: \text{Val}; \text{tid}: \text{Tid} \rangle \\
(l := v)_{\text{tid}} \triangleleft \langle \text{loc} = l; \text{val} = v; \text{tid} = \text{tid} \rangle \quad t ∈ T \triangleleft \mathbb{N} \quad v ∈ \mathcal{V} \triangleleft T \\
ts ∈ \mathcal{TState} \triangleleft \langle \text{regs} : \text{Reg} \rightarrow \text{Val}; \\
\text{cof} : \text{Loc} \rightarrow \mathcal{V}; \\
v_{\text{rNew}} : \mathcal{V}; \rangle \\
\text{(INIT)} \\
p = s_1 || \ldots || s_n \\
\text{init}(p, \langle \lambda tid. \ (s_{\text{tid}} = \langle \text{regs} = \lambda_-, \text{cof} = \lambda_-; @0; v_{\text{rNew}} = @0\rangle, [])\rangle)

\text{(MACHINE)} \\
T[\text{tid}], M \rightarrow_{\text{tid}} T', M' \\
\langle T, M \rangle \rightarrow \langle T'[\text{tid} \rightarrow T'], M'\rangle \\
\forall t ∈ \langle e_2, e_1 \rangle, M[t].\text{loc} \neq l \\
v_1 \subseteq_{\mathcal{M}, I} v_2

\text{(STORE)} \\
l \triangleleft \langle e_1 || \text{ts} : \text{regs} || e_2 \rangle || t \quad \text{ts.coh}[l] \subseteq t \quad t \subseteq \mathcal{T} \\
t = |M| + 1 \quad M' = M + \langle l := v \rangle_{\text{tid}@t} \\
ts' = \text{ts.coh}[l] \cap t \\
\langle \text{store} = \{e_1 \} e_2, ts\rangle, M \rightarrow_{\text{tid}} \langle \text{skip}, ts'\rangle, M'

\text{(LOAD)} \\
l \triangleleft \langle e || \text{ts} : \text{regs} || e_1 \rangle || t \quad \text{ts.vcoh} \subseteq \mathcal{M} / l \quad t \\
ts' = \text{ts}[r] \cap t, v_{\text{rNew}} \cap t \quad t \neq \text{ts.coh}[l] \cap t \\
\langle r := \text{load} = r, ts\rangle, M \rightarrow_{\text{tid}} \langle \text{skip}, ts'\rangle, M

\text{(MFENCE)} \\
ts' = \langle v_{\text{rNew}} \cup t \cup \text{ts.coh}[l] \rangle \\
\langle \text{mfence}, ts\rangle, M \rightarrow_{\text{tid}} \langle \text{skip}, ts'\rangle, M

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

state ts ∈ \mathcal{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 \text{loc} : \text{Loc}; \text{val} : \text{Val}; \text{tid} : \text{Tid} \rangle comprising a memory location (l), a value stored (v), and the id (tid) of the thread storing it. We write \langle l := v \rangle_{\text{tid}@t} to denote that \langle l := v \rangle_{\text{tid}@t} is issued at timestamp (index) t, starting from index \@1. For simplicity, we assume a memory contains the initial message \langle l := @0@0 \rangle for each l.

\text{Transitions of x86_{view}} In the initial state for a program p
\text{(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 \text{(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 \text{(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 := load [e]

\text{Table 1. Informal description of concurrency views}

<table>
<thead>
<tr>
<th>View</th>
<th>Past</th>
<th>Future</th>
</tr>
</thead>
<tbody>
<tr>
<td>coh[l]</td>
<td>Upper bound of past reads and writes on l</td>
<td>Lower bound of future reads and writes on l</td>
</tr>
<tr>
<td>v_{rNew}</td>
<td>Upper bound of past updates; upper bound of external reads (from other threads)</td>
<td>Lower bound of future reads</td>
</tr>
</tbody>
</table>

(LOAD), a thread determines the location \(l\), chooses a message \(\langle l := v \rangle_{\text{tid}@t}\) 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 \text{(NOT-OVERWRITTEN)} from the thread’s point of view. We describe the remaining transitions shortly.

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

\begin{align*}
(\text{a}) & \quad x := 1 \\
(\text{b}) & \quad r_1 := y \ // = 0 \\
(\text{c}) & \quad d := x \ // = 0 \\
\end{align*}

\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 \(x := 1)_{tid}@1; (b) reading \(y := 0)@0; (c) writing \(y := 1)_{tid}@2; and most importantly, (d) reading the old value \(x := 0)@0\) that is overwritten by (a).

\text{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.

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

\begin{align*}
(\text{a}) & \quad x := 20 \ // = @2 \\
(\text{b}) & \quad x := 10 \ // = @1 \\
(\text{c}) & \quad r_1 := x \ // = @1 \\
(\text{d}) & \quad r_2 := y \ // = @4 \\
(\text{e}) & \quad y := 30 \ // = @3 \\
(\text{f}) & \quad r_3 := y \ // = @3 \\
\end{align*}

\text{(Co)} The first thread issues (\(x := 20\)@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 \((y := 40)@4\), 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.

\footnote{Indeed, coherence between an access and a write is already enforced through \text{(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 Armv8_{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 \( I \), a thread state \( ts \) records a coherence view in \( ts.coh[I] \) as the timestamps (initialised to \( @0 \)) representing an index in memory. The \( ts.coh[I] \) represents the maximum (latest) timestamp observed for \( I \) by the thread; moreover, it forbids the thread from accessing messages of \( I \) with earlier timestamps than \( ts.coh[I] \). Put formally, in (load) and (store) we additionally require \( ts.coh[I] \subseteq t \) in the premise and update \( ts'.coh[I] \rightarrow t \) in the conclusion.

These changes indeed forbid the undesirable behavior in COH: (a) updates \( ts.coh[x] \) to \( @2 \), forbidding (b) and (c) from accessing \( @1 \). Similarly, (d) updates \( ts.coh[y] \) 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’:

\[
\begin{align*}
(a) \text{ data } & := 42 & (c) \ r1 := \text{ flag } & :/= 1 \\
(b) \text{ flag } & := 1 & (d) \ r2 := \text{ data } & :/\neq 0 \quad (\text{MP})
\end{align*}
\]

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 \( \lceil \text{load, flag, store} \rceil \subseteq t \) 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, \( ts.vNew \), 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 \( ts.vNew \). Put formally, in (load) we require \( ts.vNew \subseteq M.t \) (i.e., \( t \) is not overwritten by \( ts.vNew \) in \( M \) as far as \( I \) is concerned; see (not-overwritten) for details) in the premise and \( ts'.vNew \overset{\rightarrow}{\mapsto} t \) (shorthand for \( ts'.vNew = ts.vNew \cup t \) in the conclusion).

These changes ensure 'message passing' in MP: (a) the left thread issues \( \text{data } := 42 @1 \), updating \( ts.coh[\text{data}] \) to \( @1 \); and (b) issues \( \text{flag } := 1 @2 \), updating \( ts.coh[\text{flag}] \) to \( @2 \); (c) the right thread reads \( \text{flag } := 1 @2 \), updating \( ts.coh[\text{flag}] \) and \( ts.vNew \) to \( @2 \); (d) it then cannot read \( \text{data } := 0 @0 \) as \( ts.vNew = @2 \cup M.\text{data } @0 \).

**Store Buffering with Fences** As shown in SB, Intel-x86 may reorder a store and a later load on different locations. If

\[ (\text{RMW-FAIL}) \quad I = \left[ e \right]_{ts.\text{regs}} \quad M[t] = \left\{ l := 0 \right\} \quad \text{rop}_{ts.\text{regs}}(0, \bot) \quad \text{ts.coh}[I] \subseteq t \quad ts'.vNew \subseteq M.t \]

\[ ts' = ts[\text{regs}[r] \mapsto t, v.coh[I] \mapsto t, ts'.vNew \mapsto t \neq ts.coh[I] ? t] \]

\[ (r := \text{rmw rop } [e], ts), M \rightarrow_{\text{tid}} (\text{skip, ts'}), M \]

\[ \quad \text{(RMW)} \]

\[ I = \left[ e \right]_{ts.\text{regs}} \quad M[t] = \left\{ l := 01 \right\} \quad \text{rop}_{ts.\text{regs}}(01, 02) \quad t2 = [M] + 1 \quad M' = M \leftrightarrow \left\{ l := 01 \right\}_{\text{tid}@2} \quad t2 := M \quad \text{ts.coh}[I] \subseteq t1, t2 \quad ts'.vNew \subseteq M.t \]

\[ ts' = ts[\text{regs}[r] \mapsto 01, v.coh[I] \mapsto 02, ts'.vNew \mapsto t1 \cup l y \text{ts.coh}[I] \cup t2] \]

\[ (r := \text{rmw rop } [e], ts), M \rightarrow_{\text{tid}} (\text{skip, ts'}), M' \]

\[ \quad \text{Figure 5. RMW transitions of x86view} \]

necessary, one can prevent this by inserting fences:

\[ \begin{align*}
(a) \ x & := 1 & (b) \ y & := 1 \\
(b) \ m\text{fence } & \text{ } & (c) \ r1 & := \text{ y } / /= 0 \\
(c) \ r1 & := y & / /= 0 & (d) \ r2 & := x / /= 0 \quad (\text{SBFence})
\end{align*} \]

To model this, in the conclusion of (mFence) we join \( ts.vNew \) with \( \lceil \text{load, flag, store} \rceil \), thus forbidding store buffering. Without loss of generality, assume \( M = \{ (x := 1) @1, (y := 1) @2 \} \). The right thread then \( (d) \) issues \( \{ y := 1 \} @2 \), updating \( ts.coh[y] \) to \( @2 \); (e) executes \( m\text{fence } \), updating \( ts.vNew \) to \( @2 \); and \( (f) \) cannot read \( \{ x := 0 \} @0 \) as \( ts.vNew = @2 \cup M.v @0 \).

**Forwarding** By strengthening x86view we have precluded forbidden Intel-x86 behaviors. However, x86view is now too strong and must be weakened to allow store forwarding:

\[ \begin{align*}
(a) \ x & := 1 & (b) \ y & := 1 \\
(b) \ r1 & := x / /= 1 & (c) \ r3 & := y / /= 1 \\
(c) \ r2 & := y & / /= 0 & (f) \ r4 & := x / /= 0 \quad (\text{SBFWd})
\end{align*} \]

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 \( ts'.vNew \overset{\rightarrow}{\mapsto} t \neq ts.coh[I] ? t \), stating that if \( t \neq ts.coh[I] \), then \( ts'.vNew \overset{\rightarrow}{\mapsto} t \); otherwise, \( ts'.vNew \) is left unchanged. These changes then admit the behavior in SBFWd. Without loss of generality, assume \( M = \{ (x := 1) @1, (y := 1) @2 \} \). The right thread \( (d) \) writes \( \{ y := 1 \} @2 \), updating \( ts.coh[y] \) to \( @2 \); (e) reads \( \{ y := 1 \} @2 \), without updating \( ts.vNew \), thanks to forwarding; and \( (f) \) reads \( \{ x := 0 \} @0 \) as \( ts.vNew = @0 \cup M.v @0 \).

### 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;\(^3\) if an RMW fails, then

\(^{3}\)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
Future

We next develop $\text{Px86}_{\text{l}}$ 10 previously. In our Coq formalization, we also support weak compare-and-swaps. To support this by extending $\text{RMW-FAIL}$ with the effects of $\text{MFENCE}$.

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

$\text{ts} \in \text{TState} \equiv \{ \ldots; \ \text{v}_{\text{pReady}} \mapsto \forall; \ \text{v}_{\text{pAsync}}, \text{v}_{\text{pCommit}} : \text{Loc} \mapsto \forall; \}$

**FLUSH**

\[
\frac{L = [e]_{\text{ts.reg}} \ \text{v} = \sqcup \text{t}_s \text{ts.coh}([l'])}{ts' = ts[\text{v}_{\text{pAsync}} \mapsto \lambda' \cdot \text{cl}(l, l') \wedge \text{cl}(l, l') \wedge \text{v} \wedge \text{v}_{\text{pCommit}} \mapsto \lambda' \cdot \text{cl}(l, l') \wedge \text{v}]}.
\]

$(\text{flush} e, ts), M \rightarrow_{\text{tid}} (\text{skip}, ts'), M$

**FLUSHOPT**

\[
\frac{L = [e]_{\text{ts.reg}} \ \text{v} = \sqcup \text{t}_s \text{cl}(l, l') \wedge \text{ts.coh}([l'])}{ts' = ts[\text{v}_{\text{pasync}} \mapsto \lambda' \cdot \text{cl}(l, l') \wedge \text{cl}(l, l') \wedge \text{v} \wedge \text{v}_{\text{pasync}} \mapsto \lambda' \cdot \text{cl}(l, l') \wedge \text{v}]}.
\]

$(\text{flushopt} e, ts), M \rightarrow_{\text{tid}} (\text{skip}, ts'), M$

**SFENCE**

\[
\frac{ts' \equiv ts[\text{v}_{\text{pasync}} \mapsto \lambda \cdot \text{ts.coh}([l]), \text{v}_{\text{pCommit}} \mapsto \lambda \cdot \text{ts.pAsync}]}{(\text{sfence} e, ts), M \rightarrow_{\text{tid}} (\text{skip}, ts'), M}
\]

**LOAD**

\[
\frac{\ldots \ldots ts' = \ldots \cdot \text{v}_{\text{pasync}} \mapsto \lambda \cdot \text{ts.coh}([l]) \wedge \text{v}}{(r := \text{load} [e], ts), M \rightarrow_{\text{tid}} (\text{skip}, ts'), M}
\]

**CRASH**

\[
\forall l. \exists t. M[l] = (l := SM[l]) \land \forall(_{-}, ts) \in \mathcal{T}\ . \ ts.p_{\text{Commit}}[l] \sqsubseteq_{\text{tid}} t \ (\bar{T}, M, \rightarrow_{\text{crash}} SM)
\]

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

$[\text{rop}]_{\text{ts.reg}}(v_1, \bot)$ holds (e.g., $[\text{cas} 45]_{\text{rmapped}}(3, \bot)$ but not $[\text{cas} 45]_{\text{rmapped}}(4, \bot)$). A successful $\text{RMW}$ ($\text{RMW}$) atomically reads from and writes to a location; if an $\text{RMW}$ succeeds, then $[\text{rop}]_{\text{ts.reg}}(v_1, v_2)$ holds (e.g., $[\text{cas} 35]_{\text{rmapped}}(3, 5)$ or $[\text{fetch} + \text{add}]_{\text{rmapped}}(4, 5)$). Moreover, atomicty requires that there be no intervening messages on the same location between those read and written by the $\text{RMW}$; i.e., $t_2 - t_1 \leq_{\text{tid}} t_1$. Lastly, as with mfenices, we join $ts_{\text{VNew}}, \text{VNew}$ with $[\bot]_{\text{ts.coh}[l]}$.

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

### 3.5 Persistency Views

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

**Asynchronous Flush** $\text{flushopt}$ is a weaker variant of $\text{flush}$ 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:

\[
\begin{aligned}
(a) & \text{data1} := \text{data1} + \text{data2} \\
(b) & \text{data2} := 7 \\
(c) & \text{if} (\text{data2} = 0) \{ \\
(d) & \text{flushopt data1} \\
(e) & \text{flushopt data2} \quad \text{(COMMITOPT)} \\
(f) & \text{s.fence} \\
(g) & \text{commit} := 1 \}
\end{aligned}
\]

Similarly to $\text{COMMIT2}$, the invariant $l \sim commit=1 \Rightarrow data1=42 \land data2=7$ always holds. The $\text{s.fence}$ (f) awaits the completion of both (d) and (e), reducing I/O latency.

<table>
<thead>
<tr>
<th>View</th>
<th>Past</th>
<th>Future</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{v}_{\text{pReady}}$</td>
<td>Upper bound of past external reads (from other threads)</td>
<td>Lower bound of messages to be asynchronously flushed by future $\text{flushopt}$</td>
</tr>
<tr>
<td>$\text{v}_{\text{pAsync}}[l]$</td>
<td>Upper bound of past $\text{flushopt}$ on the same cache line as $l$</td>
<td>Lower bound of messages on $l$ to be persisted by future fences/updates</td>
</tr>
<tr>
<td>$\text{v}_{\text{pCommit}}[l]$</td>
<td>Upper bound of past (1) $\text{flush}/\text{flushopt}$ on the same cache line as $l$; (2) $\text{flushopt}$’ followed by fences/updates, where $l$’ is on the same cache line as $l$</td>
<td>Lower bound of persisted messages on $l$ to survive a crash</td>
</tr>
</tbody>
</table>

Table 2. Informal description of persistency views
To model flushopt instructions, we extend a thread state $t_s$ with (1) $t_s.v_{p\text{Ready}}$, denoting the view to be persisted asynchronously at a subsequent flushopt; and (2) $t_s.v_{p\text{Async}}[l]$, denoting the maximum view of messages on $l$ that have been persisted asynchronously.

The additional transitions of Px86$_{\text{view}}$ are given in Fig. 6. Executing flushopt $I$ (flushopt) or flush $I$ (flush) joins, for each location $l'$ on the same cache line as $l$, $t_s.v_{p\text{Async}}[l']$ with $t_s.v_{p\text{Ready}}$ and the maximum view, $v$, of the cache line. Executing a fence in (mfence) and (sfence), or a successful RMW in (rmw), joins $t_s.v_{p\text{Commit}}$ with $t_s.v_{p\text{Async}}$ and $t_s.v_{p\text{Ready}}$ with $\lfloor j | t.s.coh[j] \rfloor$. Executing a load or a failed RMW in (load) and (rmw-fail) joins $t_s.v_{p\text{Ready}}$ with the read message’s timestamp unless forwarded.

This allows us to establish $I'$ for CommitOpt. Without loss of generality, let $M = \{ (\text{data}1 := 42) @ 2 \} \cup \{ (\text{data}2 := 7) @ 2 \}$. The right thread then (c) reads $\langle \text{data}2 := 7 @ 2 \rangle$; updating $t_s.v_{p\text{New}}[\text{data}2]$ and $t_s.v_{p\text{Ready}}[\text{data}2]$, (c) asynchronously persists $t_s.v_{p\text{Ready}}[\text{data}2]$ to $\text{data}1$ and $\text{data}2$, updating $t_s.v_{p\text{Async}}[\text{data}1], t_s.v_{p\text{Async}}[\text{data}2]$ to $\text{data}2$, (f) waits the completion of (d) and (e), updating $t_s.v_{p\text{Commit}}[\text{data}1]$, and $t_s.v_{p\text{Commit}}[\text{data}2]$ to $\text{data}2$. (g) writes $\langle \text{commit} = 1 @ 3 \rangle$. After a crash, if (commit := 1) @ 3 is persisted, then (g) must have been executed; $t_s.v_{p\text{Commit}}[\text{data}1] \mapsto @ 2, \text{data}2 \mapsto @ 2$; and thus data1 = 42 and data2 = 7.

The resulting model, Px86$_{\text{view}}$, 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 Px86$_{\text{axiom}}$, 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 Px86$_{\text{axiom}}$ (§4.3); and compare Px86$_{\text{axiom}}$ with Px86, proving their equivalence modulo our fixes in Px86$_{\text{axiom}}$ (§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 $a$, $b$, $e$ to range over events. Each event captures the execution of a primitive command (e.g., a load) and is a triple of the form $e=(n, t_i d, l_i)$, where $n \in \mathbb{N}$ is the (unique) event identifier; $t_i d \in T l d$ identifies the executing thread; and $l \in L a b$ is the event label. Event labels are defined by the underlying memory model; for Intel-x86 a label $l$ may be (1) $(R, x, v)$ for reading (loading) value $v$ from location $x$; (2) $(W, x, v)$ for writing (storing) value $v$ to location $x$; (3) $(U, x, v, v')$ for a successful update (RMW) modifying $x$ to $v'$ when its value matches $v$; (4) $MF$ for executing an mfence. The functions $\text{loc}$, $\text{rval}$ and $\text{wval}$ respectively project the location, the read value and the written value of a label, where applicable. For instance, $\text{loc}(l) = x$ and $\text{wval}(l) = v$ for $l = (W, x, o)$. The functions $\text{thrd}$ and $\text{lab}$ respectively project the thread identifier and the label of an event.

Notation

Given a relation $r$ on a set $A$, we write $r^+$ 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 $\{ a \in A \mid \text{loc}(a) = x \}$. We write $r_{\text{int}}$ for the internal subset of $r$ (on events of the same thread), i.e., $r_{\text{int}} = \{ (a, b) \in r \mid \text{thrd}(a) = \text{thrd}(b) \}$; and $r_{\text{ext}}$ for the external subset of $r$ (on events of different threads). Finally, we write $r_1 \cdot r_2$ for the relational composition of $r_1$ and $r_2$, i.e., $\{ (a, b) \mid (a, c) \in r_1 \land (c, b) \in r_2 \}$.

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

- $E$ is a set of events, including a set of initialisation events, $I \subseteq E$, comprising a single write event with label $(W, x, 0)$ for each $x \in L o c$. The set of read events in $E$ is: $R \triangleq \{ e \in E \mid \exists x, v, \text{lab}(e) = (R, x, v) \}$; the sets of writes $(W)$, RMW $(U)$ and memory fence $(MF)$ events are analogous.
- $po \subseteq E \times E$ 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.
- $rf \subseteq (W \cup U) \times (R \cup U)$ denotes the ‘reads-from’ relation on events of the same location with matching values; i.e., $(a, b) \in rf$ $\Rightarrow$ $\text{loc}(a) = \text{loc}(b) \land \text{rval}(a) = \text{rval}(b)$. 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.
- $co \subseteq E \times E$ is the ‘coherence-order’, defined as the disjoint union of relations $\{ \text{co}_{x} \}_{x \in L o c}$, such that each $\text{co}_x$ is a strict total order on $W_x \cup U_x$ and $I_x \times ((W_x \cup U_x) \setminus I_x) \subseteq \text{co}_x$.

In the context of an execution graph $(E, po, rf, co)$, we define the ‘from-reads’ relation as $fr \triangleq rf^{-1}$; 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 $x86_{\text{axiom}}$ 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 $x86_{\text{axiom}}$. We choose $x86_{\text{axiom}}$ as the baseline as it is stylistically similar with Armv8$_{\text{axiom}}$ [44], thus allowing a more uniform treatment of Intel-x86 and Armv8 persistency.

\footnote{For clarity, we rename the relations and axioms in [3] to highlight its similarity with the axiomatic model for Armv8 concurrency [44].}
We extend x86-4.3 The Px86 model as presented in Fig. 8. We first define:

- FL and FO: the set of synchronous flush (fflush) and asynchronous flush (fflushopt) events, respectively;
- SF: the set of sffence events;
- \( pf \subseteq (W \cup U) \times (FL \cup FO) \): 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.

- \( fp \triangleq pf^{-1} \cap co \): the ‘from-persists’ relation (analogous to \( rf \)), relating a flush to co-later stores (cf. \( fr \triangleq rf^{-1} \cap co \)).

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 (MFUSF).\(^{12}\)

The ob relation 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 states that if \( P \) 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 \( Px86_{axiom} \) with axiom \( PF-MIN \) iff it is allowed under \( Px86_{axiom} \) without \( PF-MIN \).

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

### 4.4 Comparing \( Px86_{axiom} \) to Px86 in [47]

**Fix.** Our \( Px86_{axiom} \) 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 \( Px86_{axiom} \) and SPx86 are equivalent.

**Theorem 4.3.** A behavior is allowed under SPx86 iff it is allowed under \( Px86_{axiom} \).

**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 \( x86_{axiom} \) are equivalent. In particular, existing equivalence results between \( x86_{man} \) and \( x86_{axiom} \) 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 \( Px86_{view} \) in §3 to account for terminated threads.

\(^{13}\)For clarity, we adapted Px86 from [47] to match our style.
Simplification Our Px86\textsubscript{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 Px86\textsubscript{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 Px86\textsubscript{axiom}: intra-thread ones by CO-RW and CO-WR, inter-thread ones by EXTERNAL and CPU-NVM ones by PERSIST. To achieve this, Px86\textsubscript{axiom} orders fewer flush events than the Intel reference manual [20] does; e.g., unlike the manual, Px86\textsubscript{axiom} does not order FL before R.
- Px86\textsubscript{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 Px86\textsubscript{axiom} and its stylistic similarity to the axiomatic Armv8 model [44] make it easier to prove its equivalence to Px86\textsubscript{view}.

5 Equivalence of Px86\textsubscript{view} and Px86\textsubscript{axiom}

To evaluate the fidelity of Px86\textsubscript{view}, we show that it is equivalent to Px86\textsubscript{axiom}. To do this, we first prove the equivalence of x86\textsubscript{view} and x86\textsubscript{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\textsubscript{view} and x86\textsubscript{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\textsubscript{prom}, the promising view-based model for Intel-x86 concurrency, as the bridge between x86\textsubscript{view} and x86\textsubscript{axiom}. Compared to x86\textsubscript{view}, x86\textsubscript{prom} additionally allows 'promises', modeling speculative writes (see §6.2). Specifically, we employ the following proof strategy: (see §6.2).

(1) We prove that x86\textsubscript{view} and x86\textsubscript{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\textsubscript{prom} and x86\textsubscript{axiom} are equivalent by adapting the analogous equivalence proof for Armv8 concurrency [44] as x86\textsubscript{prom} and x86\textsubscript{axiom} respectively have the same style as the (view-based) Armv8\textsubscript{view} and (axiomatic) Armv8\textsubscript{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\textsubscript{prom} iff it is allowed under x86\textsubscript{view}.

Theorem 5.2. A behavior is allowed under x86\textsubscript{view} iff it is allowed under x86\textsubscript{axiom}.

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

Theorem 5.3. A behavior is allowed under Px86\textsubscript{axiom} iff it is allowed under Px86\textsubscript{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; 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
Armv8 allows an execution where speculative execution of writes, interacting with NVMFlush: 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:

\[
\begin{align*}
  (a) & \ r_1 := y \quad / / = 1 \\
  (b) & \ x := 1 \\
  (c) & \ r_2 := x \quad / / = 1 \\
  (d) & \ y := 1 \\
\end{align*}
\]

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 PArmv8view: View-Based Armv8 Persistency

As with Px86view, the view-based Armv8 persistency model, PArmv8view, follows the same interleaving model over the history of stores. However, PArmv8view differs from Px86view 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 PArmv8view thread state in [9, Fig. 15] has additional view components compared to x86view in Fig. 4. These additional components are those of Armv8view [44]; i.e., the PArmv8view thread state is that of Armv8view extended with persistency views (\(v_{p\text{Ready}}, v_{p\text{Async}}\) and \(v_{p\text{Commit}}\) 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, Armv8view [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 \(ts.prom\) 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 \(x := 1\) \(\rightarrow @1\) with \(ts_1.prom = \{ @1 \}\); (c) \(tid_2\) reads \(x := 1\) \(\rightarrow @1\), updating \(ts_2.coh[x]\) and \(ts_2.v_{\text{Old}}\) (‘old-write view’) to \(\Rightarrow @1\); (d) \(tid_2\) writes \(y := 1\) \(\rightarrow @2\), updating \(ts_2.coh[y]\) and \(ts_2.v_{\text{Old}}\) (‘old-write view’) to \(\Rightarrow @2\); (a) \(tid_1\) reads \(y := 1\) \(\rightarrow @2\), updating \(ts_1.coh[y]\) and \(ts_1.v_{\text{Old}}\) to \(\Rightarrow @2\); and (b-fulfill) \(tid_1\) fulfills \(x := 1\) \(\rightarrow @1\), yielding \(ts_1.prom = \emptyset\) and \(ts_1.v_{\text{Old}} = \emptyset\). 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 PArmv8view 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 \(\langle data := 42\rangle @1\) which is yet unfulfilled, the right thread reads \(\langle data := 42\rangle @1\) and writes \(\langle commit := 1\rangle @2\), and then a crash occurs. If upon recovery \(\langle commit := 1\rangle @2\) has persisted, then \(\langle data := 42\rangle @1\) (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 \(\langle data := 23\rangle @1\) 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 \(\langle data := 42\rangle @1\) or \(\langle data := 23\rangle @1\) 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 PArmv8view and PArmv8axiom are equivalent (see §6.4).

6.3 PArmv8axiom: Fixing and Simplifying PArmv8

We use the model of Pulte et al. [44] as the baseline axiomatic model for Armv8 concurrency, presented as Armv8axiom in [44, Appendix D].15 The Armv8axiom model is equivalent to the authoritative axiomatic model in [43] which is reviewed by Arm engineers. Note that Armv8axiom has the same style as x86axiom in Fig. 7, except that: (1) all coherence constraints are captured by a single axiom (INTERNAL) since (co-ww)

\[\text{15While reads update } v_{\text{New}} \text{ in x86view, they update } v_{\text{Old}} \text{ in Armv8view. We refer the reader to [9, Appendix B] for more details.}\]

\[\text{16We refactor the relations in [44] to replace } dmb \text{ with } dmb \cup dsb. \text{The latter is a straightforward extension as } dsb \text{ is strictly stronger than } dmb [4].}\]
We develop a stateless model checker for PArmv8 view and (3) Armv8 example [45] that emulates a persistent AtomicPersists behavior is allowed under PArmv8 iff it is equivalent by generalizing the analogous concurrency result in [44]. The proof is mechanized in [10].

7.1 Model Checking Tool

Model Checking Tool for Armv8_view We first briefly review the baseline model checking tool for Armv8_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_view model: a reachable state in Armv8_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

We extend the model checking tool for Armv8_view as follows: (1) we add persistency instructions to the executable model for sequential semantics in Intel-x86; (2) we add persistency views to the executable memory model for Armv8_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_view is similar to that for Armv8_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_max (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

---

17 We could replace internal with irreflexivity of po: (co ∪ rf ∪ fr ∪ fr; rf) for uniformity with x86_axiom. We forwent this to use Armv8_axiom [44] as is.
selectively persisting cache lines (f\texttt{lush} and f\texttt{lushopt}), 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 PTS\texttt{soyn} model for Intel-x86 that satisfies the three properties of (A)–(C) discussed in §1. In particular, PTS\texttt{soyn} supports persistent synchronization patterns even in the presence of I/O (B) as it also models f\texttt{lush} instructions synchronously like Px\texttt{86view} (§3.5). However, this problem of asynchronous modeling of f\texttt{lush} 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 c1f\texttt{lush}/c1f\texttt{lushopt}). Nevertheless, we believe that our contributions still stand for the following reasons. First, to ensure backwards compatibility, programs must support persistence in the absence of eADR. That is, a correct NVM program must defensively check whether eADR is enabled, and if not insert appropriate c1f\texttt{lush} or c1f\texttt{lushopt} 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., f\texttt{lush}). 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\texttt{man} (TSO). Recently, Raad et al. [45] developed a persistent program logic for verifying programs under the Px\texttt{86} 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


