Extending Intel-x86 Consistency and Persistency

Formalising the Semantics of Intel-x86 Memory Types and Non-Temporal Stores

AZALEA RAAD, Imperial College London, United Kingdom
LUC MARANGET, Inria, France
VIKTOR VAFEIADIS, MPI-SWS, Germany

Existing semantic formalisations of the Intel-x86 architecture cover only a small fragment of its available features that are relevant for the *consistency* semantics of multi-threaded programs as well as the *persistency* semantics of programs interfacing with non-volatile memory.

We extend these formalisations to cover: (1) non-temporal writes, which provide higher performance and are used to ensure that updates are flushed to memory; (2) reads and writes to other Intel-x86 memory types, namely uncacheable, write-combined, and write-through; as well as (3) the interaction between these features. We develop our formal model in both operational and declarative styles, and prove that the two characterisations are equivalent. We have empirically validated our formalisation of the consistency semantics of these additional features and their subtle interactions by extensive testing on different Intel-x86 implementations.

CCS Concepts: • **Theory of computation → Semantics and reasoning**; **Concurrency**; **Axiomatic semantics**; **Operational semantics**; • **Hardware → Hardware validation**.

Additional Key Words and Phrases: weak memory, memory consistency, memory persistency, non-volatile memory, Intel-x86, non-temporal accesses, memory types, cacheability

1 INTRODUCTION

Since the seminal work of Sewell et al. [2010], it is widely understood in the programming language literature that the Intel-x86 architecture follows the *TSO ('total store order')* memory consistency model. This, in particular, means that the annotated weakly consistent behaviour of the SB ('store buffering') program below is allowed on Intel-x86, whereas that of MP ('message passing') is not.[1]

$$
\begin{array}{ll}
\left.\begin{array}{l|l}
x := 1; & y := 1; \\
\mathsf{a} := y;\ \textit{// 0} & \mathsf{b} := x;\ \textit{// 0}
\end{array}\right. \text{(SB)} &
\left.\begin{array}{l|l}
x := 1; & \mathsf{a} := y;\ \textit{// 1} \\
y := 1; & \mathsf{b} := x;\ \textit{// 0}
\end{array}\right. \text{(MP)}
\end{array}
$$

This, however, constitutes a very shallow understanding of the consistency semantics of the Intel-x86 architecture. The outcomes of these litmus tests crucially depend on the *memory type* attribute of the pages containing x and y, which is set by the operating system and recorded

---

[1]In all our examples we use x, y, z for (shared) memory locations initialised with 0, and use a, b, c for (local) registers. For readability, rather than Intel-x86 assembly instructions, we use the assignment notation and write e.g. $x := v$ for writing v to x, and a := x to denote reading the value of x into a. The *// v* annotation after a read denotes that the value read is v.

Authors' addresses: Azalea Raad, Imperial College London, United Kingdom, azalea@imperial.ac.uk; Luc Maranget, Inria, France, luc.maranget@inria.fr; Viktor Vafeiadis, MPI-SWS, Germany, viktor@mpi-sws.org.

---

in the page table. The most common memory type is *write-back* (wb) memory, whose semantics corresponds to TSO, as prior work has discovered [Sewell et al. 2010]. However, Intel-x86 supports other memory types such as *uncacheable* (uc) memory, which is mainly used for memory-mapped I/O; *write-combining* (wc) memory, which is meant for bulk data transfer such as displaying videos; and *write-through* (wt) memory. All these memory types have very different semantics. For example, if x and y were in wc memory, then the weak behaviour of sb would *not* be allowed, whereas that of mp would be! To restrict the additional weak behaviours of wc memory in programs such as mp, Intel-x86 provides an additional fence instruction, **sfence** (store fence), which is strictly weaker than a memory fence (**mfence**) in that it only orders store instructions. (Existing formalizations of Intel-x86 treat **sfence** simply as a NOP, since stores are already ordered under TSO.)

We note that such weaker-than-TSO effects are not observable only under these specialised memory types. Even when using wb memory, it is possible to get similar effects with *non-temporal writes* (MOVNT). Non-temporal writes provide higher performance than regular writes in cases where the written data is not expected to be used immediately, by reducing cache contention and giving more flexibility to the hardware to reorder them with respect to other accesses. Specifically, the Intel manual (see (NT) in §2) explains non-temporal writes in terms of wc memory: a non-temporal write on x behaves as if x were in wc memory. The reality, however, is more complex in that when using non-temporal writes both sb and mp can exhibit their annotated weak behaviours!

Naturally, these different memory types can be used together in a program, leading to subtle interactions between them and further complicating their semantics. The semantics of such interactions is barely discussed in the Intel manuals, creating confusion even amongst seasoned programmers on these topics – as witnessed by contradicting answers to a relevant query on StackOverflow [Anonymous 2021] – and highlighting the need for their *formal, empirically-validated semantics*.

A closely related aspect of architectural semantics is memory *persistency*, which describes when and how memory stores are propagated to memory, and thus may be seen by an external device. Memory persistency is hugely important for systems with *non-volatile memory* (NVM) because it determines the possible contents of memory after a power failure. As with consistency, prior work [Raad et al. 2020] has only formalised the semantics of regular stores to wb memory and basic cache-line flushing instructions. This subset of features is quite limiting for practical purposes: other memory types and non-temporal stores are often used for better performance and/or simpler persistency semantics. In particular, the PMDK library for persistent programming [Intel 2015] (a large-scale open-source project) uses non-temporal writes to avoid cache-line flushes.

We address the shortcomings of the existing formal Intel-x86 consistency and persistency models by extending them to cover non-temporal writes and the wide range of memory types available. To our knowledge, we have developed the *first* formal semantic models of these architectural features, which we integrate into the existing x86-TSO [Sewell et al. 2010] and Px86$_{sim}$ [Raad et al. 2020] models. Specifically, we develop two formal models – an *operational* one in terms of a machine with a collection of buffers and a *declarative* one in terms of execution graphs – and prove their *equivalence*. Having two equivalent models is useful not only for ensuring the canonicity of the formalism, but also because one or the other formulation may be more useful for establishing different results. For instance, operational models are better suited for underpinning program logics and checking reachability of an erroneous configuration and/or robustness for finite-state programs with loops (e.g. [Abdulla et al. 2021; Bouajjani et al. 2013; Lahav and Boker 2020]), whereas declarative models are better suited for deriving stateless model checking of programs with only bounded loops (e.g. [Kokologiannakis et al. 2021, 2019b]).

We have developed our formal models through a careful reading of the Intel manuals, informal discussions/exchanges with Intel engineers, and, in the case of consistency, *empirical validation* by extensive testing of multiple Intel-x86 implementations. As setting the memory type of a page is

only possible in the kernel, the latter involved adapting the **diy** toolsuite [Alglave and Maranget 2021] to run litmus tests in kernel mode, allocate each variable on a different page, and set the page memory types accordingly. By contrast, as in previous work we could not test our persistency semantics since it requires specialised hardware to monitor the memory bus traffic (see §2).

Prevalence of Intel-x86 Non-Temporal Writes and Memory Types. Non-temporal writes provide an *application-level* mechanism for enforcing wc cacheability and avoiding cache pollution. Non-temporal writes are ubiquitous. For instance, searching for MOVNTI (an Intel-x86 non-temporal write) on GitHub returns over 300K results [GitHub 2021], spread across over ten languages, including C, C++ and Assembly. Searching for other non-temporal writes such as MOVNTQ, MOVNTDQ, MOVNTPS and MOVNTPD return similar results. Moreover, non-temporal writes are available in languages such as Rust [Rust 2021], which are in turn compiled to non-temporal writes on Intel-x86 machines. Another notable use of non-temporal writes is in the memset function in the C runtime [LWN 2007]. Similarly, non-temporal writes are used by memcpy in glibc (the core libraries for GNU/Linux) [Free Software Foundation 2016]. Consequently, a large body of existing code using glibc uses non-temporal writes by extension. As with PMDK [Intel 2015], other large-scale projects such as SPDK [SPDK 2021], DPDK [DPDK 2021] and DML [DML 2021] use non-temporal writes; e.g. DPDK and DML use MOVDIR64b (a non-temporal write) to atomically communicate with accelerators, while SPDK uses non-temporal writes to interface with NVM [GitHub 2019].

As we describe in §2, the Intel-x86 memory types are declared either through the page attribute table (PAT) or custom registers, and are thus used within *system-level* code, e.g. the Linux kernel [LWN 2008]. Moreover, wc memory is used inside the Linux kernel for e.g. frame buffer optimisation [LWN 2016]. Similarly, the Linux kernel uses uc memory for memory-mapped I/O (MMIO) [LWN 2016]. Finally, non-wb memory types are typically used to interact with non-cache-coherent DMA (direct memory access) device drivers.

Contributions and Outline. Our contributions (detailed in §2) are as follows:

§3 We develop the declarative Ex86 model as the first formal model of Intel-x86 consistency that accounts for memory types and non-temporal stores.

§4 We develop an operational Ex86 model, which we prove equivalent to our declarative model.

§5 We describe how we empirically validated Ex86 through extensive litmus testing.

§6 We extend Ex86 to develop declarative and operational characterisations of the PEx86 model as the first formal model of Intel-x86 persistency that accounts for memory types and non-temporal writes, and show that the two characterisations of PEx86 are equivalent.

We discuss related and future work in §7.

Additional Material. The proofs of all theorems stated in the paper are given in the accompanying technical appendix [Raad et al. 2022a]. Our library of litmus tests in our validation effort is available online [Raad et al. 2022b].

2 OVERVIEW

Memory *consistency* models describe the permitted behaviours of programs by constraining the *volatile memory order*, i.e. the order in which memory instructions (e.g. writes) are made visible to other threads. Analogously, memory *persistency* models describe the permitted behaviours of programs upon recovering from a crash (e.g. due to power loss) in the context of the NVM technology by defining a *persistent memory order*, i.e. the order in which the effects of memory instructions are committed to persistent memory. To distinguish between the two memory orders, memory *stores* are differentiated from memory *persists*. The former denotes the process of making an instruction (e.g. a write) visible to other threads, whilst the latter denotes the process of committing

instruction effects durably to persistent memory. We proceed with an intuitive account of our *Ex86*
('extended x86') model, the first formal Intel-x86 consistency semantics that covers memory types
and non-temporal writes (§2.1). We then present an overview of our *PEx86* ('persistent Ex86') model
as the first formal account of Intel-x86 persistency semantics that extends to memory types and
non-temporal writes. In what follows we often cite the Intel reference manual [Intel 2021] in "double
quotation marks", at times including [text in square brackets] denoting our added clarification.

2.1 Ex86: The Extended Intel-x86 Consistency Model

Validating Ex86. We have empirically validated our Ex86 consistency model (described shortly
below), including all behaviours and examples discussed in this paper. Specifically, using the **diy**
toolsuite [Alglave and Maranget 2021], we have built a vast library of *litmus tests* and ran them
against Ex86 and as kernel modules. Litmus testing involves running small, concurrent programs
(e.g. sʙ in §1) that exercise specific features of consistency on a machine while simultaneously
stressing the memory with heavy background traffic. With thousands of tests and hundreds of
thousands of runs per test, one can observe all possible behaviours on the machine, ensuring
behaviour coverage. A full catalogue of our tests and their results on various platforms is given in
[Raad et al. 2022b]. As we discuss below, we have validated all Ex86 features and instructions except
one instruction, **flush**$_{opt}$, which constitutes a small fragment of Ex86. This is because on most
modern hardware, including all machines available to us for testing, the **flush**$_{opt}$ implementation
is commonly stronger than its specification in the Intel manual [Intel 2021]. Nevertheless, we have
generated extensive tests that could be used to validate the **flush**$_{opt}$ behaviour in the future on
machines that implement it more faithfully.

Intel-x86 Memory Types at a Glance. Intel-x86 processors allow any area of system memory
to be cached. Moreover, for each individual page or region of memory, the *caching type*, also called
the *memory type* may be specified [Intel 2021, Vol. 3A, §11.3] either through the *page attribute table*
(PAT) or *memory type range registers* (MTRRs). Specifically, the memory type can be defined as
strong-uncacheable (uc), *uncacheable* (uc⁻), *(uncacheable) write-combining* (wc), *(cacheable) write-*
back (wb), *(cacheable) write-through* (wt), and *write protected* (wp). Although the memory type of a
location may be altered during execution, doing so entails a complex mechanism invoking kernel
code. As such, for simplicity we assume that memory types do not change once assigned, and thus
the sets of locations associated with memory types are pairwise disjoint. As we describe below, all
behaviours described below are *empirically validated* through extensive litmus testing [Raad et al.
2022b]. As we describe in §5, we could not support the wp memory in our validation infrastructure.
Hence, to avoid speculating about the behaviour of wp memory, we forgo wp in our Ex86 formalism.

As we elaborate shortly, the uc type is the strongest of all types and its accesses follow *strong*
ordering (Intel terminology). Specifically, accesses to uc memory cannot be reordered with respect
to any other accesses except later (in program order) reads on wb and wt memory. As such, when all
locations accessed in a program P lie in uc memory, no reordering is allowed and the P behaviours
are those observed under SC ('sequential consistency') [Lamport 1979] – see Theorem 1.

(uc) "System memory locations are not cached. All reads and writes appear on the system bus
and are executed in program order without reordering. ... [uc] is useful for memory-mapped
I/O devices."   [Intel 2021, Vol. 3A, p. 11-6]

Note that the interaction between uc and other types (e.g. reordering later wb/wt reads before uc
writes) is not discussed in the Intel manual and we have uncovered it through our experiments.

As with uc, accesses to uc⁻ memory follow the strong ordering (and thus follow the same ordering
constraints) and they only differ in how they are selected (assigned). As accesses on uc and uc⁻
memory are subject to the same ordering constraints, they exhibit the same behaviours and follow

the same semantics. As such, in our Ex86 formalism we forgo the uc⁻ memory and only model uc. Nevertheless, all behaviours and semantics we ascribe to uc memory also hold of uc⁻ memory.

(uc⁻) "[uc⁻] Has same characteristics as uc memory type, except that uc⁻ type can be overridden by programming the MTRRs for the wc memory type." [Intel 2021, Vol. 3A, p. 11-6]

The wc memory type is the weakest of all types in that its accesses follow the *weak ordering* (Intel terminology). A main characteristic of wc memory is that it allows write-write reordering: write accesses on different wc locations may be reordered, leading to weak behaviours disallowed under both SC (governing uc memory) and TSO (governing wb memory – see below). Specifically, as we show later in Theorem 1, when all accesses in a program P are on wc memory, then the behaviours of P are those observed under a strengthening of the PSO ('partial store order') model we refer to as SPSO ('strong PSO'), where only write-write reordering on different locations is allowed.

(wc) "System memory locations are not cached (as with uc). ... [wc] is appropriate for video frame buffers, where the order of writes is unimportant as long as the writes update memory so they can be seen on the graphics display." [Intel 2021, Vol. 3A, p. 11-7]

Once again, the Intel manual does not discuss the subtle interaction between wc and other memory types. For instance, as we discuss below, our tests revealed that wc writes can be reordered with respect to writes on different locations in wc/wb memory but not those in uc/wt memory.

The wb memory type provides the best performance for the typical memory access patterns of applications. When all locations in a program P lie in wb memory, the behaviours of P are those allowed under *processor ordering* (Intel terminology), also referred to as TSO [Sewell et al. 2010] – see Theorem 1. The main characteristic of TSO is that it allows write-read reordering: later reads can be reordered before earlier writes. Note that wb memory is strictly weaker than uc in terms of ordering constraints: while write-read reordering is allowed on wb, all reorderings (including write-read) are prohibited on uc. By contrast, wb and wc are incomparable: write-read reordering is allowed on wb but not on wc, write-write reordering is allowed on wc but not on wb.

(wb) "Writes and reads to and from system memory are cached ... writes are performed entirely in the cache ... Writes to a cache line are not immediately forwarded to system memory; instead, they are accumulated in the cache. The modified cache lines are written to system memory later ... [wb] provides the best performance ... " [Intel 2021, Vol. 3A, p. 11-7]

The wt memory lies between wb and uc in terms of its ordering constraints. Specifically, wt reads follow the same constraints as wb reads in that write-read reordering on wt memory is allowed. The main difference between wt and wb lies in how their writes interact with wc writes. More concretely, as we describe shortly, wb writes may be reordered with respect to wc writes. By contrast, unlike wb and as with uc memory, wt writes cannot be reordered with respect to wc writes.

(wt) "Writes and reads to and from system memory are cached. ... All writes are written to a cache line (when possible) and through to system memory. ... [wt] is appropriate for frame buffers ... " [Intel 2021, Vol. 3A, p. 11-7]

Note that as well as enforcing certain ordering constraints, memory types also prescribe memory *cacheability*. Specifically, locations in uc and wc memory are not cached (see (uc) and (wc) above); their accesses bypass the cache and directly interact with the memory. Hereafter, we refer to uc and wc memory collectively as *non-cacheable* (nc). By contrast, the locations in both wb and wt memory may be cached. As such, henceforth we refer to wb and wt memory collectively as *cacheable* (c). However, while wb writes are cached and propagated to memory only *later*, wt writes are cached and propagated to memory *immediately* (see (wb) and (wt) above).

Intel-x86 Non-Temporal Accesses*.* The Intel reference manual categorises the data referenced by a program as either *temporal* (data that will be reused, e.g. program code), or *non-temporal* (data

that will be referenced once and not reused in the immediate future, e.g. multi-media data, such as the display list in a 3-D graphics application). Ideally, to use processor caches efficiently, temporal data should be cached, whereas non-temporal data should not be cached. Filling processor caches with non-temporal data is referred to as "polluting the caches". To minimise cache pollution, Intel-x86 architectures provide several *non-temporal store instructions* that treat the memory accessed as wc memory, storing data to memory directly and bypassing the caches. More concretely, if a program executes a non-temporal store on location x, *and* x lies in wb, wt or wc memory, then the store on x is written to memory with wc semantics. If, however, x lies in uc memory, then the non-temporal hint is ignored and the store follows uc semantics. Through our validation effort we have confirmed that non-temporal stores (on wb/wt/wc memory) indeed follow wc semantics, as intended.

(NT)   "If a program specifies a non-temporal store and the memory type of the destination region is wb, wt or wc, the processor will do the following: if the memory location being written to is present in the cache hierarchy, the data in the caches is evicted; the non-temporal data is written to memory with wc semantics."   [Intel 2021, Vol. 1, p. 10-12].

In addition to several non-temporal *store* instructions, Intel-x86 architectures provide a single non-temporal *load* instruction. However, as our private correspondence with the lead architect of the Intel instruction set system architecture has revealed, the non-temporal load instruction has been a source of implementation issues, it has not been implemented consistently, and there has been ambiguity regarding its semantics. Moreover, we have been unable to validate the behaviour of non-temporal loads (see §5). As such, rather than speculating about its ambiguous and inconsistent semantics, in our Ex86 formalism we forgo the single non-temporal load instruction.

Litmus Test Notation. In what follows we elaborate on the behaviour of Ex86 memory types and how they interact with one another through several representative *litmus test* programs. Given a location x and memory type $t \in \{uc, wc, nc, wb, wt, c\}$, we write $x \in \text{Loc}_t$ to denote that x is in t memory. When the memory type of location x is immaterial and the exhibited behaviours hold regardless of the memory type of x, we forgo the type annotation and write $x \in \text{Loc}$.

In our programs we write $x := v$ to denote writing (storing) value v to (shared) memory location x, and write $a := x$ to denote reading (loading) the value of location x into the (thread-local) register a. Analogously, we write $x :=_{NT} v$ to denote storing v to location x with a non-temporal hint. As discussed above, the non-temporal hint is ignored if $x \notin \text{Loc}_{wb \cup wt \cup wc}$.

Ordering Constraints of **mfence,** **sfence** ***and Atomic Updates***. In order to afford more control over instruction reordering, Intel-x86 provides *fence instructions*, including *memory fences*, written **mfence**, and *store fences*, written **sfence**. Memory fences are strictly stronger than store fences: while memory fences cannot be reordered with respect to any memory instruction, store fences may be reordered with respect to only reads. Additionally, Intel-x86 provides instructions for *atomically updating* the memory through 'read-modify-write' (RMW) operations such as **CAS** ('compare-and-set') and **FAA** ('fetch-and-add'). RMW instructions follow the same ordering constraints as memory fences in that they cannot be reordered with respect to any memory instruction.

Ordering between Earlier Reads and Later Memory Instructions. Earlier read instructions cannot be reordered with respect to any later memory instruction (including later reads and writes). This is illustrated in the programs of Fig. 1a and Fig. 1b, showing examples of read-read and read-write reordering, respectively. More concretely, the program in Fig. 1a depicts a variant of the canonical 'message passing' (MP) litmus test with an **mfence** between the two writes of the left thread. This intervening **mfence** ensures that the two writes cannot be reordered (regardless of the memory types of x and y), and thus the two writes are executed in program order. Let us assume it were possible for the two reads of the right thread to be reordered. Then it would be possible to

$x, y \in \text{Loc}$	$x, y \in \text{Loc}$	$x \in \text{Loc}_c, y \in \text{Loc}$	$x \in \text{Loc}_{nc}, y \in \text{Loc}$	$x \in \text{Loc}_c \lor y \in \text{Loc}_c$
$x := 1;$ **mfence**; $y := 1;$ ‖ $a := y;$ $b := x;$	$x := 2;$ **mfence**; $y := 1;$ ‖ $a := y;$ $x := 1;$	$x := 1;$ **mfence**; $a := y;$ ‖ $y := 1;$ $b := x;$	$x := 1;$ **mfence**; $a := y;$ ‖ $y := 1;$ $b := x;$	$x := 1;$ **sfence**; $a := y;$ ‖ $y := 1;$ **sfence**; $b := x;$
$a=1 \land b=0$: ✗	$a=1 \land x=2$: ✗	$a=0 \land b=0$: ✓	$a=0 \land b=0$: ✗	$a=0 \land b=0$: ✓
(a) READ-READ	(b) READ-WRITE	(c) WRITE-READ	(d) WRITE-READ	(e) SFENCE-READ

$x, y \in \text{Loc}_{uc \cup wb \cup wt}$	$x, y \in \text{Loc}_{wc \cup wb},$ $x \in \text{Loc}_{wc} \lor y \in \text{Loc}_{wc}$	$x, y \in \text{Loc}_{wb \cup wt \cup wc}$	$x \in \text{Loc}_{wb \cup wt \cup wc},$ $y \in \text{Loc}_{wb \cup wc}$	$y \in \text{Loc}_{wb \cup wt \cup wc},$ $x \in \text{Loc}_{wb \cup wc}$
$x := 2;$ **mfence**; $y := 1;$ ‖ $y := 2;$ $x := 1;$	$x := 2;$ **mfence**; $y := 1;$ ‖ $y := 2;$ $x := 1;$	$x := 2;$ **mfence**; $y := 1;$ ‖ $y :=_{NT} 2;$ $x :=_{NT} 1;$	$x := 2;$ **mfence**; $y := 1;$ ‖ $y := 2;$ $x :=_{NT} 1;$	$x := 2;$ **mfence**; $y := 1;$ ‖ $y :=_{NT} 2;$ $x := 1;$
$x=2 \land y=2$: ✗	$x=2 \land y=2$: ✓	$x=2 \land y=2$: ✓	$x=2 \land y=2$: ✓	$x=2 \land y=2$: ✓
(f) WRITE-WRITE	(g) WRITE-WRITE	(h) WRITE-WRITE	(i) WRITE-WRITE	(j) WRITE-WRITE

Fig. 1. Ex86 Litmus tests illustrating possible reordering of read and write instructions and the resulting weak behaviours, where ✓ (resp. ✗) denotes that the depicted weak behaviour is (resp. is not) observed. The underlined subfigure captions are hyperlinks to the corresponding (class of) tests in our validation effort.

observe $a=1 \land b=0$: when $b := x$ is reordered before $a := y$, then this outcome is observed when all instructions of the left thread are executed between $b := x$ and $a := y$. However, as corroborated by our validation (see READ-READ), we ran each variant (with different memory types for x and y) of the Fig. 1a program 1.5 billion times, and never observed $a=1 \land b=0$. Analogously, the $a=1 \land x=2$ behaviour was never observed in any of the variants of the program in Fig. 1b (see READ-WRITE). We therefore conclude that read-read and read-write reordering are not allowed under Ex86, regardless of the memory types of the underlying locations.

***Earlier Reads and Later* sfence** ($\checkmark^\dagger$ *in Fig. 2*). As noted by Raad et al. [2020], although **sfence** instructions are not ordered with respect to reads, reordering an earlier read after a later **sfence** does not affect the program behaviour: as earlier reads are ordered with respect to all other later instructions, reordering them after a later **sfence** does not alter the program behaviour. That is, given a program $P \triangleq a := x; \textbf{sfence}; c$ with $c \neq \textbf{sfence}$, the $a := x$ read cannot be reordered after c, and reordering it after **sfence** alone does not affect the behaviour of P. As such, as in [Raad et al. 2020], we opt to order earlier reads and *all* later memory instructions, including later **sfence**.

Write-Write Reordering. As briefly discussed above (1) write-write reordering is allowed on wc memory; and (2) wb writes can be reordered (in both directions) with respect to wc writes. That is, write-write reordering is allowed when one write is on wc memory, and the other is on either wc or wb memory. This is illustrated in Fig. 1g: as before, the writes of the left thread cannot be reordered thanks to **mfence**; however, the two writes of the right thread may be reordered since $x, y \in \text{Loc}_{wc \cup wb}$ and $x \in \text{Loc}_{wc} \lor y \in \text{Loc}_{wc}$. As such, when all instructions of the left thread execute before $x := 1$ and after $y := 1$, then the $x=2 \land y=2$ weak behaviour shown can be observed, as confirmed by our experiments (see WRITE-WRITE). By contrast, write-write ordering is disallowed when neither write is on wc memory, as shown in Fig. 1f: as $x, y \in \text{Loc}_{uc \cup wb \cup wt}$, the writes of the right thread cannot be reordered and thus the weak behaviour $x=2 \land y=2$ cannot be observed.

Note that since non-temporal writes follow wc semantics, write-write reordering is also allowed when one write is non-temporal (on wb/wt/wc memory) and the other is either a non-temporal write or a write on wc or wb memory. This is illustrated in the examples of Figs. 1h to 1j.

Write-Read Reordering. As discussed above, write-read reordering is allowed on both wb and wt. Indeed, later reads on wb and wt memory (i.e. on c memory) can be reordered before *all* writes, regardless of their memory type. This is illustrated in Fig. 1c, depicting a variant of the 'store-buffering' (SB) litmus test with an additional **mfence**. As before, the instructions of the left thread cannot be reordered due to the intervening **mfence**. However, the $b := x$ read on c memory ($x \in \text{Loc}_c$) can be reordered before the earlier $y := 1$ write regardless of the memory type of y ($y \in \text{Loc}$). As such, when all instructions of the left thread execute between $b := x$ and $y := 1$, then the a=0 ∧ b=0 behaviour shown can be observed, as confirmed by our experiments (see WRITE-READ).

By contrast, the reads on non-cacheable (nc) memory cannot be reordered before any earlier writes, as shown in Fig. 1d. That is, unlike in Fig. 1c, the $b := x$ read on nc memory ($x \in \text{Loc}_{nc}$) cannot be reordered before $y := 1$ ($y \in \text{Loc}$), and thus the a=0 ∧ b=0 weak behaviour shown is not observed, as corroborated by our experiments (see WRITE-READ).

Finally, as non-temporal writes follow wc semantics, write-read reordering is also allowed (resp. disallowed) for a non-temporal write and a read on c (resp. nc) memory. That is, if we replace $y := 1$ in Figs. 1c and 1d with $y :=_{NT} 1$ ($y \in \text{Loc}_{wb \cup wt \cup wc}$), then the outcomes shown will remain unchanged.

Earlier **sfence** *and Later Non-Cacheable Reads* (✓‡ *in* Fig. 2). Although **sfence** instructions are not ordered with respect to reads, reordering a later nc read before an earlier **sfence** does not affect the program behaviour: as later nc reads are ordered with respect to all other earlier instructions under Ex86, reordering them before **sfence** does not alter the program behaviour. That is, given a program $P \triangleq c; \textbf{sfence}; a := x$ with $c \neq \textbf{sfence}$ and $x \in \text{Loc}_{nc}$, the $a := x$ read cannot be reordered before c, and reordering it before **sfence** alone does not affect the P behaviour. As such, for simplicity we opt to order *all* earlier instructions (including earlier **sfence**) and later nc reads.

By contrast, reordering a later c read before an earlier **sfence** can be observed because later c reads can be reordered before earlier writes. An example of this is illustrated in Fig. 1e, depicting a variant of SB with **sfence** instructions: without loss of generality, when $x \in \text{Loc}_c$, then the $b := x$ read can be reordered before both **sfence** and $y := 1$, allowing us to observe a=0 ∧ b=0.

Cache Line Instructions. As discussed above, a location x in wb memory may be cached in that a store to x does not immediately reach the memory; rather, for better performance the store may be cached and written back (evicted) to memory at a later time (e.g. when the cache is full). In order to afford more control over when stores reach the memory, Intel-x86 architectures provide three *persist instructions* (described at length in §2.2 below) for writing back a given cache line to memory: **flush** x, **flush**$_{\text{opt}}$ x and **wb** x.[2] More concretely, given a location x in cache line (set of locations) X, written $x \in X$, executing **flush** x, **flush**$_{\text{opt}}$ x or **wb** x writes back the X cache line to memory. As described in [Intel 2021] and formalised by Raad et al. [2020], persist instructions vary in strength (their constraints on instruction reordering) and performance: **flush** is the strongest of the three (enforcing additional ordering constraints), while **flush**$_{\text{opt}}$ and **wb** are equally weak with **wb** offering better performance than **flush**$_{\text{opt}}$. That is, **flush**$_{\text{opt}}$ and **wb** have the same specification and exhibit equivalent behaviour; as such, as in [Raad et al. 2020], we only model **flush** and **flush**$_{\text{opt}}$. Nevertheless, all behaviours and specifications ascribed to **flush**$_{\text{opt}}$ also hold of **wb**.

As mentioned above, **flush** instructions enforce strong ordering constraints with respect to writes and other **flush** in that they are ordered with respect to (1) *all* earlier and later **flush**; and (2) *all* earlier and later (non-temporal) writes regardless of their memory type. As such, **flush**

---

[2]In [Intel 2021], **flush** is referred to as CLFLUSH, **flush**$_{\text{opt}}$ is referred to as CLFLUSHOPT and **wb** is referred to as CLWB.

behaves as an **sfence** when inserted between two writes that could otherwise be reordered. For instance, were we to insert **flush** *z* between the two stores of the right threads in Figs. 1g to 1j, then they could no longer be reordered and thus the weak behaviour $x=2 \wedge y=2$ could not be observed. Our tests in our validation effort have confirmed the strong ordering enforced by **flush** (see HERE).

By contrast, **flush**$_{\text{opt}}$ instructions are only ordered with respect to *earlier* writes on the *same cache line*. As such, unlike **flush**, they cannot serve as an **sfence** as they can be reordered after later writes. Although we ran extensive tests on several machines to validate the behaviour of **flush**$_{\text{opt}}$, we could not observe the weak behaviours enabled by **flush**$_{\text{opt}}$ except in a few cases. This is because the **flush**$_{\text{opt}}$ implementation is commonly stronger than its specification (in the Intel manual), as was the case on all machines available to us for testing. Nevertheless, as our validation endeavour confirmed that the manual faithfully captures the behaviour of **flush**, we chose to 'trust' the weak behaviour of **flush**$_{\text{opt}}$ as specified in the manual and reflected it in our Ex86 formalism, albeit without validating it. However, note that **flush**$_{\text{opt}}$ instructions constitute a small fragment of the Ex86 language and are the only instruction in our Ex86 model that we could not validate. As such, we argue that Ex86 is fully validated up to implementation fidelity.

Ordering between* flush *and* flush$_{opt}$ ($\textcolor{red}{✗}^*$ *in Fig. 2). An earlier version of the Intel reference manual [Intel 2019] stated that **flush** and **flush**$_{\text{opt}}$ on the *same cache line* are ordered with respect to one another, as later reflected in the Px86$_{\text{sim}}$ model of Raad et al. [2020]. This older version of the manual has since been removed from the Intel pages and is currently only available through the Internet Archive [Intel 2019]. However, the latest version of the manual [Intel 2021] has weakened this constraint, guaranteeing no ordering between **flush** and **flush**$_{\text{opt}}$. As such, we have accordingly left out this constraint from our Ex86 model, thus minorly diverging from Px86$_{\text{sim}}$.

Putting it All Together: Ex86 Ordering Constraints. Fig. 2 presents a summary of ordering constraints between earlier (in program order) instructions (rows) and later instructions (columns) under Ex86, where R_{t} and W_{t} denote reads and writes on t memory, respectively; *MF*, *SF* and *U* denote **mfence**, **sfence** and RMW (update) instructions, respectively; *FL* and *FO* denote **flush** and **flush**$_{\text{opt}}$ instructions, respectively; and *NTW* denote *non-temporal* writes. If two instructions are ordered (denoted by $\textcolor{green}{✓}$), they cannot be reordered and thus their program order (i.e. the order in which they appear in the program) and store order (the order in which they are made visible to other threads) always agree. Conversely, if two instructions are unordered and thus can be reordered (denoted by $\textcolor{red}{✗}$), then their program and store orders may disagree. The sloc (resp. scl) entries denote that two instructions are ordered if and only if they access the same location (resp. cache line). The white (not highlighted) cells in the table correspond to the Px86$_{\text{sim}}$ model by Raad et al. [2020]. We develop Ex86 by extending Px86$_{\text{sim}}$ with Intel-x86 memory types and non-temporal stores. The ordering constraints on the Ex86 extensions are denoted by the highlighted cells. All permitted reorderings (i.e. non-$\textcolor{green}{✓}$ entries), except those of **flush**$_{\text{opt}}$ (i.e. in the *FO* row or column), are annotated with links to our validation tests that *witness* the reordering as a weak behaviour.

2.2 PEx86: The Persistent Ex86 Model

We next present PEx86, our extension of Ex86 that additionally accounts for the persistency semantics of Intel-x86 memory types and non-temporal writes in the presence of non-volatile memory. Persistency models are typically categorised along two axes: (1) strict or relaxed persistency; and (2) unbuffered or buffered persistency. Under *strict persistency*, instruction effects persist to memory in the order they become visible to other threads, i.e. the volatile and persistent memory orders coincide. By contrast, relaxed persistency allows for volatile and persistent memory orders to disagree. The second categorisation describes *when* persists occur. Under *unbuffered persistency*, persists occur *synchronously*: instruction effects are immediately committed to persistent memory

Earlier in Program Order

Later in Program Order

		1 R_{wb}	2 R_{wt}	3 R_{uc}	4 R_{wc}	5 W_{wb}	6 W_{wt}	7 W_{uc}	8 W_{wc}	9 NTW	10 U	11 MF	12 SF	13 FL	14 FO
A	R_{wb}	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓†	✓	✓
B	R_{wt}	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓†	✓	✓
C	R_{uc}	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓†	✓	✓
D	R_{wc}	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓†	✓	✓
E	W_{wb}	✗ 1, 2	✗ 3, 4	✓	✓	✓	✓	✓	✗ 5, 6, 7, 8	sloc 9, 10, 11, 12, 13	✓	✓	✓	✓	scl
F	W_{wt}	✗ 14, 15	✗ 16, 17	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
G	W_{uc}	✗ 18, 19	✗ 20, 21	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
H	W_{wc}	✗ 22, 23	✗ 24, 25	✓	✓	✗ 26, 27 28, 29	✓	✓	sloc 30, 31, 32 33, 34	sloc 35, 36, 37 38, 39, 40	✓	✓	✓	✓	scl
I	NTW	✗ 41, 42, 43, 44	✗ 45, 46, 47, 48	✓	✓	sloc 49, 50, 51 52, 53, 54	✓	✓	sloc 55, 56 57 58, 59	sloc 60, 61, 62, 63 64, 65, 66, 67 68, 69, 70, 71	✓	✓	✓	✓	scl
J	U	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
K	MF	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
L	SF	✗ 72, 73, 74, 75 76, 77, 78, 79	✗ 80, 81, 82, 83 84, 85, 86, 87	✓‡	✓‡	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
M	FL	✗ 88	✗ 89	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗*
N	FO	✗	✗	✓	✓	✗	✓	✓	✗	✗	✓	✓	✓	✗*	✗

Fig. 2. The Ex86 ordering constraints where ✓ denotes that two instructions are ordered, ✗ denotes they are not ordered (and thus may be reordered), and sloc (resp. scl) denotes that they are ordered iff they are on the same location (resp. cache line); see pp. 7, 8 and 9 for explanations of ✓†, ✓‡ and ✗*, respectively. The highlighted cells denote the Ex86 extensions from Px86$_{sim}$ [Raad et al. 2020]. All non-✓ entries (but those of FO) are accompanied with links to litmus tests in our validation effort that witness the associated reordering.

upon execution; i.e. execution is stalled by persists. By contrast, *buffered persistency* allows memory persists to occur *asynchronously* [Condit et al. 2009], buffering memory persists in a queue to be committed to memory at a later time. This way, persists occur after their corresponding stores and as prescribed by the persistent memory order; however, execution may proceed ahead of persists. As such, upon crash recovery, only a *prefix* of the persistent memory order may have persisted.

The persistency semantics of wb memory has been previously formalised by Raad et al. [2020] and later refined in [Cho et al. 2021; Khyzha and Lahav 2021]. As we describe shortly, wb memory follows relaxed, buffered persistency. Specifically, recall that wb writes are cached and propagated (persisted) to memory at a later time, thus following buffered persistency. Moreover, for better performance, cached wb writes may be persisted to memory in a different order than that they were cached (made visible to to other threads), thus enabling relaxed persistency.

We develop PEx86 as the first formal Intel-x86 persistency semantics that accounts for wc, wt and uc memory and non-temporal writes. As we describe below, wc, wt and uc all follow strict, unbuffered persistency. In the case of nc (wc and uc) memory, this is due to their non-cacheability: nc writes bypass the caches and directly interact with memory. As such, nc writes reach (persist

$x, y \in \text{Loc}_{wb}$	$x, x', y \in \text{Loc}_{wb}$	$x, x', y \in \text{Loc}_{wb}$	$x, x', y \in \text{Loc}_{wb}$	$x, x', y \in \text{Loc}_{wb}$	$x \in \text{Loc}_{wb},$ $y \in \text{Loc}$
$x := 1;$ $y := 1$	$x := 1;$ **flush** $x';$ $y := 1$	$x := 1;$ **flush**$_{\text{opt}}$ $x';$ $y := 1$	$x := 1;$ **flush**$_{\text{opt}}$ $x';$ **FAA**$(y, 1)$	$x := 1;$ **flush**$_{\text{opt}}$ $x';$ **sfence**; $y := 1$	$\mathsf{C} \triangleq x := 1;$ $y := 1$
(a)	(b)	(c)	(d)	(e)	(f)
rec: $x, y \in \{0, 1\}$	rec: $y{=}1 \Rightarrow x{=}1$	rec: $x, y \in \{0, 1\}$	rec: $y{=}1 \Rightarrow x{=}1$	rec: $y{=}1 \Rightarrow x{=}1$	rec: $x, y \in \{0, 1\}$
$x, x' \in \text{Loc}_{wb},$ $y \in \text{Loc}$	$x \in \text{Loc}_{wc},$ $y \in \text{Loc}_{uc \cup wt}$	$x \in \text{Loc}_{wc},$ $y \in \text{Loc}_{wc \cup wb}$	$x \in \text{Loc}_{uc \cup wt}$ $y \in \text{Loc}$	$x \in \text{Loc}_{wb \cup wt \cup wc}$ $y \in \text{Loc}_{uc \cup wt}$	$x \in \text{Loc}_{wb \cup wt \cup wc}$ $y \in \text{Loc}_{wc \cup wb}$
$\mathsf{C_P} \triangleq x := 1;$ **persist** $x';$ $y := 1$	$x := 1;$ $y := 1$	$x := 1;$ $y := 1$	$x := 1;$ $y := 1$	$x := 1;$ $x :=_{\text{NT}} 2;$ $y := 1$	$x := 1;$ $x :=_{\text{NT}} 2;$ **sfence**; $y := 1$
(g)	(h)	(i)	(j)	(k)	(l)
rec: $y{=}1 \Rightarrow x{=}1$	rec: $y{=}1 \Rightarrow x{=}1$	rec: $x, y \in \{0, 1\}$	rec: $y{=}1 \Rightarrow x{=}1$	rec: $y{=}1 \Rightarrow x{=}2$	rec: $y{=}1 \Rightarrow x{=}2$

Fig. 3. Examples of PEx86 programs and possible values of x, y upon recovery; in all examples x, y are locations in persistent memory where $x{=}y{=}0$ initially, a is a (local) register, $x, x' \in X$ (x, x' are in cache line X), $y \notin X$, and persist x denotes either **flush** x or **flush**$_{\text{opt}}$ x; c_b, where c_b is either **mfence**, **sfence** or an atomic update (RMW). Replacing an **sfence** with **mfence** or an RMW yields the same result upon recovery.

to) memory synchronously, at which point they also become visible to other threads (i.e. the store and persist orders are one and the same), thus following strict, unbuffered persistency. Analogously, recall that wt writes are cached (made visible to other threads) and also written to memory immediately (i.e. persisting the writes synchronously and in the same order as they were made visible to other threads), therefore following strict, unbuffered persistency. Lastly, as non-temporal writes are subject to wc semantics, they also follow strict, unbuffered persistency.

We next describe PEx86 persistency through several examples in Figs. 3 and 4. The rec specification below each example denotes possible values observed in memory upon recovery from a crash at an *arbitrary* program point. That is, we make no assumptions of when the crash occurs; rather, we assume that a crash may occur at any point during the execution.

***Persistency of* wb *Memory*.** The relaxed, buffered persistency of wb is reflected in the example of Fig. 3a. Due to the buffered nature of persists, if a crash occurs during the execution of this program, at crash time either write may or may not have already persisted and thus $x, y \in \{0, 1\}$ upon recovery. Note that the relaxed nature of wb persistency admits somewhat surprising behaviours that are not possible during normal (non-crashing) executions. In particular, at no point during the normal execution of the program the $x{=}0$, $y{\neq}1$ behaviour is observable: the two wb writes cannot be reordered under Intel-x86. Nevertheless, in case of a crash it is possible to observe $y{=}1$, $x{=}0$ after recovery. This is due to relaxed persistency of wb: the store order, describing the order in which writes are made visible to other threads (x before y), is separate from the persist order, describing the order in which writes are persisted to memory (y before x). More concretely, two wb writes may be persisted (1) in any order, when they are on distinct locations; or (2) in the store order, when they are on the same location. That is, for each wb location, its store and persist orders coincide.

In order to afford more control over when pending writes are persisted, the persist instructions, namely **flush** x and **flush**$_{\text{opt}}$ x, can be used to persist the pending writes on all locations in the

cache line of x. The persist behaviour of **flush** is illustrated in Fig. 3b: when $x, x' \in X$ (i.e. x, x' are in cache line X), executing **flush** x' persists the earlier write on X (i.e. $x := 1$) to memory. As such, if a crash occurs during the execution of Fig. 3b and upon recovery $y=1$, then $x=1$. That is, if $y := 1$ has executed and persisted before the crash, then so must the earlier $x := 1$; **flush** x'. Note that this behaviour is guaranteed thanks to the ordering constraints on **flush** instructions. Specifically, as we discussed in §2.1, **flush** instructions are ordered with respect to all writes; as such, **flush** x' in Fig. 3b cannot be reordered with respect to either write, and thus upon recovery $y=1 \Rightarrow x=1$.

However, instruction reorderings mean that persist instructions may not execute at the intended program point and thus may not guarantee the intended persist ordering. For instance, recall that when $x' \in X$, **flush**$_{\text{opt}}$ x' is only ordered with respect to earlier writes on the same cache line X and thus may be reordered with respect to writes on locations in different cache lines, i.e. those not in X. This is illustrated in the example of Fig. 3c: since $y \notin X$, the **flush**$_{\text{opt}}$ x' in Fig. 3c is not ordered with respect to $y := 1$ and may be reordered after it. Therefore, if a crash occurs after $y := 1$ has executed and persisted but before **flush**$_{\text{opt}}$ x' has executed, then upon recovery it is also possible to observe $y=1, x=0$. That is, there is no guarantee that $x := 1$ persists before $y := 1$, *despite* the intervening **flush**$_{\text{opt}}$ x'. By contrast, recall that **flush**$_{\text{opt}}$ instructions are ordered with respect to (earlier and later) RMW, **sfence** and **mfence** instructions. As such, **flush**$_{\text{opt}}$ x' in Fig. 3d cannot be reordered after **FAA**$(y, 1)$ and thus $y=1 \Rightarrow x=1$ upon recovery. Similarly, **flush**$_{\text{opt}}$ x' in Fig. 3e cannot be reordered after **sfence** and once again $y=1 \Rightarrow x=1$ upon recovery.

In summary, given wb locations x, x' in the same cache line, to ensure that a write on x is persisted, one can use a *persist sequence* on x', written persist x', via either (1) **flush** x'; or (2) **flush**$_{\text{opt}}$ x'; c_b, where c_b is an **mfence**/**sfence**/RMW. Conceptually, one can think of **flush** as *synchronous* (it blocks until all pending writes in its cache line have persisted), and of **flush**$_{\text{opt}}$ as *asynchronous* (it may not persist the pending writes at the intended program point) *unless* followed by a barrier as in (2). This pattern is shown in Fig. 3g, where persist x' ensures that $x := 1$ on wb memory is persisted before executing $y := 1$, ensuring $y=1 \Rightarrow x=1$ on recovery. That is, regardless of the y memory type ($y \in \text{Loc}$), the persist x' sequence cannot be reordered after $y := 1$ (as **flush**, **mfence**, **sfence** and RMWs are ordered with respect to all writes), ensuring $y=1 \Rightarrow x=1$ on recovery. By contrast, without the persist sequence in Fig. 3f, the wb write $x := 1$ may not have yet persisted when executing $y := 1$ on arbitrary memory ($y \in \text{Loc}$), and thus if a crash occurs after $y := 1$ has executed and persisted but before $x := 1$ has persisted, then one may also observe $y=1, x=0$ on recovery.

***Persistency of* wc, uc *and* wt *Memory*.** As mentioned above, wc, uc and wt memory all follow strict, unbuffered persistency, as shown in Figs. 3h and 3j. In the case of Fig. 3h, as $x \in \text{Loc}_{\text{wc}}$, executing $x := 1$ directly writes to memory; moreover, as $y \in \text{Loc}_{\text{uc} \cup \text{wt}}$, the two writes cannot be reordered. As such if upon recovery $y := 1$ has executed and persisted ($y=1$), then so must the earlier $x := 1$ ($x=1$). Similarly, in the case of Fig. 3j, as $x \in \text{Loc}_{\text{uc} \cup \text{wt}}$, the two writes cannot be reordered and executing $x := 1$ directly writes to memory, and thus $y=1 \Rightarrow x=1$ on recovery.

Note that as wc writes can be reordered with respect to wb/wc writes, it may be executed (and thus persisted) out of order. This is illustrated in Fig. 3i: as $x \in \text{Loc}_{\text{wc}}$ and $y \in \text{Loc}_{\text{wc} \cup \text{wb}}$, the $x := 1$ can be reordered after $y := 1$. As such, if a crash occurs after $y := 1$ has executed and persisted but before $x := 1$ has executed, then it is also possible to observe $y=1, x=0$ upon recovery.

***Persistency of Non-Temporal Writes*.** As mentioned above, as with wc memory, non-temporal writes follow strict unbuffered persistency. For instance, were we to replace $x := 1$ in Figs. 3h and 3j with $x :=_{\text{NT}} 1$ such that $x \in \text{Loc}_{\text{wb} \cup \text{wt} \cup \text{wc}}$, then their recovery specifications would remain unchanged. Interestingly, a non-temporal store on $x \in \text{Loc}_{\text{wb}}$ additionally serves as a *persist instruction* in that it persists (evicts) the pending (in-cache) writes on x to memory – see (NT) above. This is illustrated in Fig. 3k when $x \in \text{Loc}_{\text{wb}}$. Specifically, (1) as $y \in \text{Loc}_{\text{uc} \cup \text{wt}}$ and the first two writes are on the

$x \in \mathrm{Loc_{wb}}$, $y \in \mathrm{Loc}$	$x \in \mathrm{Loc_{wb}}$, $y \in \mathrm{Loc}$	$x \in \mathrm{Loc_{wc}}$, $y \in \mathrm{Loc_{wc \cup wb}}$	$x \in \mathrm{Loc_{wc}}$, $y \in \mathrm{Loc_{uc \cup wt}}$	$x \in \mathrm{Loc_{uc \cup wt}}$, $y \in \mathrm{Loc_{wb}}$	$x \in \mathrm{Loc_{uc \cup wt}}$, $y \notin \mathrm{Loc_{wb}}$
$\mathrm{C_P} \parallel$ `a:= y;` `if (a = 1)` `z:= 1;`	$\mathrm{C} \parallel$ `a:= y;` `if (a = 1)` `z:= 1;`	$\mathrm{C} \parallel$ `a:= y;` `if (a = 1)` `z:= 1;`	$\mathrm{C} \parallel$ `a:= y;` `if (a = 1)` `z:= 1;`	$\mathrm{C} \parallel$ `a:= y;` `if (a = 1)` `z:= 1;`	$\mathrm{C} \parallel$ `a:= y;` `if (a = 1)` `z:= 1;`
(a)	(b)	(c)	(d)	(e)	(f)
rec: $\mathrm{x^P} \land \mathrm{Y} \land \mathrm{Y^P}$	rec: $\mathrm{x} \land \mathrm{Y} \land \mathrm{Y^P}$	rec: $\mathrm{x} \land \mathrm{Y} \land \mathrm{Y^P}$	rec: $\mathrm{x^P} \land \mathrm{Y^P}$	rec: $\mathrm{x^P} \land \mathrm{Y}$	rec: $\mathrm{x^P} \land \mathrm{Y^P}$

with

$$\mathrm{x} \triangleq z{=}1 \Rightarrow x \in \{0, 1\} \qquad \mathrm{x^P} \triangleq z{=}1 \Rightarrow x{=}1 \qquad \mathrm{Y} \triangleq z{=}1 \land y \in \mathrm{Loc_{wb}} \Rightarrow y \in \{0,1\} \qquad \mathrm{Y^P} \triangleq z{=}1 \land y \notin \mathrm{Loc_{wb}} \Rightarrow y{=}1$$

Fig. 4. Examples of concurrent PEx86 programs and possible values of x, y, z upon recovery, where C and $\mathrm{C_P}$ are as defined in Fig. 3; all examples use the same notation and conventions as in Fig. 3.

same location x, all three writes are ordered with respect to one another and thus no reordering is allowed; and (2) executing $x :=_{\mathrm{NT}} 2$ first ensures that the earlier $x := 1$ is persisted to memory, and then writes 2 to x in memory. As such, if upon recovery $y := 1$ has executed and persisted ($y{=}1$), then so must $x :=_{\mathrm{NT}} 2$ ($x{=}2$). Note that if y in Fig. 3k were instead in wc/wb memory, $y := 1$ could be reordered before both writes on x, and on recovery it would be possible to observe $x \in \{0, 1, 2\}$ even when $y{=}1$. To prevent this reordering, we can insert an **sfence** (or **mfence**/RMW) before $y := 1$, as illustrated in Fig. 3l, restoring $y{=}1 \Rightarrow x{=}2$.

Concurrent Persistency Examples. The examples discussed thus far all concern sequential programs and the persist orderings in the *same* thread. The example in Fig. 4 illustrates how persist orderings can be imposed on *different* threads. Note that $\mathrm{C_P}$ and C in the left threads of Figs. 4a to 4f are as defined in Fig. 3. Under PEx86 one can use *message-passing* between threads to enforce persist ordering. A message is passed from thread τ_1 to τ_2 when τ_2 reads a value written by τ_1.

For instance, if the right thread in Fig. 4a reads 1 from y (written by $\mathrm{C_P}$ in the left thread), then the left thread passes a message to the right thread. Message passing ensures that the instruction writing the message (e.g. $y := 1$) is executed (ordered) before the instruction reading it (e.g. a := y). As such, since $x := 1$; persist x' (in $\mathrm{C_P}$) is executed before $y := 1$ (as in Fig. 3g), $y := 1$ is executed before a := y, and $z := 1$ is executed after a := y when a=1, we know $x := 1$; persist x' is executed before $z := 1$. Consequently, if upon recovery $z{=}1$ (i.e. $z := 1$ has executed and persisted before the crash), then $x{=}1$ ($x := 1$; persist x' must have also persisted before the crash, as given by $\mathrm{x^P}$).

By contrast, in the absence of persist x' in Fig. 4b, we have $z{=}1 \Rightarrow x \in \{0, 1\}$, as denoted by x. This is because as $x \in \mathrm{Loc_{wb}}$, the $x := 1$ (in C) may persist after $z := 1$, even though it is executed before it. As such, if a crash occurs after $z := 1$ has executed and persisted but before $x := 1$ has persisted, it is possible to observe $z{=}1$, $x{=}0$ on recovery, even though $z{=}1$, $x{=}0$ is never possible during non-crashing executions. Similarly, if $y \in \mathrm{Loc_{wb}}$, then in both Figs. 4a and 4b the $y := 1$ may persist after $z := 1$, and thus it is possible to observe $z{=}1$, $y{=}0$ on recovery, as captured by Y. On the other hand, if $y \notin \mathrm{Loc_{wb}}$ ($y \in \mathrm{Loc_{wc \cup uc \cup wt}}$), then in both Figs. 4a and 4b the $y := 1$ is persisted to memory as soon as it is executed (wc/uc/wt follow strict, unbuffered persistency); that is, $y := 1$ is persisted before $z := 1$ is executed. Therefore, if upon recovery $z{=}1$, then $y{=}1$, as denoted by $\mathrm{Y^P}$.

The remaining examples in Figs. 4c to 4f all use message passing and thus the values of y upon crash recovery are analogously captured by Y and $\mathrm{Y^P}$, depending on the memory type of y. Moreover, in all Figs. 4c to 4f, x is persisted as soon as it is executed since it is in wc/uc/wt memory (subject to strict, unbuffered persistency). In Figs. 4d to 4f, the $x := 1$ and $y := 1$ writes in C cannot be reordered as either x or y is in uc/wt memory. As such, in Figs. 4d to 4f $x := 1$ is executed before $y := 1$, and (due

to message passing) $y := 1$ is executed before $z := 1$; consequently, upon recovery if $z=1$, then $x=1$ (given by x^P). By contrast, in Fig. 4c as $x \in \text{Loc}_{wc}$ and $y \in \text{Loc}_{wc \cup wb}$, the $x := 1$ and $y := 1$ writes in C may be reordered. Consequently, $z := 1$ may be executed after $y := 1$ but before $x := 1$, additionally allowing us to observe $z=1$, $x=0$, upon recovery (given by x).

The Challenges of Validating Persistency. As discussed in §2.1, we have successfully extended the **diy** toolsuite [Alglave and Maranget 2021] to validate the Ex86 *consistency* model, whereby litmus testing techniques are used to monitor the *store order* (the order in which writes are made visible to threads). Unfortunately, however, these techniques cannot currently be used to validate a *persistency* model: doing so involves monitoring the *persist order* (the order in which writes are persisted from processor caches to non-volatile memory), which is invisible to the processors. Specifically, the contents of memory cannot be observed directly while employing caches, and a program (litmus test) alone cannot distinguish volatile (in-cache) data from persistent (in-memory) data. The only reliable way to ensure a write persists is to contrive a crash (turn off the machine) and then read persistent data from memory once the machine restarts. However, this only allows one to observe the latest persisted write for each memory location, and is not sufficient to infer the order in which earlier writes persisted. Moreover, continually restarting a machine makes it infeasible to run the large number of tests required for high-coverage validation.

As such, we could not validate PEx86 using conventional litmus-testing techniques. Nevertheless, we argue that PEx86 is a faithful description of Intel-x86 persistency. First, we note that PEx86 is an extension of Px86_{sim} (for wb persistency) which was developed in collaboration with (and endorsed by) Intel engineers [Raad et al. 2020]. Second, we formalised the PEx86 extensions (for wc/uc/wt memory and non-temporal writes) by thoroughly studying the Intel reference manual Intel [2021]. Lastly, we formalised these extensions after detailed discussions with the lead architect of the Intel instruction set system architecture, and they have confirmed that our understanding of the persistency semantics of these extensions and our reading of the manual text are accurate. Nevertheless, as we discuss in §7, we aim to validate PEx86 in the future by using custom hardware that allows us to monitor the traffic between the CPU caches and the memory, thus inferring the persist ordering. However, doing so is beyond the scope of this paper.

Verification Techniques for Ex86 and PEx86. As we discuss later, the Ex86/PEx86 models are conservative extensions of the $\text{TSO}/\text{Px86}_{sim}$ models, respectively, in that for programs that do not use the additional features of $\text{Ex86}/\text{Px86}_{sim}$ (non-temporal writes and non-wb memory types), the behaviours of Ex86 and TSO (respectively PEx86 and Px86_{sim}) coincide. As such, existing verification techniques for $\text{TSO}/\text{Px86}_{sim}$ can be used to reason about such Ex86/PEx86 programs, e.g. the program logics of OGRA [Lahav and Vafeiadis 2015] and POG [Raad et al. 2020], as well as the model checkers Nidhugg [Abdulla et al. 2015a] and GenMC [Kokologiannakis et al. 2019b]. Moreover, the full Ex86/PEx86 models (including non-temporal writes and all memory types) both meet the conditions stipulated by GenMC; as such, the Ex86 (resp. PEx86) model can be fed into GenMC as-is, yielding a model-checking technique for Ex86 (resp. PEx86) for free.

3 THE DECLARATIVE Ex86 SEMANTICS

Addresses, Locations and Cache Lines. We assume a set of *addresses*, ADDR, and define memory *locations* as pairs comprising an address and a memory type. We then define *cache lines* as disjoint location sets of the same memory type.

Definition 1 (Memory types). A *memory type* $t \in$ MTYPE may be *(strong)-uncacheable*, uc; *write-through*, wt; *write-back*, wb; or *(uncacheable)-write-combining*, wc.

Basic domains		Expressions and sequential commands	
$a \in \text{REG}$	Registers	$\text{EXP} \ni e ::= v \mid a \mid e + e \mid \cdots$	$x_{\text{nt}} \in \text{LOC}_{\text{wb} \cup \text{wt} \cup \text{wc}}$
$v \in \text{VAL}$	Values	$\text{PCOM} \ni c ::= \textbf{load}(x) \mid \textbf{store}(x, e) \mid \textbf{ntstore}(x_{\text{nt}}, e) \mid \textbf{CAS}(x, e, e')$	
$\tau \in \text{TID}$	Thread IDs	$\mid \textbf{mfence} \mid \textbf{sfence} \mid \textbf{flush}_{\text{opt}} \, x \mid \textbf{flush} \, x$	
Programs		$\text{COM} \ni C ::= e \mid c \mid \textbf{let} \, a := C \, \textbf{in} \, C$	
$P \in \text{PROG} \triangleq \text{TID} \xrightarrow{\text{fin}} \text{COM}$		$\mid \textbf{if} \, (C) \, \textbf{then} \, C \, \textbf{else} \, C \mid \textbf{repeat} \, C$	

Fig. 5. A simple concurrent programming language for Ex86 and PEx86

Derived memory types. We refer to the uc and wc types collectively as *non-cacheable*: $\text{nc} \triangleq \text{uc} \cup \text{wc}$. Analogously, we refer to wt and wb collectively as *cacheable* types: $\text{c} \triangleq \text{wt} \cup \text{wb}$.

Definition 2 (Locations and cache lines). Assume a set of *memory addresses* ADDR. The set of *memory locations*, $\text{LOC} \subseteq \text{ADDR} \times \text{MTYPE}$, is the largest subset of $\text{ADDR} \times \text{MTYPE}$ such that:

$$\forall \alpha, \beta, t, t'. \, (\alpha, t), (\beta, t') \in \text{LOC} \wedge t \neq t' \Rightarrow \alpha \neq \beta$$

Assume a set of cache lines, $\text{CL} \subseteq \mathcal{P}(\text{LOC})$, such that:

(1) all locations in a cache line have the same memory type: $\forall X \in \text{CL}. \, \exists t. \, X \subseteq \text{LOC}_t$; and
(2) distinct cache lines are disjoint: $\forall X, Y \in \text{CL}. \, X = Y \vee X \cap Y = \emptyset$.

Given a memory type t, we write LOC_t for $\text{LOC} \cap \{(\alpha, t) \mid \alpha \in \text{ADDR}\}$; e.g. LOC_{uc} denotes locations in uc memory. Recall from §2 that we assume that the types of locations cannot change once assigned. As such, the definition of locations above requires that the addresses in LOC_{uc}, LOC_{wt}, LOC_{wb} and LOC_{wc} sets be pairwise disjoint. Given a memory type t, we use $x_t, y_t, \cdots$ as meta-variables for locations in LOC_t. We write x_{nt} when $x \in \text{LOC}_{\text{wb} \cup \text{wt} \cup \text{wc}}$, i.e. x is a location for which the non-temporal hint will not be ignored. When the memory type is immaterial, we simply use $x, y, \cdots$ as meta-variables for locations in LOC; we use $X, Y, \ldots$ as meta-variables for cache lines.

Programming Language. We employ a simple concurrent programming language as given in Fig. 5. We assume a finite set REG of registers (local variables); a finite set VAL of values; a finite set $\text{TID} \subseteq \mathbb{N}^+$ of thread identifiers; and any standard interpreted language for expressions, EXP, containing registers and values. We use v as a metavariable for values, τ for thread identifiers, and e for expressions. We model a multi-threaded program P as a function mapping each thread to its (sequential) program. We write $P = C_1 \mid\mid \cdots \mid\mid C_n$ when $dom(P) = \{\tau_1 \cdots \tau_n\}$ and $P(\tau_i) = C_i$. Sequential programs are described by the COM grammar and include *primitives* (c), as well as the standard constructs of expressions, assignments, conditionals and loops.

The $\textbf{load}(x)$ denotes a *read* from location x; similarly, the $\textbf{store}(x, e)$ (resp. $\textbf{ntstore}(x_{\text{nt}}, e)$) denotes a *write* (resp. non-temporal write) to x (resp. x_{nt}). The $\textbf{CAS}(x, e, e')$ denotes the atomic 'compare-and-swap', where the value of location x is compared against e: if the values match then the value of x is set to e' and 1 is returned; otherwise x is unchanged and 0 is returned. Analogously, $\textbf{FAA}(x, e)$ denotes the atomic 'fetch-and-add' operation, where the value of x is incremented by e and its old value is returned. The $\textbf{CAS}$ and $\textbf{FAA}$ are collectively known as *atomic update* or RMW ('read-modify-write') instructions. The $\textbf{mfence}$ and $\textbf{sfence}$ denote a *memory fence* and a *store fence*, respectively. Lastly, $\textbf{flush}_{\text{opt}}$ and $\textbf{flush}$ denote persist instructions as discussed in §2.

To aid readability, we may not follow syntactic conventions and write e.g. $a := C$ for $\textbf{let} \, a := C \, \textbf{in} \, a$, and $C_1; C_2$ for $\textbf{let} \, a := C_1 \, \textbf{in} \, C_2$, where a is a fresh local variable. We write $a := x$ for $\textbf{let} \, a := \textbf{load}(x) \, \textbf{in} \, a$; similarly for $\textbf{CAS}/\textbf{FAA}$. We write $x := e$ and $x :=_{\text{NT}} e$ for $\textbf{store}(x, e)$ and $\textbf{ntstore}(x, e)$, respectively.

3.1 Ex86: The Extended Intel-x86 Consistency Model

Executions and Events. In the literature of declarative models, the traces of shared memory accesses generated by a program are commonly represented as a set of *executions*, where each execution is a graph comprising: (i) a set of events (graph nodes); and (ii) a number of relations on events (graph edges). Each event corresponds to the execution of a primitive command ($c \in \text{PCom}$ in Fig. 5) and is a tuple of the form $e = \langle n, \tau, l \rangle$, where $n \in \mathbb{N}$ is the (unique) *event identifier*; $\tau \in \text{TID}$ is the thread identifier of the executing thread; and $l \in \text{LAB}$ is the event *label*, defined below.

Definition 3 (Labels and events). An *event* $e \in \text{EVENT}$ is a tuple $\langle n, \tau, l \rangle$, where $n \in \mathbb{N}$ is an event identifier, $\tau \in \text{TID}$ is a thread identifier, and $l \in \text{LAB}$ is an event *label*. A label may be:

- (R, x, v) to denote reading v from location x;
- (W, x, v) to denote writing v to x;
- $(\text{NTW}, x_{\text{nt}}, v)$ to denote non-temporally writing v to x_{nt};
- (U, x, v, v') to denote a successful update (RMW) modifying x to v' when its value matches v;
- $(\text{U}, x, v, \bot)$ to denote a failed update (RMW) when x holds v, with $\bot \notin \text{VAL}$;
- MF or SF to denote the execution of **mfence** or **sfence**, respectively; and
- (FO, x) or (FL, x) to denote the execution of **flush**$_{\text{opt}}$ x or **flush** x, respectively.

The set of *read events* is: $R \triangleq \{e \in \text{EVENT} \mid \exists x, v. \text{lab}(e) = (\text{R}, x, v)\}$. The sets of *writes* ($W$), *non-temporal writes* (NTW), *successful updates* (U_s), *failed updates* (U_f), *memory fences* (MF), *store fences* (SF), *flush-opt events* (FO) and *flushes* (FL) are defined analogously. We write U for successful and failed updates: $U \triangleq U_s \cup U_f$; and write ST for events that store to memory: $ST \triangleq W \cup U_s \cup NTW$.

The functions loc, val_r and val_w respectively project the location, the read value and the written value of a label, where applicable. For instance, $\text{loc}(l) = x$ and $\text{val}_w(l) = v$ for $l = (\text{W}, x, v)$. We typically use a, b and e to range over events. The functions tid and lab respectively project the thread identifier and the label of an event. We lift the label functions loc, val_r and val_w to events, and given an event e, we write e.g. $\text{loc}(e)$ for $\text{loc}(\text{lab}(e))$. An event e is an *initialisation* event iff $\text{tid}(e) = 0$. Given a set of events E, we write E^0 for E restricted to initialisation events.

Notation. Given a set of events A, a location x and a memory type t, we write A_x for $\{a \in A \mid \text{loc}(a) = x\}$ and A_t for $\{e \in E \mid \text{loc}(e) \in \text{LOC}_t\}$, e.g. R_{uc} denotes the set of read events on uc locations. Similarly, given a relation r, we write r_x for $r \cap (A_x \times A_x)$ and r_t for $r \cap (A_t \times A_t)$. Given a relation r and a set A, we write $r^?$, r^+ and r^* for the reflexive, transitive and reflexive-transitive closures of r, respectively. We write r^{-1} for the inverse of r; $r|_A$ for $r \cap (A \times A)$; $[A]$ for the identity relation on A, i.e. $\{(a, a) \mid a \in A\}$; $\text{irreflexive}(r)$ for $\nexists a. (a, a) \in r$; and $\text{acyclic}(r)$ for $\text{irreflexive}(r^+)$. We write $r_1; r_2$ for the relational composition of r_1 and r_2, i.e. $\{(a, b) \mid \exists c. (a, c) \in r_1 \land (c, b) \in r_2\}$. Finally, we define the *same-location* relation as $\text{sloc} \triangleq \{(a, b) \in \text{EVENT} \times \text{EVENT} \mid \text{loc}(a) = \text{loc}(b)\}$, and the *same-cache-line* relation as $\text{scl} \triangleq \{(a, b) \in \text{EVENT} \times \text{EVENT} \mid \exists X \in \text{CL}. \text{loc}(a) \in X \land \text{loc}(b) \in X\}$.

Definition 4 (Ex86 executions). An *execution*, $G \in \text{EXEC}$, is a tuple $(E, \text{po}, \text{rf}, \text{mo})$, where:

- E denotes a set of *events*, including a set of *initialisation* events, $E^0 \subseteq E$, comprising a single write event with label $(\text{W}, x, 0)$ for each $x \in \text{LOC}$.
- $\text{po} \subseteq E \times E$ denotes the *'program-order'* relation, defined as a disjoint union of strict total orders, each ordering the events of one thread, with $E^0 \times (E \setminus E^0) \subseteq \text{po}$.
- $\text{rf} \subseteq (E \cap ST) \times (E \cap (R \cup U))$ denotes the *'reads-from'* relation on events of the same location with matching values; i.e. $(a, b) \in \text{rf} \Rightarrow (a, b) \in \text{sloc} \land \text{val}_w(a) = \text{val}_r(b)$. Moreover, rf is total and functional on its range: every read/update is related to exactly one (non-temporal) write/update.
- $\text{mo} \triangleq \{\text{mo}_x\}_{x \in \text{LOC}}$ is the *'modification-order'*, such that each mo_x is a strict total order on ST_x, and $E_x^0 \times (ST_x \setminus E_x^0) \subseteq \text{mo}_x$.

ppo **Component**	**Label**	✓ in Fig. 2
ord $\triangleq E^0 \times E$	(INIT)	
$\cup\ E \times (E_{uc} \cup R_{wc} \cup W_{wt} \cup MF \cup SF \cup U) \cup (E \setminus FO) \times FL$	(COL)	Cols. 3,4,6,7,10,11,12,13
$\cup\ (R \cup U \cup MF) \times E$	(ROW)	Rows A,B,C,D,J,K
$\cup\ (W_{wt} \cup W_{uc} \cup SF) \times (E \setminus R_c) \cup FL \times (E \setminus FO \setminus R_c)$	(ROW2)	Rows F,G,L,M
$\cup\ W_{wb} \times W_{wb}$	(W-WB)	E5
$\cup\ ((W \cup NTW) \times (W \cup NTW)) \cap \mathsf{sloc}$	(W-LOC)	E5:I9
$\cup\ ((W \cup NTW) \times FO) \cap \mathsf{scl}$	(FO)	E14:I14

Fig. 6. The Ex86 ppo components on a given program order po $\subseteq E \times E$ with ppo(po) $\triangleq$ (po $\cap$ ord)$^+$

Given an execution G, we typically use the "G." prefix to extract the various components of G, e.g. G.po. When the choice of G is clear from the context, we drop the "G." prefix. The 'modification-order' mo describes the store order on the (non-temporal) writes and updates of each location.

Derived Relations. Given an execution $(E, \text{po}, \text{rf}, \text{mo})$, the derived *'reads-before'* relation is defined as rb $\triangleq$ rf^{-1}; mo, relating each read r to all writes that are mo-after the write r reads from. We further define the 'preserved program order' on po as in Def. 5 below. Intuitively, ppo $\subseteq$ po denotes the po edges between instructions that cannot be reordered; i.e. the non-✗ entries in Fig. 2.

Definition 5 (The ppo relation). Given $E \subseteq$ EVENT, a relation r $\subseteq E \times E$ is a *program order* iff (1) r orders the initialisation events in E before all others in E; and (2) r additionally comprises a disjoint union of strict total orders on the events of each thread. That is, r $\triangleq E^0 \times (E \setminus E^0) \cup \biguplus r_\tau$, where $r_\tau \subseteq E_\tau \times E_\tau$ is a strict total order on $E_\tau \triangleq \{e \in E \mid \text{tid}(e) = \tau\}$.

Given a program order po $\subseteq E \times E$, the Ex86 *'preserved-program-order'* on po is ppo(po) $\triangleq$ (po $\cap$ ord)$^+$, where ord is as defined in Fig. 6 with $E^0 \triangleq \{e \in E \mid \text{tid}(e)=0\}$.

Note that given an execution G, the conditions on G.po in Def. 4 ensure that G.po is indeed a program order. As such, in the context of an execution G we simply write ppo in lieu of ppo(G.po).

Definition 6 (Ex86-consistency). An execution $(E, \text{po}, \text{rf}, \text{mo})$ is *Ex86-consistent* iff:

- mo$_i$ $\cup$ rf$_i$ $\cup$ rb$_i$ $\subseteq$ po   (INTERNAL)
- acyclic(ob), where ob denotes the *'ordered-before'* relation defined below: (EXTERNAL)

$$\text{ob} \triangleq \text{ppo} \cup \text{mo}_e \cup \text{rf}_e \cup \text{rb}_e$$

Note that the INTERNAL and EXTERNAL axioms are identical to those of TSO in [Alglave et al. 2014], except that the Ex86 ppo relation is more elaborate than that of TSO: while ppo for TSO is simply po\$(W \times R)$ thus prohibiting write-read reordering (see Def. 7), the Ex86 ppo is as in Def. 5.

We next relate Ex86-consistency to the exiting memory models of SC ('sequential consistency') [Lamport 1979] and TSO ('total store order') [Sewell et al. 2010]. While SC allows no reordering, TSO allows write-read reordering (later reads (in program order) can be reordered before earlier writes). We further relate Ex86-consistency to a variant of PSO ('partial store order') which we refer to as *SPSO* ('strong PSO'). PSO is a weakening of TSO: in addition to write-read reordering it also allows write-write reordering on different locations. The SPSO model is a strengthening of PSO where write-read reordering is prohibited and thus only allows write-write reordering on different locations. As such, TSO and SPSO are incomparable: TSO allows write-read but not write-write reordering, whereas SPSO allows write-write (on different locations) but not write-read reordering.

In Theorem 1 we show that given an Ex86-consistent execution G comprising read, write, update and memory fences: (1) if all locations accessed in G are in uc memory, then G is also SC-consistent; (2) if all locations accessed in G are in c memory, then G is also TSO-consistent; and (3) if all locations accessed in G are in wc memory, then G is also SPSO-consistent. Note that (2) demonstrates that

Thread transitions: $\text{Com} \xrightarrow{\text{TID:LAB} \cup \{\epsilon\}} \text{Com}$ **Program transitions:** $\text{PROG} \xrightarrow{\text{TID:LAB} \cup \{\epsilon\}} \text{PROG}$

$$\frac{C_1 \xrightarrow{\tau:l} C_1'}{\textbf{let } a:=C_1 \textbf{ in } C_2 \xrightarrow{\tau:l} \textbf{let } a:=C_1' \textbf{ in } C_2} \text{ T-Let1} \qquad \frac{}{\textbf{let } a:=v \textbf{ in } C \xrightarrow{\tau:\epsilon} C[v/a]} \text{ T-Let2}$$

$$\frac{C \xrightarrow{\tau:l} C'}{\textbf{if } (C) \textbf{ then } C_1 \textbf{ else } C_2 \xrightarrow{\tau:l} \textbf{if } (C') \textbf{ then } C_1 \textbf{ else } C_2} \text{ T-If1} \qquad \frac{v \neq 0 \Rightarrow C=C_1 \quad v=0 \Rightarrow C=C_2}{\textbf{if } (v) \textbf{ then } C_1 \textbf{ else } C_2 \xrightarrow{\tau:\epsilon} C} \text{ T-If2}$$

$$\frac{}{\textbf{repeat } C \xrightarrow{\tau:\epsilon} \textbf{if } (C) \textbf{ then } (\textbf{repeat } C) \textbf{ else } 0} \text{ T-Repeat} \qquad \frac{}{\textbf{store}(x,v) \xrightarrow{\tau:(\text{W},x,v)} v} \text{ T-Write}$$

$$\frac{}{\textbf{ntstore}(x,v) \xrightarrow{\tau:(\text{NTW},x,v)} v} \text{ T-NTW} \qquad \frac{}{\textbf{load}(x) \xrightarrow{\tau:(\text{R},x,v)} v} \text{ T-Read} \qquad \frac{}{\textbf{mfence} \xrightarrow{\tau:\text{MF}} 1} \text{ T-MF}$$

$$\frac{v \neq v_1}{\textbf{CAS}(x,v_1,v_2) \xrightarrow{\tau:(\text{U},x,v,\perp)} 0} \text{ T-CAS0} \qquad \frac{}{\textbf{CAS}(x,v_1,v_2) \xrightarrow{\tau:(\text{U},x,v_1,v_2)} 1} \text{ T-CAS1} \qquad \frac{}{\textbf{sfence} \xrightarrow{\tau:\text{SF}} 1} \text{ T-SF}$$

$$\frac{}{\textbf{flush}_{\text{opt}} \, x \xrightarrow{\tau:(\text{FO},x)} 1} \text{ T-FO} \qquad \frac{}{\textbf{flush } x \xrightarrow{\tau:(\text{FL},x)} 1} \text{ T-FL} \qquad \frac{P(\tau) \xrightarrow{\tau:l} C}{P \xrightarrow{\tau:l} P[\tau \mapsto C]} \text{ PROG}$$

Fig. 7. Program transitions in Ex86 and PEx86

Ex86 is a *conservative* extension of TSO: for programs that do not use the additional features of Ex86 (non-temporal writes and non-wb memory types), the behaviours of Ex86 and TSO coincide. As such, existing verification techniques for TSO [Abdulla et al. 2015a; Kokologiannakis et al. 2019b; Lahav and Vafeiadis 2015] can be used to reason about such Ex86 programs.

Definition 7 (SC, TSO and SPSO consistency). Given a memory model MM $\in$ {SC, TSO, SPSO}, an execution $(E, \text{po}, \text{rf}, \text{mo})$ is MM-consistent if:

- $\text{mo}_i \cup \text{rf}_i \cup \text{rb}_i \subseteq \text{po}$ (MM-internal)
- $\text{acyclic}(\text{ppo}_{\text{MM}} \cup \text{mo}_e \cup \text{rf}_e \cup \text{rb}_e)$ with: (MM-external)
  $\text{ppo}_{\text{SC}} \triangleq \text{po}$ $\text{ppo}_{\text{TSO}} \triangleq \text{po} \setminus (W \times R)$ $\text{ppo}_{\text{SPSO}} \triangleq \text{po} \setminus ((W \times W) \setminus \text{sloc})$

Theorem 1. *For all executions* $G = (E, \text{po}, \text{rf}, \text{mo})$ *such that* $E \subseteq R \cup W \cup U \cup MF$:

(1) if $\text{loc}(E) \subseteq Loc_{\text{uc}}$, *then G is SC-consistent iff G is Ex86-consistent, where* $\text{loc}(E) \triangleq \{\text{loc}(e) \mid e \in E\}$;
(2) if $\text{loc}(E) \subseteq Loc_{\text{c}}$, *then G is TSO-consistent iff G is Ex86-consistent; and*
(3) if $\text{loc}(E) \subseteq Loc_{\text{wc}}$, *then G is SPSO-consistent iff G is Ex86-consistent.*

4 THE OPERATIONAL Ex86 SEMANTICS

We describe the Ex86 operational model by separating the transitions of its *program* and *storage* subsystems. The former describe the steps in program execution, e.g. how a conditional branch is triggered. The latter describe how the storage subsystem evolves throughout the execution, e.g. how memory writes reach the memory. The Ex86 operational semantics is then defined by combining the transitions of its program and storage subsystems.

Ex86 Program Transitions. The Ex86 program transitions in Fig. 7 are defined via the transitions of their constituent threads. Thread transitions are of the form: $C \xrightarrow{\tau:l} C'$, where $C, C' \in \text{COM}$ (Fig. 5). The $\tau:l$ marks the transition by recording the identifier of the executing thread τ, as well as the transition label l, which may be ϵ for no-ups, or in LAB (Def. 3) for primitive commands.

Most thread transitions are standard. The T-CAS0 transition describes the reduction of the $\textbf{CAS}(x, v_1, v_2)$ instruction when unsuccessful; i.e. when the value read (v) is different from v_1. The

T-CAS1 transition dually describes the reduction of a **CAS** when successful. The T-MF and T-SF transitions respectively describe executing an **mfence** and **sfence**, reducing to value 1 (successful termination). Analogously, T-FO and T-FL describe the execution of persist instructions.

Program transitions are of the form: $P \xrightarrow{\tau:l} P'$, where $P, P' \in$ Prog denote multi-threaded programs (Fig. 5). Program transitions are given by simply lifting the transitions of their threads.

We model the Ex86 storage subsystem after that of TSO by Sewell et al. [2010] (accounting for wb memory only). We thus proceed with a brief account of the TSO storage subsystem and then describe how we generalise it to model the Ex86 storage subsystem.

The TSO Storage Subsystem. In the TSO storage subsystem, each thread is connected to the (volatile) memory via a *local buffer*, as illustrated in Fig. 8. Specifically, the execution of writes is *delayed*: when a thread issues a write, the write is recorded only in its buffer. The delayed writes are *debuffered in FIFO order* and propagated to the memory at non-deterministic times. By contrast, reads are executed in *real time*. When a thread issues a read from a location x, it first consults its own buffer. If it contains delayed writes for x, it reads the value of the *last* buffered write to x; otherwise, it consults the memory. In other words, local buffers are used to model instruction reordering: one can model the reordering of a later read r before an earlier write w (cell E1 of Fig. 2) by delaying the debuffering of w until after r has executed in real time. Programmers can use **mfence** and RMW instructions to control this debuffering: executing an **mfence** or an RMW blocks until all delayed writes in the buffer of the executing thread are propagated to the memory.

The Ex86 Storage Subsystem. As in TSO, in the Ex86 storage subsystem each thread is connected to the (volatile) memory via a local buffer (as in Fig. 8) used to model instruction reordering. Analogously, as cacheable reads (on c locations) can be reordered before write, non-temporal write, **flush**, **flush**$_{opt}$ and **sfence** instructions, we delay the execution of these instructions by recording them in the local buffer, while executing reads in real-time. Intuitively, in both TSO and Ex86 the order in which the entries are added to the buffer corresponds to po, while the order they are removed (debuffered) corresponds to ppo. Specifically, since the TSO buffers only contain writes and write-write ordering is preserved under TSO (po $\cap (W \times W) \subseteq$ ppo$_{TSO}$), under TSO the order in which entries are added and re-

Fig. 8. The TSO/Ex86 storage subsystem

moved from the local buffer agree, i.e. the buffer entries are removed in the FIFO order. By contrast, the Ex86 local buffers contain write, non-temporal write, **flush**, **flush**$_{opt}$ and **sfence** entries, and since their respective orders are not included in ppo, the FIFO order and ppo may disagree, and it is possible for a later entry to be debuffered before an earlier one. For instance, under Ex86 a later write w may be reordered before a **flush**$_{opt}$ instruction fo (e.g. cell N5 in Fig. 2); as such, fo will be added to the buffer before w, but may be debuffered after w.
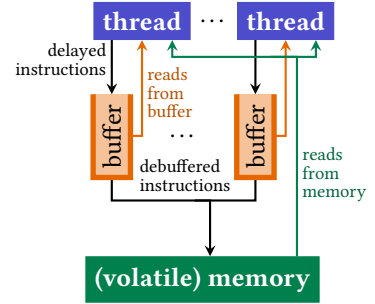
Ex86 Storage Transitions. The Ex86 storage transitions are given in Fig. 9 and are of the form: $M, B \xrightarrow{\tau:l} M', B'$, where $M, M' \in$ Mem denote the *memory*, modelled as a (finite) map from locations to values; and $B, B' \in$ BMap denote the *buffer map*, associating each thread with its *local buffer*. Each local buffer $b \in$ Buff is modelled as as a sequence of BLab, recording the labels of delayed instruction, namely those of writes, non-temporal writes, **flush**, **flush**$_{opt}$ and **sfence** instructions.

Ex86 Storage transitions: $\text{SConf} \xrightarrow{\text{Tid:Lab} \cup \{\epsilon\}} \text{SConf}$ $\qquad\qquad\qquad \Sigma \in \text{SConf} \triangleq \text{Mem} \times \text{BMap}$

$$M \in \text{Mem} \triangleq \text{Loc} \xrightarrow{\text{fin}} \text{Val} \qquad\qquad \text{BLab} \triangleq \text{Lab} \cap \{(W, x, v), (\text{NTW}, x, v), (\text{FL}, x), (\text{FO}, x), \text{SF}\}$$

$$B \in \text{BMap} \triangleq \text{Tid} \xrightarrow{\text{fin}} \text{Buff} \qquad\qquad b \in \text{Buff} \triangleq \text{Seq} \langle \text{BLab} \rangle$$

$$\frac{x \in \text{Loc}_c \qquad \text{rd}(M, B(\tau), x) = v}{M, B \xrightarrow{\tau:(R, x, v)} M, B} \text{ M-ReadC} \qquad\qquad \frac{x \in \text{Loc}_{nc} \qquad B(\tau) = \epsilon \qquad M(x) = v}{M, B \xrightarrow{\tau:(R, x_{nc}, v)} M, B} \text{ M-ReadNC}$$

$$\frac{l \in \text{BLab} \qquad B(\tau) = b}{M, B \xrightarrow{\tau:l} M, B[\tau \mapsto b.l]} \text{ M-Buff} \qquad\qquad \frac{B(\tau) = \epsilon \qquad M(x) = v_r}{M, B \xrightarrow{\tau:(U, x_{wb}, v_r, v_w)} M[x \mapsto v_w], B} \text{ M-RmwS}$$

$$\frac{B(\tau) = \epsilon}{M, B \xrightarrow{\tau:\text{MF}} M, B} \text{ M-MF} \qquad\qquad \frac{B(\tau) = \epsilon \qquad M(x) = v_r}{M, B \xrightarrow{\tau:(U, x_{wb}, v_r, \perp)} M, B} \text{ M-RmwF}$$

$$\frac{B(\tau) = b_1.l.b_2 \qquad l \in \{(W, x, v), (\text{NTW}, x, v)\} \qquad b_1 \cap \text{LPPO}(B(\tau), \tau) = \emptyset}{M, B \xrightarrow{\tau:\epsilon} M[x \mapsto v], B[\tau \mapsto b_1.b_2]} \text{ M-PropW+NTW}$$

$$\frac{B(\tau) = b_1.l.b_2 \qquad l \in \{(\text{FL}, x), (\text{FO}, x), \text{SF}\} \qquad b_1 \cap \text{LPPO}(B(\tau), \tau) = \emptyset}{M, B \xrightarrow{\tau:\epsilon} M, B[\tau \mapsto b_1.b_2]} \text{ M-PropFL+FO+SF}$$

$$\text{rd}(M, b, x) \triangleq \begin{cases} v & \text{if } \exists b_1, b_2, l. \ b = b_1.l.b_2 \wedge l \in \{(W, x, v), (\text{NTW}, x, v)\} \wedge \forall v'. \ (W, x, v'), (\text{NTW}, x, v') \notin b_2 \\ M(x) & \text{otherwise} \end{cases}$$

$$\text{PO}(b, \tau) \triangleq \left\{ ((n_1, \tau, l_1), (n_2, \tau, l_2)) \ \middle| \ b\#_{n_1} = l_1 \wedge b\#_{n_2} = l_2 \wedge n_1 < n_2 \right\}$$
$$\text{LPPO}(b, \tau) \triangleq \left\{ (l_1, l_2) \ \middle| \ (-, -, l_1), (-, -, l_2) \in \text{ppo}(\text{PO}(b, \tau)) \right\}$$

Fig. 9. Storage transitions in Ex86 with ppo(.) as defined in Fig. 6

When a thread executes a delayed instruction with label $l \in \text{BLab}$, it appends l to its local buffer as described by M-Buff. Recall that when a thread reads from x, it first consults its own buffer, followed by the non-volatile memory (if no write on x is found in the local buffer). This lookup chain is captured by $\text{rd}(M, b, x)$ in the premise of M-ReadC, defined at the bottom of Fig. 9.

Note that M-ReadC captures reading from *cacheable* memory ($x \in \text{Loc}_c$), while M-ReadNC captures that of *non-cacheable* memory ($x \in \text{Loc}_{nc}$). Recall that unlike later cacheable reads, later non-cacheable reads cannot be reordered before any instruction. As such, in M-ReadNC we additionally require that the local buffer of the executing thread be empty ($B(\tau) = \epsilon$). This way, we do not allow the (real-time) execution of non-cacheable reads to overtake delayed instructions, thereby precluding later non-cacheable reads from being reordered before them. As the local buffer is empty (i.e. contains no writes on x), the value of x is read directly from the memory ($M(x) = v$).

Analogously, M-MF, M-RmwS and M-RmwF require that $B(\tau) = \epsilon$, ensuring that **mfence** and RMW instructions are not reordered with respect to other instructions. As such, since $B(\tau) = \epsilon$, in M-RmwS and M-RmwF the value of x is read directly from the memory ($M(x) = v_r$).

The M-PropW+NTW describes the debuffering of delayed writes and non-temporal writes. Recall that delayed entries are added and removed (debuffered) in po and ppo orders, respectively. To this end, we first compute the po on the delayed entries of τ via $\text{PO}(B(\tau), \tau)$, defined at the bottom of Fig. 9, where $b\#_n$ denotes the nth entry in b. Given po on delayed entries, we then use $\text{LPPO}(B(\tau), \tau)$ to compute ppo on delayed entries and lift it to their labels. That is, $\text{LPPO}(B(\tau), \tau)$ captures ppo on delayed entries and thus their debuffering must respect $\text{LPPO}(B(\tau), \tau)$, as stipulated

Operational semantics: $\textsc{Prog} \times \textsc{SConf} \Rightarrow \textsc{Prog} \times \textsc{SConf}$

$$\frac{P \xrightarrow{\tau:\epsilon} P'}{P, \Sigma \Rightarrow P', \Sigma} \ \textsc{SilentP} \qquad \frac{\Sigma \xrightarrow{\tau:\epsilon} \Sigma'}{P, \Sigma \Rightarrow P, \Sigma'} \ \textsc{SilentS} \qquad \frac{P \xrightarrow{\tau:l} P' \quad \Sigma \xrightarrow{\tau:l} \Sigma'}{P, \Sigma \Rightarrow P', \Sigma'} \ \textsc{Step}$$

Fig. 10. The operational semantics of Ex86 and PEx86

by $b_1 \cap \text{LPPO}(B(\tau), \tau) = \emptyset$: when $B(\tau) = b_1.l.b_2$, then l can be debuffered provided that the earlier entries in the buffer (b_1) are not ppo-before it. Once the write/non-temporal write is debuffered, its associated value is written to memory, thus capturing when its associated store finally takes place. Analogously, M-PropFL+FO+SF captures the debuffering of **flush**, **flush**$_{\text{opt}}$ and **sfence**.

Ex86 Combined Transitions. The Ex86 operational semantics is defined by combining the Ex86 program and storage transitions as shown in Fig. 10. SilentP describes the case where the program subsystem takes a silent step and thus the storage subsystem is left unchanged; *mutatis mutandis* for SilentS. Step describes the case where the program and storage subsystems both take the *same* transition (with the same label) and thus the transition effect is that of their combined effects.

Finally, we show that the declarative and operational characterisations of Ex86 are equivalent.

Theorem 2 (Ex86 equivalence). *The declarative and operational Ex86 models are equivalent.*

5 VALIDATING THE Ex86 MODEL

We validate our Ex86 model using the **diy** toolsuite [Alglave and Maranget 2021]. A more detailed account of our experiments, including our test base, executable Cat model and run logs, is available at [Raad et al. 2022b]. Our approach is as follows:

(1) We have transliterated the Ex86 declarative model (§3) into Cat, the domain-specific language of the **herd** simulator for memory models. This translation was immediate because Cat supports the relational syntax used in §3.
(2) We have built a test base of over 2200 tests. Most of our tests are derived from 20 tests (with up to 3 threads) that demonstrate SC ('sequential consistency') violations – see here. We have also included several tests that target specific objectives such as highlighting a given relaxation or non-relaxation (see here). The SC violations build upon an extension of the *critical cycles* of Shasha and Snir [1988] with up to 3 threads. Using these cycles, we have used **diy** to systematically construct variants of these basic tests by a) considering all possible memory types for each shared location; b) using non-temporal writes in place of standard writes; and c) inserting **flush**, **flush**$_{\text{opt}}$, **sfence** and **mfence** instructions at appropriate program points.
(3) For each of our tests, we have used an extension of the **herd** simulator to record the valid outcomes according to our Cat model. This extension implements the Intel-x86 memory types as well as the **flush** and **flush**$_{\text{opt}}$ instructions.
(4) We have also run our tests on existing hardware and recorded the set of observed outcomes. To do so, we have used multiple testing machines with various Intel-x86 CPU implementations, including multiple versions of the Intel coreI5, coreI7 and Xeon CPU. We have run each test at least 6×10^8 times, with some critical tests run up to a few billion times.
(5) To run the tests, we have extended the **klitmus** tool of Alglave et al. [2018] for testing the kernel memory model. Specifically, a) we have extended it to accept tests written in Intel-x86 assembly (**klitmus** originally only accepted tests written in idiomatic C following the Linux kernel conventions); and b) we have further extended it to allocate each shared variable in a different virtual page to allow for tests that specify the memory type of each variable. For the

$x, \in \text{Loc}_{\text{wb}}, y \in \text{Loc}_{\text{wc}}$		$x, y \in \text{Loc}_{\text{wb}}$		$x \in \text{Loc}_{\text{wc}}, y \in \text{Loc}_{\text{wb}}$		$x \in \text{Loc}_{\text{wb}}, y \in \text{Loc}_{\text{wb} \cup \text{wt}}$	
$x := 1;$	$a := y;$	$x :=_{\text{NT}} 1;$	$a := y;$	$x := 1;$	$a := y;$	$x := 1;$	$a := y;$
flush$_{\text{opt}}$ $y;$	**mfence**;	**flush**$_{\text{opt}}$ $y;$	**mfence**;	**flush**$_{\text{opt}}$ $z;$	**mfence**;	**flush**$_{\text{opt}}$ $z;$	**mfence**;
$y := 1;$	$b := x;$	$y := 1;$	$b := x;$	$y := 1;$	$b := x;$	$y :=_{\text{NT}} 1;$	$b := x;$
a=1 ∧ b=0: ✓		a=1 ∧ b=0: ✓		a=1 ∧ b=0: ✓		a=1 ∧ b=0: ✓	
(a)		(b)		(c)		(d)	

Fig. 11. Ex86 Selected litmus tests illustrating the weak behaviours of **flush**$_{\text{opt}}$, where ✓ denotes that the depicted weak behaviour is observed; see HERE for a full list of tests. In all examples, the weak behaviour depicted is not observed if the **flush**$_{\text{opt}}$ instruction is replaces with a corresponding **flush** instruction.

latter, our **klitmus** extension produces Linux kernel modules that call several Linux kernel functions, including those that change the memory-types of virtual pages.[3]

(6) Finally, we have compared the set of outcomes allowed by our formal Ex86 model against the set of outcomes observed by our experiments.

The main result of our experiments is the absence of invalid behaviours, thereby empirically validating our Ex86 model. That is, in our experiments we did not observe any behaviour that is forbidden by Ex86. Moreover, only 64 tests include behaviours that are allowed by Ex86 and yet not observed on hardware, showing that Ex86 closely captures existing implementations. Among those 64 tests, 53 are SC violations that involve the **flush**$_{\text{opt}}$ instruction, showing that the *implementation* of this relatively recent instruction is stronger than its *specification* both in the Intel manual and in Ex86, at least on the machines tested. The remaining (11) unobserved outcomes are valid SC behaviours whose concrete occurrence is thwarted by timing and synchronisation considerations.

Finally, we have empirically validated the results in Theorem 1: (1) changing the memory type of all shared variable from (the default) wb to uc rules out non-SC behaviours; and (2) changing the memory type of a shared variable to wc relaxes (the default) TSO model by allowing write-write reordering (when changing the memory type of any of the two writes involved).

The Weak flush$_{opt}$ Behaviours Observed. As discussed above, our experiments could not validate the weak behaviours of **flush**$_{\text{opt}}$ described in the Intel manual (as captured by Ex86), and in most cases the behaviours of **flush**$_{\text{opt}}$ and **flush** coincide. Nonetheless, a few of our tests confirmed the weaker behaviour of **flush**$_{\text{opt}}$ compared to that of **flush**; we present several such tests in Fig. 11. The examples shown are variants of the canonical 'message passing' (MP) litmus test, with an **mfence** between the reads in the right thread and a **flush**$_{\text{opt}}$ between the writes in the left thread.

Observe that the read instructions cannot be reordered under Ex86, and the intervening **mfence** merely emphasises the absence of such read-read reordering. Note that were we to remove the **flush**$_{\text{opt}}$ in each example, the two writes could be reordered under Ex86 (given the use of non-temporal writes and the memory types of x, y), and thus the weak behaviour shown (a=1 ∧ b=0) could be observed. On the other hand, were we to replace the **flush**$_{\text{opt}}$ in each example with a corresponding **flush**, such write-write reordering would be prohibited (**flush** instructions are ordered against *all* earlier and later writes) and the weak behaviour shown would not be observed.

By contrast, our experiments confirmed that adding an intervening **flush**$_{\text{opt}}$ between the two writes (as shown in the examples) does not preclude such write-write reordering, allowing us to observe the weak behaviour shown. Nevertheless, it is unclear what *direction* of reordering enables the weak behaviour observed. For instance, in the case of Fig. 11a, it is unclear whether

---

[3]A previous attempt of building over the kvm infrastructure failed because we found no technique to control memory-types in a virtualised setting.

the weak behaviour shown is due to write-**flush**$_{opt}$ reordering (i.e. reordering $x := 1$ and **flush**$_{opt} y$) or **flush**$_{opt}$-write reordering (i.e. reordering **flush**$_{opt} y$ and $y := 1$). Specifically,our Ex86 model (as well as the Intel manual) admits both directions of reordering (cells E14 and N8 in Fig. 2), and it is thus unclear to which reordering we can attribute the weak behaviour observed. More generally, note that we cannot devise litmus tests that definitively distinguish write-**flush**$_{opt}$ reordering from **flush**$_{opt}$-write reordering. As such, since (1) the implementation of **flush**$_{opt}$ (on the machines tested) is in most cases stronger than its specification in the Intel manual, and (2) in cases where the weak behaviour of **flush**$_{opt}$ is observed we cannot conclusively determine the reordering direction, in Ex86 we opt to model all weak behaviours of **flush**$_{opt}$ as described in the Intel manual.

Caveats. Beyond the expected limitation that testing can only *validate* and not *prove* model compliance, our experiments have limitations in scope. Specifically, we could not test the wp (write-protected) memory type since no kernel function supports assigning this memory type. Moreover, we could not test non-temporal loads as they require enabling SSE ('streaming SIMD extensions'), which is disabled while compiling kernel code.

6 PEx86: THE PERSISTENT Ex86 MODEL

We extend Ex86 to develop declarative and operational characterisations of *PEx86* as the *first* formal model of Intel-x86 persistency that accounts for memory types and non-temporal writes. We show that the two characterisations of PEx86 are equivalent.

6.1 The Declarative PEx86 Model

Durable Events. In order to define the persistency semantics of Intel-x86, we first introduce the notion of *durable* events. Durable events are those whose effects *may* be observed when recovering from a crash. For instance, the effects of $x := v$ may be observed upon recovery if the write of v on x has persisted before the crash. As such, write events are durable. Note that durability does not reflect whether the effects of the instruction *do persist*; rather that its effect *could persist*. That is, regardless of whether the effects of $x := v$ persist, its associated label is deemed durable. By contrast, **mfence**, **sfence** and read instructions have no durable effects and their events are thus not durable.

Definition 8 (Durable events). The set of *durable events* is: $D \triangleq ST$.

Definition 9 (PEx86 executions). A *PEx86 execution*, G, is a tuple $(E, P, \text{po}, \text{rf}, \text{mo}, \text{pf})$, where:

- E, po, rf and mo are as defined in Def. 4.
- $P : \text{Loc} \rightarrow E \cap D$ associates each location x with a durable event on x (i.e. $\forall x. \, \text{loc}(P(x)) = x$).
- $\text{pf} \subseteq (E \cap ST) \times (E \cap (FL \cup FO))$ denotes the 'persists-from' relation on events of the same cache line: $(a, b) \in \text{pf} \Rightarrow (a, b) \in \text{scl}$. Moreover, for each event $e \in FL_x \cup FO_x$, and each location x' on the same cache line as x (i.e. $(x, x') \in \text{scl}$), pf relates e to exactly one durable event on x'.

Intuitively, for each location x, the $P(x)$ denotes the last durable event (store) on x whose effects have reached the persistent memory. The 'persists-from' relation relates each flush/flush-opt event e to the mo-latest store for each location persisted by e. This is analogous to rf; however, while rf relates a read/update to a single store, pf relates a flush/flush-opt to multiple stores (one for each location) on the same cache line. Analogously, the derived *'persists-before'* relation is defined as $\text{pb} \triangleq \text{pf}^{-1}; \text{mo}$, relating each flush/flush-opt to mo-later stores (*cf.* rb).

Definition 10 (PEx86-consistency). An execution $G = (E, P, \text{po}, \text{rf}, \text{mo}, \text{pf})$ is *PEx86-consistent* iff:

- G satisfies the INTERNAL axiom in Def. 6
- acyclic(ob), where ob is extended from Def. 6 as follows: (EXTERNAL-REVISED)

$$\text{ob} \triangleq \text{ppo} \cup \text{mo}_e \cup \text{rf}_e \cup \text{rb}_e \cup \text{pf} \cup \text{pb}$$

- $\forall x \in \text{Loc}_{\text{nc} \cup \text{wt}}.\ P(x) = \max(\text{mo}_x)$ (STRICT-PERSIST)
- $\forall x \in \text{Loc}_{\text{wb}}, e \in S_x.\ (e, P(x)) \in \text{mo}^?$ (WEAK-PERSIST)

  where $S \triangleq NTW_{\text{wb}} \cup dom(\text{pf}; [FL]) \cup dom(\text{pf}; [FO]; \text{po}; [MF \cup SF \cup U])$.

As with Ex86, the PEx86 axioms ensure internal and external consistency, with ob extended with persistency relations pf and pb in EXTERNAL-REVISED. The STRICT-PERSIST describes the strict, unbuffered persistency of nc and wt memory: the last store persisted for $x \in \text{Loc}_{\text{nc} \cup \text{wt}}$ is the mo-latest write on x; i.e. writes on nc and wt memory become persistent (reach their persist point) when they become visible to other threads (reach their store point). Analogously, WEAK-PERSIST describes the weak persistency of wb memory: the last store persisted for $x \in \text{Loc}_{\text{wb}}$ must be mo-after all other stores to x that are guaranteed to have persisted, namely (1) non-temporal writes on wb memory (NTW_{wb}) since non-temporal writes follow strict, unbuffered persistency; and (2) wb writes persisted by a *persist* sequence, i.e. either a **flush** ($dom(\text{pf}; [FL])$) or a **flush**$_{\text{opt}}$ followed by **mfence**/**sfence**/RMW ($dom(\text{pf}; [FO]; \text{po}; [MF \cup SF \cup U])$).

6.2 The Operational PEx86 Model

As with Ex86, we describe the PEx86 operational model by separating the transitions of its program and storage subsystems. The PEx86 program transitions are those of Ex86 in Fig. 7; its storage transitions are more complex and are presented in Fig. 13. The PEx86 operational semantics is then defined by combining the transitions of its program and storage subsystems, as in Fig. 10.

The PEx86 Storage Subsystem. Recall that nc and wt memory follow strict, unbuffered persistency (their store and persist orders agree), and thus as in Ex86, when these writes are debuffered from thread-local buffers, they are immediately written to (non-volatile) memory. On the other hand, wb memory follows relaxed, buffered persistency; to model this, the PEx86 storage system has an additional layer compared to Ex86, namely a *persistence buffer*, as illustrated in Fig. 12. Intuitively, the persistence buffer contains those wb *writes* that are pending to be persisted to the (non-volatile) memory. As with the memory, the persistence buffer is accessible by all threads. However,

Fig. 12. The storage subsystem of PEx86

while the memory is non-volatile, the persistence buffer is volatile and its contents are lost upon a crash. When a delayed wb write w in the local buffer is debuffered, it is propagated to the persistence buffer; this debuffering denotes the *store* associated with w, i.e. when w is made visible to other threads. A pending wb writes w in the persistence buffer is in turn debuffered and propagated to the memory at non-deterministic points in time; this debuffering denotes the *persist* associated with w, i.e. when w is written durably to memory. This hierarchy models the notion that the store of each write takes place before its associated persist. Note that the real time execution of reads must accordingly traverse this hierarchy: when reading from x, the thread first inspects its own buffer and reads the value of the last buffered write to x if such a write exists; otherwise, it consults the persistence buffer for the value of the last persist-pending store to x if such a store exists; otherwise, it reads x from the memory. Recall that wb writes on distinct locations may persist in any order (see Fig. 3a), while those on the same location persist in the store order. Following [Khyzha and Lahav 2021], we thus model the persistence buffer as a per-location map (PBMAP in Fig. 13), associating each wb location x with a queue of persist-pending writes on x (PBUFF in Fig. 13). The pending
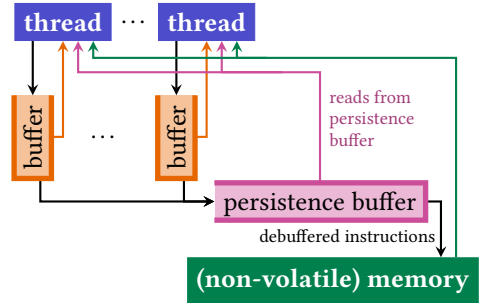
writes on each location x (within the same queue) are persisted in the FIFO order, while those on different locations (in different queues) are propagated in an arbitrary order.

Lastly, recall that while **flush** is synchronous, **flush**$_{\text{opt}}$ is asynchronous unless followed by **mfence**/**sfence**/RMWs. As such, (FL, x) entries may be debuffered from the local buffer (i.e. executed) only when there are no persist-pending writes in $\text{PB}(x')$, for each x' in the same cache line as x, thus modelling its synchronous behaviour. By contrast, when (FO, x) is debuffered, it is instead added to $\text{PB}(x')$ (to be removed at a later point) for each x' in the same cache line as x, thus modelling its asynchronous behaviour. Moreover, since executing **mfence**/RMWs renders **flush**$_{\text{opt}}$ instructions synchronous, they may be executed only if there are no pending **flush**$_{\text{opt}}$ entries by the executing thread in the persistence buffer. Similarly, an **sfence** entry may be removed from the local buffer (i.e. executed) only when there are no pending **flush**$_{\text{opt}}$ entries by the executing thread.

PEx86 Storage Transitions. The storage transitions are of the form M, PB, B $\xrightarrow{\tau:l}$ M′, PB′, B′, where M, M′ ∈ Mem and B, B′ ∈ BMap are as in Fig. 9, and PB, PB′ ∈ PBMap denotes the per-location *persistence buffer map*, associating each wb location with its *persistence buffer*. A persistence buffer, pb ∈ PBuff, is modelled as a sequence of entries of the form (1) w(v), denoting a persist pending write with value v on x; or (2) fo(τ), denoting a pending **flush**$_{\text{opt}}$ by thread τ on x.

The M-ReadNC and M-Buff are as in Ex86. Recall that when τ executes a read on x in wb memory, it first consults its local buffer B(τ), followed by the persistence buffer PB(x) (if no write to x is found in B(τ)), and then the memory (if no store to x is found in B(τ) or PB(x)). This lookup chain is captured by rd(M, PB(x), B(τ), x) in the premise of M-ReadC, defined at the bottom of Fig. 13.

As described above, executing **mfence**/RMWs by τ in M-MF, M-RmwS1, M-RmwS2, M-RmwF1 and M-RmwF2 additionally ensures that there are no pending **flush**$_{\text{opt}}$ entries by τ on any location in the persistence buffer: $\forall y.\, \text{fo}(\tau) \notin \text{PB}(y)$. Note that M-RmwS1 captures executing a successful RMW on a non-wb location ($x \notin \text{Loc}_{\text{wb}}$), where the update is written directly to memory. By contrast, M-RmwS2 denotes executing a successful RMW on a wb location ($x \in \text{Loc}_{\text{wb}}$), where the update is written to the persistence buffer PB(x) instead. Moreover, when reading the value of x, we must additionally check the persistence buffer (since it may contain persist-pending writes on x), as captured by the lookup chain rd(M, PB(x), ϵ, x). Analogously, M-RmwF1 and M-RmFS2 capture executing a failed RMW on a non-wb and wb location, respectively.

M-PropW1 describes the debuffering of wb writes from the local buffer, where the debuffered write is propagated to the persistence buffer. Analogously, M-PropW2 and M-PropNTW respectively describe the debuffering of non-wb writes and non-temporal writes, where the debuffered write is propagated directly to the memory. Recall from §2.2 that a non-temporal store on x in wb memory additionally behaves as a persist instruction in that it persists the pending writes on x to memory (see NT and Fig. 3k). This is captured by $x \in \text{Loc}_{\text{wb}} \Rightarrow \text{PB}(x) = \epsilon$ in the premise of M-PropNTW.

M-PropFL describes debuffering (FL, x) from the local buffer; as discussed above, this can happen only when PB(y) is empty for each y in the cache line of x: $\forall y.\, (x, y) \in \text{scl} \Rightarrow \text{PB}(y) = \epsilon$. Analogously, M-PropFO describes debuffering (FO, x) by thread τ, appending fo(τ) to PB(y) for each y is in the cache line of x, as denoted by the standard ternary conditional operator $(-?-:-)$ as PB′ = $\lambda y.\, (y, x) \in \text{scl} ?\, \text{PB}(y).\text{fo}(\tau) : \text{PB}(y)$. M-PropSF describes the debuffering of **sfence** entries by τ, ensuring that there are no pending fo(τ) on any location in the persistence buffer.

Lastly, M-PersistW and M-PersistFO describe removing entries from per-location persistence buffers in the FIFO order. In the former the removed persist-pending write is written (persisted) to memory. In the latter removing a pending **flush**$_{\text{opt}}$ from PB(x) is simply discarded: all earlier pending writes on x have already been persisted to memory in the FIFO order.

Finally, we show that the declarative and operational characterisations of PEx86 are equivalent.

PEx86 Storage transitions: $\text{SConf} \xrightarrow{\text{TID}:\text{LAB} \cup \{\epsilon\}} \text{SConf}$ $\qquad \Sigma \in \text{SConf} \triangleq \text{Mem} \times \text{PBMAP} \times \text{BMap}$

$\text{PB} \in \text{PBMAP} \triangleq \text{Loc}_{\text{wb}} \mapsto \text{PBuff}$ $\qquad \text{pb} \in \text{PBuff} \triangleq \text{Seq} \left\langle \left\{ \text{w}(v), \text{fo}(\tau) \mid v \in \text{Val}, \tau \in \text{TID} \right\} \right\rangle$

$$\frac{x \in \text{Loc}_{\text{c}} \qquad \text{rd}(M, \text{PB}(x), B(\tau), x) = v}{M, \text{PB}, B \xrightarrow{\tau:(R,x,v)} M, \text{PB}, B} \text{ M-ReadC} \qquad \frac{x \in \text{Loc}_{\text{nc}} \qquad B(\tau) = \epsilon \qquad M(x) = v}{M, \text{PB}, B \xrightarrow{\tau:(R,x_{\text{nc}},v)} M, \text{PB}, B} \text{ M-ReadNC}$$

$$\frac{l \in \text{BLab} \qquad B(\tau) = b}{M, \text{PB}, B \xrightarrow{\tau:l} M, \text{PB}, B[\tau \mapsto b.l]} \text{ M-Buff} \qquad \frac{B(\tau) = \epsilon \quad M(x) = v_r \quad x \notin \text{Loc}_{\text{wb}} \quad \forall y. \text{fo}(\tau) \notin \text{PB}(y)}{M, \text{PB}, B \xrightarrow{\tau:(U,x_{\text{wb}},v_r,v_w)} M[x \mapsto v_w], \text{PB}, B} \text{ M-RmwS1}$$

$$\frac{B(\tau) = \epsilon \quad \forall y. \text{fo}(\tau) \notin \text{PB}(y)}{M, \text{PB}, B \xrightarrow{\tau:\text{MF}} M, \text{PB}, B} \text{ M-MF} \qquad \frac{\begin{array}{c} B(\tau) = \epsilon \quad \text{rd}(M, \text{PB}(x), \epsilon, x) = v_r \quad x \in \text{Loc}_{\text{wb}} \\ \forall y. \text{fo}(\tau) \notin \text{PB}(y) \qquad \text{PB}' = \text{PB}[x \mapsto \text{PB}(x).\text{w}(v_w)] \end{array}}{M, \text{PB}, B \xrightarrow{\tau:(U,x,v_r,v_w)} M, \text{PB}', B} \text{ M-RmwS2}$$

$$\frac{\begin{array}{c} B(\tau) = \epsilon \qquad M(x) = v_r \\ x \notin \text{Loc}_{\text{wb}} \qquad \forall y. \text{fo}(\tau) \notin \text{PB}(y) \end{array}}{M, \text{PB}, B \xrightarrow{\tau:(U,x_{\text{wb}},v_r,\perp)} M, \text{PB}, B} \text{ M-RmwF1} \qquad \frac{\begin{array}{c} B(\tau) = \epsilon \qquad \text{rd}(M, \text{PB}(x), \epsilon, x) = v_r \\ x \in \text{Loc}_{\text{wb}} \qquad \forall y. \text{fo}(\tau) \notin \text{PB}(y) \end{array}}{M, \text{PB}, B \xrightarrow{\tau:(U,x,v_r,\perp)} M, \text{PB}, B} \text{ M-RmwF2}$$

$$\frac{B(\tau) = b_1.(\text{W}, x, v).b_2 \qquad b_1 \cap \text{LPPO}(B(\tau), \tau) = \emptyset \qquad x \in \text{Loc}_{\text{wb}} \qquad \text{PB}(x) = \text{pb}}{M, \text{PB}, B \xrightarrow{\tau:\epsilon} M, \text{PB}[x \mapsto \text{pb}.\text{w}(v)], B[\tau \mapsto b_1.b_2]} \text{ M-PropW1}$$

$$\frac{B(\tau) = b_1.(\text{W}, x, v).b_2 \qquad b_1 \cap \text{LPPO}(B(\tau), \tau) = \emptyset \qquad x \notin \text{Loc}_{\text{wb}}}{M, \text{PB}, B \xrightarrow{\tau:\epsilon} M[x \mapsto v], \text{PB}, B[\tau \mapsto b_1.b_2]} \text{ M-PropW2}$$

$$\frac{B(\tau) = b_1.(\text{NTW}, x, v).b_2 \qquad b_1 \cap \text{LPPO}(B(\tau), \tau) = \emptyset \qquad x \in \text{Loc}_{\text{wb}} \Rightarrow \text{PB}(x) = \epsilon}{M, \text{PB}, B \xrightarrow{\tau:\epsilon} M[x \mapsto v], \text{PB}, B[\tau \mapsto b_1.b_2]} \text{ M-PropNTW}$$

$$\frac{B(\tau) = b_1.(\text{FL}, x).b_2 \qquad b_1 \cap \text{LPPO}(B(\tau), \tau) = \emptyset \qquad \forall y. (x, y) \in \text{scl} \Rightarrow \text{PB}(y) = \epsilon}{M, \text{PB}, B \xrightarrow{\tau:\epsilon} M, \text{PB}, B[\tau \mapsto b_1.b_2]} \text{ M-PropFL}$$

$$\frac{\begin{array}{c} B(\tau) = b_1.(\text{FO},x).b_2 \quad b_1 \cap \text{LPPO}(B(\tau),\tau) = \emptyset \\ \text{PB}' = \lambda y. (y, x) \in \text{scl} ? \text{PB}(y).\text{fo}(\tau) : \text{PB}(y) \end{array}}{M, \text{PB}, B \xrightarrow{\tau:\epsilon} M, \text{PB}', B[\tau \mapsto b_1.b_2]} \text{ M-PropFO} \qquad \frac{\begin{array}{c} B(\tau) = b_1.\text{SF}.b_2 \quad \forall y. \text{fo}(\tau) \notin \text{PB}(y) \\ b_1 \cap \text{LPPO}(B(\tau),\tau) = \emptyset \end{array}}{M, \text{PB}, B \xrightarrow{\tau:\epsilon} M, \text{PB}, B[\tau \mapsto b]} \text{ M-PropSF}$$

$$\frac{\text{PB}(x) = \text{w}(v).\text{pb}}{M, \text{PB}, B \xrightarrow{\tau:\epsilon} M[x \mapsto v], \text{PB}[x \mapsto \text{pb}], B} \text{ M-PersistW} \qquad \frac{\text{PB}(x) = \text{fo}(-).\text{pb}}{M, \text{PB}, B \xrightarrow{\tau:\epsilon} M, \text{PB}[x \mapsto \text{pb}], B} \text{ M-PersistFO}$$

$$\text{rd}(M, \text{pb}, b, x) \triangleq \begin{cases} v & \text{if } \exists b_1, b_2, l. \ b = b_1.l.b_2 \wedge l \in \{(\text{W}, x, v), (\text{NTW}, x, v)\} \wedge \forall v'. (\text{W}, x, v'), (\text{NTW}, x, v') \notin b_2 \\ v & \text{else if } \exists \text{pb}_1, \text{pb}_2. \ \text{pb} = \text{pb}_1.\text{w}(v).\text{pb}_2 \wedge \forall v'. \text{w}(v') \notin \text{pb}_2 \\ M(x) & \text{otherwise} \end{cases}$$

Fig. 13. Storage transitions in PEx86, where highlighted sections denote extensions from Ex86

Theorem 3 (PEx86 equivalence). *The declarative and operational PEx86 models are equivalent.*

7 RELATED AND FUTURE WORK

Existing literature includes several examples of *consistency* models, both at hardware and software levels. On the hardware side, Sewell et al. [2010] presented the first formal model of Intel-x86 consistency, later reformulated by Alglave et al. [2014]. Subsequently, Abdulla et al. [2015b] presented an alternative formulation of TSO (using load buffers rather than store buffers) which they argued to be more suitable for model checking. However, none of these works covered the consistency semantics of Intel-x86 memory types and non-temporal writes. Similarly, several works have formalised the semantics of the ARMv8 and POWER architectures, both operationally and declaratively [Alglave et al. 2021, 2014; Chakraborty and Vafeiadis 2019; Flur et al. 2016; Mador-Haim et al. 2012; Pulte et al. 2018; Sarkar et al. 2011]. On the software side, there has been a number of formal models for C11 consistency [Batty et al. 2011; Kang et al. 2017; Lahav et al. 2016, 2017; Lee et al. 2020; Nienhuis et al. 2016; Pichon-Pharabod and Sewell 2016] with verified compilation schemes [Moiseenko et al. 2020; Podkopaev et al. 2017, 2019], Java [Bender and Palsberg 2019; Manson et al. 2005], the Linux kernel [Alglave et al. 2018] and the ext4 filesystem [Kokologiannakis et al. 2021].

Due to the emerging nature of NVM technology, the literature on formal persistency models is more limited. On the hardware side, Raad and Vafeiadis [2018] developed a *proposal* for Intel-x86 persistency, which is rather different from the existing Intel-x86 model in [Intel 2021]. Raad et al. [2020] later formalised the persistency semantics of Intel-x86 as described in [Intel 2021], which was later refined in [Abdulla et al. 2021; Cho et al. 2021; Khyzha and Lahav 2021]. However, all three formal models covered the persistency semantics of wb memory alone, and excluded the other Intel-x86 memory types and non-temporal stores. Similarly, Raad et al. [2019] formalised the persistency semantics of the ARMv8 architecture declaratively; Cho et al. [2021] later developed an operational ARMv8 persistency model. On the software side, there are several proposals of language-level persistency [Alshboul et al. 2021; Gogte et al. 2018, 2020; Kolli et al. 2017], as well as higher-level persistency approaches such as transactions [Avni et al. 2015; Intel 2015; Kolli et al. 2016; Raad et al. 2019; Shu et al. 2018; Tavakkol et al. 2018]. Lastly, Kokologiannakis et al. [2021] recently formalised the persistency semantics of the ext4 filesystem.

Future Work. We plan to build on top of this work in several ways. First, we will revisit the compilation schemes from high-level languages such as C11 to Intel-x86, and study how they can take advantage of the additional Intel-x86 memory types and non-temporal writes for better performance. Second, equipped with a formal understanding of Intel-x86 non-temporal writes, we will specify and verify several crucial fragments of the PMDK library [Intel 2015] that use this feature (e.g. for persistent transactions). Recall from §2 that a key challenge of testing persistency is that the persist order is not directly observable. To address this, as a third direction of future work we will build custom hardware that allows us to monitor the traffic to persistent memory, and thus to observe the persist order directly. This can be achieved when the processor under test is a component of a system-on-chip (SoC) FPGA [Jain et al. 2018]. Lastly, we will develop verification techniques for Ex86 and PEx86, including program logics such as those of [Dalvandi et al. 2020; Doko and Vafeiadis 2016, 2017; Kaiser et al. 2017; Raad et al. 2020; Turon et al. 2014; Vafeiadis and Narayan 2013], and stateless model checking [Kokologiannakis et al. 2021, 2019a,b; Kokologiannakis and Vafeiadis 2020]. The latter would allow us to verify an Ex86/PEx86 program by exhaustively generating its executions and inspecting them for consistency/persistency violations.

REFERENCES

Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. 2015a. Stateless Model Checking for TSO and PSO *(LNCS)*, Vol. 9035. Springer, Berlin, Heidelberg, 353–367. https://doi.org/10.1007/978-3-662-46681-0_28

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Ahmed Bouajjani, K. Narayan Kumar, and Prakash Saivasan. 2021. Deciding Reachability under Persistent X86-TSO. *Proc. ACM Program. Lang.* 5, POPL, Article 56 (Jan. 2021), 32 pages. https://doi.org/10.1145/3434337

Parosh Aziz Abdulla, Mohamed Faouzi Atig, and Tuan-Phong Ngo. 2015b. The Best of Both Worlds: Trading Efficiency and Optimality in Fence Insertion for TSO. In *Proceedings of the 24th European Symposium on Programming on Programming Languages and Systems - Volume 9032*. Springer-Verlag New York, Inc., New York, NY, USA, 308–332. https://doi.org/10.1007/978-3-662-46669-8_13

Jade Alglave, William Deacon, Richard Grisenthwaite, Antoine Hacquard, and Luc Maranget. 2021. Armed cats: formal concurrency modelling at Arm. *ACM Trans. Program. Lang. Syst.* (2021). http://www0.cs.ucl.ac.uk/staff/j.alglave/papers/toplas21.pdf

Jade Alglave and Luc Maranget. 2011–2021. The diy7 tool suite, Software and Documentation. (2011–2021). http://diy.inria.fr/

Jade Alglave, Luc Maranget, Paul E. McKenney, Andrea Parri, and Alan Stern. 2018. Frightening Small Children and Disconcerting Grown-Ups: Concurrency in the Linux Kernel. *SIGPLAN Not.* 53, 2 (March 2018), 405âĂŞ418. https://doi.org/10.1145/3296957.3177156

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

Mohammad Alshboul, Prakash Ramrakhyani, William Wang, James Tuck, and Yan Solihin. 2021. BBB: Simplifying Persistent Programming using Battery-Backed Buffers. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 111–124. https://doi.org/10.1109/HPCA51647.2021.00019

Anonymous. 2021. Intel-x86:The interaction between WC, WB and UC Memory. (2021). https://stackoverflow.com/questions/66978388/intel-x86the-interaction-between-wc-wb-and-uc-memory

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 (DISC 2015)*. Springer-Verlag, Berlin, Heidelberg, 617–630. https://doi.org/10.1007/978-3-662-48653-5_41

Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ Concurrency. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. ACM, New York, NY, USA, 55–66. https://doi.org/10.1145/1926385.1926394

John Bender and Jens Palsberg. 2019. A Formalization of JavaâĂŹs Concurrent Access Modes. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 142 (Oct. 2019), 28 pages. https://doi.org/10.1145/3360568

Ahmed Bouajjani, Egor Derevenetc, and Roland Meyer. 2013. Checking and Enforcing Robustness against TSO. In *ESOP 2013 (LNCS)*, Vol. 7792. Springer, 533–553. https://doi.org/10.1007/978-3-642-37036-6_29

Soham Chakraborty and Viktor Vafeiadis. 2019. Grounding Thin-Air Reads with Event Structures. *Proc. ACM Program. Lang.* 3, POPL, Article 70 (Jan. 2019), 28 pages. https://doi.org/10.1145/3290383

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 2021)*. Association for Computing Machinery, New York, NY, USA, 16âĂŞ31. https://doi.org/10.1145/3453483.3454027

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 (SOSP '09)*. ACM, New York, NY, USA, 133–146. https://doi.org/10.1145/1629575.1629589

Sadegh Dalvandi, Simon Doherty, Brijesh Dongol, and Heike Wehrheim. 2020. Owicki-Gries Reasoning for C11 RAR. In *34th European Conference on Object-Oriented Programming (ECOOP 2020) (Leibniz International Proceedings in Informatics (LIPIcs))*, Robert Hirschfeld and Tobias Pape (Eds.), Vol. 166. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 11:1–11:26. https://doi.org/10.4230/LIPIcs.ECOOP.2020.11

DML. 2021. (2021). https://github.com/intel/DML

Marko Doko and Viktor Vafeiadis. 2016. A Program Logic for C11 Memory Fences. In *Verification, Model Checking, and Abstract Interpretation*, Barbara Jobstmann and K. Rustan M. Leino (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 413–430.

Marko Doko and Viktor Vafeiadis. 2017. Tackling Real-Life Relaxed Concurrency with FSL++. In *Programming Languages and Systems*, Hongseok Yang (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 448–475.

DPDK. 2021. (2021). https://www.dpdk.org/

Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. 2016. Modelling the ARMv8 Architecture, Operationally: Concurrency and ISA. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. Association for Computing Machinery, New York, NY, USA, 608âĂŞ621. https://doi.org/10.1145/2837614.2837615

Free Software Foundation. 2016. (2016). https://elixir.bootlin.com/glibc/glibc-2.34/source/sysdeps/x86_64/multiarch/memmove-vec-unaligned-erms.S#L36

GitHub. 2019. (2019). https://github.com/spdk/spdk/commit/7b0579df170f90b2d6b704116dea65739f9442cd

GitHub. 2021. (2021). https://github.com/search?q=MOVNTI&type=Code

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 (PLDI 2018)*. ACM, New York, NY, USA, 46–61. https://doi.org/10.1145/3192366.3192367

Vaibhav Gogte, William Wang, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. 2020. Relaxed Persist Ordering Using Strand Persistency. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 652–665. https://doi.org/10.1109/ISCA45697.2020.00060

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

Intel. 2019. Intel 64 and IA-32 Architectures Software Developer's Manual (Combined Volumes). This is an old version of the manual that has been since removed from the Intel pages and the referenced document is an internet archive. (May 2019). https://web.archive.org/web/20190525125151/https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf Order Number: 325462-070US.

Intel. 2021. Intel 64 and IA-32 Architectures Software Developer's Manual (Combined Volumes). (April 2021). https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4.html Order Number: 325462-074US.

Abhishek Kumar Jain, Scott Lloyd, and Maya Gokhale. 2018. Microscope on Memory: MPSoC-Enabled Computer Memory System Assessments. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 173–180. https://doi.org/10.1109/FCCM.2018.00035

Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. 2017. Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris. In *31st European Conference on Object-Oriented Programming (ECOOP 2017) (Leibniz International Proceedings in Informatics (LIPIcs))*, Peter Müller (Ed.), Vol. 74. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 17:1–17:29. https://doi.org/10.4230/LIPIcs.ECOOP.2017.17

Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A Promising Semantics for Relaxed-Memory Concurrency. *SIGPLAN Not.* 52, 1 (Jan. 2017), 175âĂŞ189. https://doi.org/10.1145/3093333.3009850

Artem Khyzha and Ori Lahav. 2021. Taming X86-TSO Persistency. *Proc. ACM Program. Lang.* 5, POPL, Article 47 (Jan. 2021), 29 pages. https://doi.org/10.1145/3434328

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

Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. 2019a. Effective Lock Handling in Stateless Model Checking. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 173 (Oct. 2019), 26 pages. https://doi.org/10.1145/3360599

Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. 2019b. Model Checking for Weakly Consistent Libraries. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. ACM, New York, NY, USA, 96–110. https://doi.org/10.1145/3314221.3314609

Michalis Kokologiannakis and Viktor Vafeiadis. 2020. HMC: Model Checking for Hardware Memory Models. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 1157âĂŞ1171. https://doi.org/10.1145/3373376.3378480

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 (ISCA '17)*. ACM, New York, NY, USA, 481–493. https://doi.org/10.1145/3079856.3080229

Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. 2016. High-Performance Transactions for Persistent Memories. *SIGPLAN Not.* 51, 4 (March 2016), 399–411. https://doi.org/10.1145/2954679.2872381

Ori Lahav and Udi Boker. 2020. Decidable verification under a causally consistent shared memory. In *PLDI 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 211–226. https://doi.org/10.1145/3385412.3385966

Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. 2016. Taming Release-Acquire Consistency. *SIGPLAN Not.* 51, 1 (Jan. 2016), 649âĂŞ662. https://doi.org/10.1145/2914770.2837643

Ori Lahav and Viktor Vafeiadis. 2015. Owicki-Gries Reasoning for Weak Memory Models. In *Automata, Languages, and Programming*, Magnús M. Halldórsson, Kazuo Iwama, Naoki Kobayashi, and Bettina Speckmann (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 311–323.

Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing Sequential Consistency in C/C++11. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 618–632. https://doi.org/10.1145/3062341.3062352

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

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. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 362âĂŞ376. https://doi.org/10.1145/3385412.3386010

LWN. 2007. (2007). https://lwn.net/Articles/255364/

LWN. 2008. (2008). https://lwn.net/Articles/282250/

LWN. 2016. (2016). https://lwn.net/Articles/698014/

Sela Mador-Haim, Luc Maranget, Susmit Sarkar, Kayvan Memarian, Jade Alglave, Scott Owens, Rajeev Alur, Milo M. K. Martin, Peter Sewell, and Derek Williams. 2012. An Axiomatic Memory Model for POWER Multiprocessors. In *Computer Aided Verification*, P. Madhusudan and Sanjit A. Seshia (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 495–512.

Jeremy Manson, William Pugh, and Sarita V. Adve. 2005. The Java Memory Model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '05)*. Association for Computing Machinery, New York, NY, USA, 378âĂŞ391. https://doi.org/10.1145/1040305.1040336

Evgenii Moiseenko, Anton Podkopaev, Ori Lahav, Orestis Melkonian, and Viktor Vafeiadis. 2020. Reconciling Event Structures with Modern Multiprocessors (Artifact). *Dagstuhl Artifacts Series* 6, 2 (2020), 4:1–4:3. https://doi.org/10.4230/DARTS.6.2.4

Kyndylan Nienhuis, Kayvan Memarian, and Peter Sewell. 2016. An Operational Semantics for C/C++11 Concurrency. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. Association for Computing Machinery, New York, NY, USA, 111âĂŞ128. https://doi.org/10.1145/2983990.2983997

Jean Pichon-Pharabod and Peter Sewell. 2016. A Concurrency Semantics for Relaxed Atomics That Permits Optimisation and Avoids Thin-Air Executions. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. Association for Computing Machinery, New York, NY, USA, 622âĂŞ633. https://doi.org/10.1145/2837614.2837616

Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. 2017. Promising Compilation to ARMv8 POP. In *31st European Conference on Object-Oriented Programming (ECOOP 2017) (Leibniz International Proceedings in Informatics (LIPIcs))*, Peter Müller (Ed.), Vol. 74. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 22:1–22:28. https://doi.org/10.4230/LIPIcs.ECOOP.2017.22

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

Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2018. Simplifying ARM Concurrency: Multicopy-atomic Axiomatic and Operational Models for ARMv8. *Proc. ACM Program. Lang.* 2, POPL, Article 19 (Dec. 2018), 29 pages. https://doi.org/10.1145/3158107

Azalea Raad, Ori Lahav, and Viktor Vafeiadis. 2020. Persistent Owicki-Gries Reasoning: A Program Logic for Reasoning about Persistent Programs on Intel-X86. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 151 (Nov. 2020), 28 pages. https://doi.org/10.1145/3428219

Azalea Raad, Luc Maranget, and Viktor Vafeiadis. 2022a. Technical Appendix. (2022). https://www.soundandcomplete.org/papers/POPL2022/NT/appendix.pdf

Azalea Raad, Luc Maranget, and Viktor Vafeiadis. 2022b. X86_64 Memory Type Tests. (2022). http://diy.inria.fr/x86-memtype/reproduce.html

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

Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis. 2020. Persistency Semantics of the Intel-X86 Architecture. *Proc. ACM Program. Lang.* 4, POPL, Article 11 (Dec. 2020), 31 pages. https://doi.org/10.1145/3371079

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

Rust. 2021. (2021). https://doc.rust-lang.org/core/intrinsics/fn.nontemporal_store.html

Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. 2011. Understanding POWER Multiprocessors. *SIGPLAN Not.* 46, 6 (June 2011), 175âĂŞ186. https://doi.org/10.1145/1993316.1993520

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 (July 2010), 89–97. https://doi.org/10.
    1145/1785414.1785443

Dennis Shasha and Marc Snir. 1988. Efficient and correct execution of parallel programs that share memory. *ACM
    Transactions on Programming Languages and Systems (TOPLAS)* 10, 2 (1988), 282–312.

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

SPDK. 2021. (2021). https://spdk.io/

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

Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. 2014. GPS: Navigating Weak Memory with Ghosts, Protocols, and
    Separation. *SIGPLAN Not.* 49, 10 (Oct. 2014), 691âĂŞ707. https://doi.org/10.1145/2714064.2660243

Viktor Vafeiadis and Chinmay Narayan. 2013. Relaxed Separation Logic: A Program Logic for C11 Concurrency. In
    *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and
    Applications (OOPSLA '13)*. Association for Computing Machinery, New York, NY, USA, 867âĂŞ884. https://doi.org/10.
    1145/2509136.2509532