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

ACM Reference Format:

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.\footnote{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 $v$.}

\[
\begin{align*}
  &x := 1; \quad // 0 \\
  &a := y; \quad // 0 \\
  &b := x; \quad // 0 \\
  &y := 1; \quad (sb) \\
  &b := x; \quad // 0 \\
  &a := y; \quad // 1 \\
  &y := 1; \quad (mp)
\end{align*}
\]

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...
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 persistence, 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. 2020b] 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 Px86sim [Raad et al. 2020b] 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 cachability 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 or custom registers, and are thus used within system-level code, e.g. the Linux kernel [LWN 2008]. Moreover, the Linux kernel uses wc memory for e.g. frame buffer optimisation [LWN 2016] and 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. sb 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, flushopt, which constitutes a small fragment of Ex86. This is because on most modern hardware, including all machines available to us for testing, the flushopt 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 flushopt 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−] 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 \( \text{wb} \), \( \text{wt} \) 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 \( \text{wb}/\text{wt}/\text{wc} \) memory) indeed follow wc semantics, as intended.

\( \text{(nt)} \) "If a program specifies a non-temporal store and the memory type of the destination region is \( \text{wb} \), \( \text{wt} \) 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 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 \( t \in \{ \text{uc}, \text{wc}, \text{nc}, \text{wb}, \text{wt}, \text{c} \} \), 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 \( a \leftarrow x \) to denote reading (loading) the value of location \( x \) into the (thread-local) register \( a \). Analogously, we write \( x := \text{NT} v \) to denote storing \( v \) to location \( x \) with a non-temporal hint. As discussed above, the non-temporal hint is ignored if \( x \notin \text{Loc}_{\text{wb} \cup \text{wt} \cup \text{wc}} \).

**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
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 (✓† in Fig. 2).** As noted by Raad et al. [2020b], 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 \( P = a := x; sfence; c \) with \( c \neq sfence \), 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. 2020b], 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 \( x, y \in \text{Loc}_{\text{uc} \cup \text{wb} \cup \text{wt}} \) and \( x \in \text{Loc}_{\text{wc}} \lor y \in \text{Loc}_{\text{wc}} \). 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 \( x, y \in \text{Loc}_{\text{uc} \cup \text{wb} \cup \text{wt}} \), 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.

<table>
<thead>
<tr>
<th>( x, y \in \text{Loc} )</th>
<th>( x, y \in \text{Loc} )</th>
<th>( x \in \text{Loc}_{\text{wc}}, y \in \text{Loc} )</th>
<th>( x \in \text{Loc}_{\text{nc}}, y \in \text{Loc} )</th>
<th>( x \in \text{Loc}<em>{\text{wc}} \lor y \in \text{Loc}</em>{\text{c}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x := 1 ); ( \text{mfence}; ) ( y := 1 ); ( a := y ); ( b := x ); ( a = 1 \land b = 0 ): ✓</td>
<td>( x := 2 ); ( \text{mfence}; ) ( a := y ); ( b := x ); ( a = 1 \land x = 2 ): ✓</td>
<td>( y := 1 ); ( \text{mfence}; ) ( y := 1 ); ( a := y ); ( b := x ); ( a = 0 \land b = 0 ): ✓</td>
<td>( x := 1 ); ( \text{mfence}; ) ( y := 1 ); ( a := y ); ( b := x ); ( a = 0 \land b = 0 ): ✓</td>
<td>( x := 1 ); ( \text{mfence}; ) ( y := 1 ); ( a := y ); ( b := x ); ( a = 0 \land b = 0 ): ✓</td>
</tr>
<tr>
<td>(a) READ-READ</td>
<td>(b) READ-WRITE</td>
<td>(c) WRITE-READ</td>
<td>(d) WRITE-READ</td>
<td>(e) SFENCE-READ</td>
</tr>
</tbody>
</table>

![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.](image-url)
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 ∈ Loc_c) can be reordered before the earlier y:= 1 write regardless of the memory type of y (y ∈ 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 ∈ Loc_nc) cannot be reordered before y:= 1 (y ∈ 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 ∈ Locwb∪wt∪wc), then the outcomes shown will remain unchanged.

Earlier sfence and Later Non-Cacheable Reads (\(\forall c \in \text{Fig. 2}\)). 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; sfence; a:= x with c \(\not\triangleq\) sfence and x ∈ 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 ∈ 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, flushopt x and wb x. More concretely, given a location x in cache line (set of locations) X, written x ∈ X, executing flush x, flushopt x or wb x writes back the X cache line to memory. As described in [Intel 2021] and formalised by Raad et al. [2020b], persist instructions vary in strength (their constraints on instruction reordering) and performance: flush is the strongest of the three (enforcing additional ordering constraints), while flushopt and wb are equally weak with wb offering better performance than flushopt. That is, flushopt and wb have the same specification and exhibit equivalent behaviour; as such, as in [Raad et al. 2020b], we only model flush and flushopt. Nevertheless, all behaviours and specifications ascribed to flushopt 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 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

---

2In [Intel 2021], flush is referred to as CLFLUSH, flushopt is referred to as CLFLUSHOPT and wb is referred to as CLWB.
they could no longer be reordered and thus the weak behaviour $x=2 \land y=2$ could not be observed. Our tests in our validation effort have confirmed the strong ordering enforced by `flush` (see HERE).

By contrast, `flush_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_opt`, we could not observe the weak behaviours enabled by `flush_opt` except in a few cases. This is because the `flush_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_opt` as specified in the manual and reflected it in our Ex86 formalism, albeit without validating it. However, note that `flush_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 (K* in Fig. 2).** An earlier version of the Intel reference manual [Intel 2019] stated that `flush` and `flush_opt` on the same cache line are ordered with respect to one another, as later reflected in the Px86sim model of Raad et al. [2020b]. 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_opt`. As such, we have accordingly left out this constraint from our Ex86 model, thus minorly diverging from Px86sim.

**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_opt` instructions, respectively; and $NTW$ denote non-temporal writes. If two instructions are ordered (denoted by ✓), 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 ✗), 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 Px86sim model by Raad et al. [2020b]. We develop Ex86 by extending Px86sim 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-✓ entries), except those of `flush_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 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
Analogously, recall that writes are cached and propagated (persisted) to memory at a later time, thus following buffered persistency. Moreover, for better performance, cached writes reach (persist to) memory synchronously, at which point they also become visible to other threads (i.e. the memory has been previously formalised by Khyzha and Lahav 2021). All non-✓ entries (but those of NTW) are accompanied with links to litmus tests in our validation effort that witness the associated reordering.

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. [2020b] 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 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 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.

Fig. 2. The Ex86 ordering constraints where ✓ denotes that two instructions are ordered, X denotes they are not ordered (and thus may be reordered), and scl (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 X, respectively. The highlighted cells denote the Ex86 extensions from Px86sim [Raad et al. 2020b]. All non-✓ entries (but those of FO) are accompanied with links to litmus tests in our validation effort that witness the associated reordering.

![Table](https://example.com/table.png)

<table>
<thead>
<tr>
<th>Earlier in Program Order</th>
<th>Later in Program Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

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` and `flush_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 ⇒ 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′ ∈ X, flushopt 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 ∉ X, the flushopt 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 flushopt 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 flushopt x′. By contrast, recall that flushopt instructions are ordered with respect to (earlier and later) RMW, sfence and mfence instructions. As such, flushopt x′ in Fig. 3d cannot be reordered after FAA(y, 1) and thus y=1 ⇒ x=1 upon recovery. Similarly, flushopt x′ in Fig. 3e cannot be reordered after sfence and once again y=1 ⇒ 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) flushopt x′; cb, where cb 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 flushopt 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 ⇒ x=1 on recovery. That is, regardless of the y memory type (y∈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 ⇒ 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∈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 ∈ Locwc, executing x:=1 directly writes to memory; moreover, as y ∈ Locuc∪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 ∈ Locuc∪wt, the two writes cannot be reordered and executing x:=1 directly writes to memory, and thus y=1 ⇒ 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 ∈ Locwc and y ∈ Locwc∪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:=NT 1 such that x ∈ Locnt, then their recovery specifications would remain unchanged. Interestingly, a non-temporal store on x ∈ Locnt 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 ∈ Locnt\; \text{Specifically, (1)} as y ∈ Locnt∪wt and the first two writes are on the same location x, all three writes are ordered with respect to one another and thus no reordering is allowed; and (2) executing x:=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:=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 \texttt{sfence} (or \texttt{mfence}/\texttt{RMW}) before \( y:=1 \), as illustrated in Fig. 3, 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 \( C_P \) 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 \( t_1 \) to \( t_2 \) when \( t_2 \) reads a value written by \( t_1 \).

For instance, if the right thread in Fig. 4a reads 1 from \( y \) (written by \( C_P \) 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 \( C_P \)) 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 \( x^P \)).

By contrast, in the absence of persist \( x' \) in Fig. 4b, we have \( x:=1 \Rightarrow x \in \{0, 1\} \), as denoted by \( x \). This is because as \( x \in \text{Loc}_{ab} \), the \( x:=1 \) (in \( C \)) may persist after \( z:=1 \), even though \( x\) 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 \( y \in \text{Loc}_{ab} \), 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 \( y \notin \text{Loc}_{ab} \) (\( y \in \text{Loc}_{uc/wt} \)), 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 \( y^P \).

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 \( y^P \), 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 \( x \in \text{Loc}_{uc} \) and \( y \in \text{Loc}_{uc/wt} \), 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 (for wb persistency) which was developed in collaboration with (and endorsed by) Intel engineers [Raad et al. 2020b]. 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.

Verifications Techniques for Ex86 and PEx86. As we discuss later, the Ex86/PEx86 models are conservative extensions of the TSO/Px86sim models, respectively, in that for programs that do not use the additional features of Ex86/Px86sim (non-temporal writes and non-wb memory types), the behaviours of Ex86 and TSO (respectively PEx86 and Px86sim) coincide. As such, existing verification techniques for TSO/Px86sim 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. 2020a], 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

We describe a simple programming language for the Ex86 and PEx86 models in §3.1 and present the Ex86 declarative semantics in §3.2.

3.1 Preliminaries

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.

Derived memory types. We refer to the uc and wc types collectively as non-cacheable: \( nc \triangleq uc \cup wc \). Analogously, we refer to wt and wb collectively as cacheable types: \( c \triangleq wt \cup wb \).
that we assume that the types of locations cannot change once assigned.

\[ \forall (1) \text{ all locations in a cache line have the same memory type:} \]
\[ \forall (2) \text{ distinct cache lines are disjoint:} \]
\[ \forall \text{locations} \in \text{cache line} \]

\[ \text{constructs of expressions, assignments, conditionals and loops.} \]

\[ \text{expressions. We model a multi-threaded program} \]
\[ \text{containing registers and values.} \]

\[ \text{We use} \]
\[ \text{executions} \]

\[ \text{in the literature of declarative models, the traces of shared memory} \]
\[ \text{accesses generated by a program are commonly represented as a set of} \]
\[ \text{executions,} \]

\[ \text{Fig. 5. A simple concurrent programming language for Ex86 and PEx86} \]

\[ \text{Definition 2 (Locations and cache lines).} \]
\[ \text{Assume a set of memory addresses Addr.} \]
\[ \text{Assume a set of cache lines, CL} \]

\[ \text{Given a memory type t, we write LocC} \]
\[ \text{for LocC} \]

\[ \text{as meta-variables for locations in Loc; we use} \]
\[ \text{x, y,} \]
\[ \text{as meta-variables for cache lines.} \]

\[ \text{Programming Language.} \]
\[ \text{We employ a simple concurrent programming language as given in} \]
\[ \text{Fig. 5.} \]

\[ \text{We assume a finite set Scratch} \]
\[ \text{of registers (local variables);} \]
\[ \text{a finite set Val} \]
\[ \text{of values; and any standard interpreted language for expressions, Exp,} \]
\[ \text{containing registers and values.} \]
\[ \text{For a} \]
\[ \text{as a metavariable for values,} \]
\[ \text{for thread identifiers, and} \]
\[ \text{for expressions.} \]

\[ \text{We model a multi-threaded program P as a function mapping each thread to its} \]
\[ \text{se reveals a set of program points and} \]
\[ \text{instructions} \]

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

\[ \text{To aid readability, we may not follow syntactic conventions and write e.g. a := C in a, and} \]
\[ \text{let a := C1 in C2, where a is a fresh local variable. We write a := x for let a := load(x) in a; similarly for CAS/FAA. We write x := e and x := nt|e for store(x, e) and ntstore(x, e), respectively.} \]

\[ \text{3.2 Ex86: The Extended Intel-x86 Consistency Model} \]

\[ \text{Executions and Events.} \]

\[ \text{In the literature of declarative models, the traces of shared memory} \]
\[ \text{accesses generated by a program are commonly represented as a set of executions,} \]
\[ \text{where each} \]
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 ∈ PCom in Fig. 5) and is a tuple of the form e=(n, τ, l), where n ∈ N is the (unique) event identifier; τ ∈ TId is the thread identifier of the executing thread; and l ∈ Lab is the event label, defined below.

Definition 3 (Labels and events). An event e ∈ Event is a tuple ⟨n, τ, l⟩, where n ∈ N is an event identifier, τ ∈ TId is a thread identifier, and l ∈ 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;
- (NTW, xint, v) to denote non-temporally writing v to xint;
- (U, x, v, v′) to denote a successful update (RMW) modifying x to v′ when its value matches v;
- (U, x, v, ⊥) to denote a failed update (RMW) when x holds v, with ⊥ \notin Val;
- MF or SF to denote the execution of mfence or sfence, respectively; and
- (F0, x) or (FL, x) to denote the execution of flush_opt x or flush x, respectively.

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

The functions loc, valf, valy respectively project the location, the read value and the written value of a label, where applicable. For instance, \( loc(l)=x \) and \( val_f(l)=v \) for \( l=(W, x, v) \). 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 \( loc \), \( val_f \), and \( val_y \) to events, and given an event \( e \), we write \( eoc(e) \) for \( 1oc(lab(e)) \). An event \( e \) is an initialisation event iff \( tid(e)=0 \). 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 \( \{ a \in A \mid 1oc(a)=x \} \) and \( A_t \) for \( \{ e \in E \mid 1oc(e) \in Loc_t \} \), 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 \( r^r \), \( r^\tau \) 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. \( \{(a, a) \mid a \in A\} \); irreflexive \((r) \) for \( \exists a \sqsubseteq A. (a, a) \not\in r \); and acyclic \((r) \) for irreflexive \((r^*) \). We write \( r_1 \cup r_2 \) 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 \( \text{sloc} \triangleq \{(a, b) \in Event \times Event \mid 1oc(a) = 1oc(b) \} \), and the same-cache-line relation as \( \text{scl} \triangleq \{(a, b) \in Event \times Event \mid \exists c \in CL. 1oc(a) = X \land 1oc(b) = X \}. \)

Definition 4 (Ex86 executions). An execution, \( G \) ∈ Exec, is a tuple \((E, po, rf, mo)\), where:

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

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 \((E, po, rf, mo)\), 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-\(X\) 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, \(r \triangleq \bigcup \{ E^t \times (E^t \backslash F_0) \cup \{ r_t \} : r_t \subseteq E^t \times E^t \text{ is a strict total order on } E^t \} \). Given a program order \(po \subseteq E \times E\), the Ex86 ‘preserved-program-order’ on \(po\) is \(ppo(po) \triangleq (po \cap \text{ord})^*\), where ord is as defined in Fig. 6 with \(E^0 \triangleq \{ e \in E : \text{tid}(e) = 0 \}\).

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 \((E, po, rf, mo)\) is Ex86-consistent iff:

- \(mo_i \cup rf_j \cup rb_i \subseteq po\) (INTERNAL)
- \(\text{acyclic}(ob)\), where \(ob\) denotes the ‘ordered-before’ relation defined below: (EXTERNAL)

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 \(po_1 \times (W \times R)\) 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 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.
Thread transitions: \[ \text{COM} \xrightarrow{T\text{-LET}1} \text{COM} \]

\[ C_1 \xrightarrow{r_1} C'_1 \]

\[ \text{let } a := C_1 \text{ in } C_2 \xrightarrow{r_1} \text{let } a := C'_1 \text{ in } C_2 \]

Program transitions: \[ \text{PROG} \xrightarrow{T\text{-LET}2} \text{PROG} \]

\[ \text{let } a := v \text{ in } C \xrightarrow{r_2} C[v/a] \]

\[ \text{if } (C) \text{ then } C_1 \text{ else } C_2 \xrightarrow{r_1} \text{if } (C') \text{ then } C_1 \text{ else } C_2 \]

Definition 7 (SC, TSO and SPSO consistency). Given a memory model \( MM \in \{ \text{SC}, \text{TSO}, \text{SPSO} \} \), 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)
- acyclic(\( \text{ppo}_\text{MM} \cup \text{mo}_e \cup \text{rf}_e \cup \text{rb}_e \)) with:
  - \( \text{ppo}_\text{SC} \triangleq \text{po} \)
  - \( \text{ppo}_\text{TSO} \triangleq \text{po} \setminus (W \times R) \)
  - \( \text{ppo}_\text{SPSO} \triangleq \text{po} \setminus ((W \times W) \setminus \text{sloc}) \)

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

1. if \( \text{loc}(E) \subseteq \text{LOC}_{\text{SC}} \), then \( G \) is SC-consistent iff \( G \) is Ex86-consistent, where \( \text{loc}(E) \triangleq \{ \text{loc}(e) \mid e \in E \} \);
2. if \( \text{loc}(E) \subseteq \text{LOC}_{\text{TSO}} \), then \( G \) is TSO-consistent iff \( G \) is Ex86-consistent; and
3. if \( \text{loc}(E) \subseteq \text{LOC}_{\text{SC}} \), 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: \( C \xrightarrow{r} C' \), where \( C, C' \in \text{COM} \) (Fig. 5). The \( r \) marks the transition by recording the identifier of the executing thread \( r \), as well as the transition label \( l \), which may be \( \epsilon \) for no-ups, or in \( \text{Lab} \) (Def. 3) for primitive commands.

Most thread transitions are standard. The T-CAS0 transition describes the reduction of the \( \text{CAS}(x, v_1, v_2) \) 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 \( \text{CAS} \) when successful. The T-MF and T-SF transitions respectively describe executing an \( \text{mfence} \) and \( \text{sfence} \), reducing to value 1 (successful termination). Analogously, T-FO and T-FL describe the execution of persist instructions.
Program transitions are of the form: \( P \xrightarrow{rd} P' \), where \( P,P' \in \text{Proc} \) 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 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 \texttt{mfence} and RMW instructions to control this debuffering: executing an \texttt{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, \texttt{flush, flush\textsubscript{opt}} and \texttt{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 \texttt{po}, while the order they are removed (debuffered) corresponds to \texttt{ppo}. Specifically, since the TSO buffers only contain writes and write-write ordering is preserved under TSO \( (\texttt{po} \cap (W \times W) \subseteq \texttt{ppo\textsubscript{TSO}}) \), under TSO the order in which entries are added and removed 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, \texttt{flush, flush\textsubscript{opt}} and \texttt{sfence} entries, and since their respective orders are not included in \texttt{ppo}, the FIFO order and \texttt{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 \texttt{flush\textsubscript{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: \( M, B \xrightarrow{rd} M', B' \), where \( M, M' \in \text{Mem} \) denote the memory, modelled as a (finite) map from locations to values; and \( B, B' \in \text{BMap} \) denote the buffer map, associating each thread with its local buffer. Each local buffer \( b \in \text{Buff} \) is modelled as a sequence of BLAB, recording the labels of delayed instruction, namely those of writes, non-temporal writes, \texttt{flush, flush\textsubscript{opt}} and \texttt{sfence} instructions.

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 \( rd(M, b, x) \) in the premise of M-READC, defined at the bottom of Fig. 9.
**Ex86 Storage transitions:**

\[ \text{SConf} \xrightarrow{\text{TidLab} \cup \{ \epsilon \}} \text{SConf} \]

\[ \Sigma \in \text{SConf} \triangleq \text{MEM} \times \text{BMAP} \]

\[ M \in \text{MEM} \triangleq \text{Loc} \xrightarrow{\text{fin}} \text{Val} \]

\[ B \in \text{BMAP} \triangleq \text{Tid} \xrightarrow{\text{fin}} \text{Buff} \]

\[ x \in \text{Loc}_c \]

\[ r.d(M, B, (x, r), \epsilon) = \]

\[ M \xrightarrow{\tau(r, x, v), b} M, B \]

\[ B \in \text{BLAB} \]

\[ B(r) = b \]

\[ M, B \xrightarrow{b} M, B[\tau \mapsto b, l] \]

\[ B \in \text{Buff} \]

\[ B(r) = \epsilon \]

\[ M, B \xrightarrow{\tau, \text{MF}} M, B \]

\[ B(r) = b_1 \cdot l \cdot b_2 \]

\[ l \in \{(w, x, v), (m, x, v)\} \]

\[ b_1 \cap \text{LPPO}(B(r), \tau) = \emptyset \]

\[ M, B \xrightarrow{\tau} M[\tau \mapsto b_1, b_2] \]

\[ B \in \text{Mem} \]

\[ B(r) = b_1 \cdot l \cdot b_2 \]

\[ l \in \{(l, x), (f, x), \text{SF}\} \]

\[ b_1 \cap \text{LPPO}(B(r), \tau) = \emptyset \]

\[ M, B \xrightarrow{\tau} M, B[\tau \mapsto b_1, b_2] \]

\[ \text{rd}(M, b, x) \triangleq \]

\[ \begin{cases} 
\{v\} & \text{if } b_1, b_2, b = b_1, b_2 \land l \in \{(w, x, v), (m, x, v)\} \land \forall a'. (w, x, v'), (m, x, v') \notin b_2 \\
M(x) & \text{otherwise} 
\end{cases} \]

\[ \text{PO}(b, r) \triangleq \{(n_1, \tau, l_1), (n_2, \tau, l_2)\} | b^\#_{n_1} = l_1 \land b^\#_{n_2} = l_2 \land n_1 < n_2 \}

\[ \text{LPPO}(b, r) \triangleq \{(l_1, l_2) | (\neg, \neg, l_1), (\neg, \neg, l_2) \in \text{ppo}(\text{PO}(b, r))\} \]

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

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 (\(B(r) = \epsilon\)). 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 (\(M(x) = \nu\)).

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

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 \(r\) via \(\text{PO}(B(r), \tau)\), defined at the bottom of Fig. 9, where \(b^\#_{n}\) denotes the \(n\)th entry in \(b\). Given po on delayed entries, we then use LPPO(B(r), \(\tau\)) to compute ppo on delayed entries and lift it to their labels. That is, LPPO(B(r), \(\tau\)) captures ppo on delayed entries and thus their debuffering must respect LPPO(B(r), \(\tau\)), as stipulated by \(b_1 \cap \text{LPPO}(B(r), \tau) = \emptyset\): when \(B(r) = b_1 \cdot l_1 \cdot b_2\), 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, flushopt, and sfence.
Operational semantics: $\text{Prog} \times S\text{Conf} \Rightarrow \text{Prog} \times S\text{Conf}$

$\begin{align*}
\text{P} & \overset{\tau \in \text{SilentP}}{\mapsto} \text{P', S} \\
\text{P}, \Sigma & \Rightarrow \text{P', S, } \Sigma
\end{align*}$

$\begin{align*}
\text{P} & \overset{\tau \in \text{SilentS}}{\mapsto} \Sigma' \\
\text{P}, \Sigma & \Rightarrow \text{P, S, } \Sigma', \Sigma
\end{align*}$

$\begin{align*}
\text{P} & \overset{\tau \in \text{Step}}{\mapsto} \Sigma' \\
\text{P}, \Sigma & \Rightarrow \text{P, S, } \Sigma', \Sigma
\end{align*}$

Fig. 10. The operational semantics of Ex86 and PEx86

**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*$_{\text{opt}}, \text{sfence}$ 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 $6 \times 10^8$ 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 latter, our *klitmus* extension produces Linux kernel modules that call several Linux kernel functions, including those that change the memory-types of virtual pages.  

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

---

3A previous attempt of building over the kvm infrastructure failed because we found no technique to control memory-types in a virtualised setting.
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 \texttt{flush} instruction, showing that the \textit{implementation} of this relatively recent instruction is stronger than its \textit{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) \texttt{wb} to \texttt{uc} rules out non-SC behaviours; and (2) changing the memory type of a shared variable to \texttt{wc} relaxes (the default) TSO model by allowing write-write reordering (when changing the memory type of any of the two writes involved).

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

Observe that the read instructions cannot be reordered under Ex86, and the intervening \texttt{mfence} merely emphasises the absence of such read-read reordering. Note that were we to remove the \texttt{flush}\textsubscript{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 ($a=1 \land b=0$) could be observed. On the other hand, were we to replace the \texttt{flush}\textsubscript{opt} in each example with a corresponding \texttt{flush}, such write-write reordering would be prohibited (\texttt{flush} instructions are ordered against \textit{all} earlier and later writes) and the weak behaviour shown would not be observed.

By contrast, our experiments confirmed that adding an intervening \texttt{flush}\textsubscript{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 the weak behaviour shown is due to write-\texttt{flush}\textsubscript{opt} reordering (i.e. reordering $x := 1$ and \texttt{flush}\textsubscript{opt}$y$) or \texttt{flush}\textsubscript{opt}-write reordering (i.e. reordering \texttt{flush}\textsubscript{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-\texttt{flush}\textsubscript{opt} reordering from \texttt{flush}\textsubscript{opt}-write reordering. As such, since (1) the implementation of \texttt{flush}\textsubscript{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 \texttt{flush\textsubscript{opt}} is observed we cannot conclusively determine the reordering direction, in Ex86 we opt to model all weak behaviours of \texttt{flush\textsubscript{opt}} as described in the Intel manual.

\textbf{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 \texttt{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

\textbf{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 \texttt{x:=v} may be observed upon recovery if the write of \texttt{v} on \texttt{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 \texttt{x:=v} persist, its associated label is deemed durable. By contrast, \texttt{mfence}, \texttt{sfence} and read instructions have no durable effects and their events are thus not durable.

\textbf{Definition 8 (Durable events).} The set of durable events is: \( D = ST \).

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

- \( E, po, rf \) and \( mo \) are as defined in Def. 4.
- \( P: Loc \rightarrow E \cap D \) associates each location \( x \) with a durable event on \( x \) (i.e. \( \forall x. 1oc(P(x)) = x \)).
- \( pf \subseteq (E \cap ST) \times (E \cap (FL \cup FO)) \) denotes the ‘persists-from’ relation on events of the same cache line: \((a, b) \in pf \Rightarrow (a, b) \in scl\). Moreover, for each event \( e \in FL_x \cup FO_x \), and each location \( x' \) on the same cache line as \( x \) (i.e. \( (x, x') \in scl \)), \( pf \) 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 \texttt{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 \( pb \triangleq pf^{-1}; mo \), relating each flush/flush-opt to \texttt{mo}-later stores (cf. \texttt{rb}).

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

- \( G \) satisfies the \textbf{internal} axiom in Def. 6.
- \textbf{acyclic} (ob), where \( ob \) is extended from Def. 6 as follows:

\begin{align*}
\textbf{ob} & \triangleq po \cup mo_e \cup rf_e \cup rb_e \cup pf \cup pb \\
\forall x \in Loc_{nc \cup wt}. P(x) & = \max (mo_x) \quad \text{(strict-persist)} \\
\forall x \in Loc_{ab}, e \in S_x, (e, P(x)) & \in mo_e \quad \text{(weak-persist)}
\end{align*}

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

As with Ex86, the PEx86 axioms ensure internal and external consistency, with \( ob \) extended with persistency relations \( pf \) and \( pb \) in \textbf{EXTERNAL-REVISED}. The \textbf{strict-persist} describes the strict, unbuffered persistency of \texttt{nc} and \texttt{wt} memory; the last store persisted for \( x \in Loc_{nc \cup wt} \) 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 $x \in \text{Loc}_{wb}$ must be mo-after all other stores to $x$ that are guaranteed to have persisted, namely (1) non-temporal writes on wb memory ($\text{NTW}_{wb}$) since non-temporal writes follow strict, unbuffered persistency; and (2) wb writes persisted by a persist sequence, i.e. either a flush ($\text{dom}(\text{pf}; [\text{FL}])$) or a flush$_{opt}$ followed by mfence/sfence/RMW ($\text{dom}($pf$; [\text{FO}]$); po; $[\text{MF} \cup \text{SF} \cup U]$).

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, 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 stores 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$_{opt}$ 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 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 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 \texttt{mfence}/RMWs renders \texttt{flush}_{\text{opt}} instructions synchronous, they may be executed only if there are no pending \texttt{flush}_{\text{opt}} entries by the executing thread in the persistence buffer. Similarly, an \texttt{sfence} entry may be removed from the local buffer (i.e. executed) only when there are no pending \texttt{flush}_{\text{opt}} entries by the executing thread.

**PE\textsubscript{Ex86} Storage Transitions.** The storage transitions are of the form $M, PB, B \xrightarrow{r_d} M', PB', B'$, where $M, M' \in \text{MEM}$ and $B, B' \in \text{BMAP}$ as are in Fig. 9, and $PB, PB' \in \text{PBMAP}$ denotes the per-location persistence buffer map, associating each \texttt{wb} location with its persistence buffer. A persistence buffer, $pb \in \text{PBUFF}$, is modelled as a sequence of entries of the form $(1)$ \texttt{w}(\nu), denoting a persist pending write with value $\nu$ on $x$; or $(2)$ \texttt{fo}(\tau), denoting a pending \texttt{flush}_{\text{opt}} by thread $\tau$ on $x$.

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

As described above, executing \texttt{mfence}/RMWs by $\tau$ in $M$-$\text{MF}$, \textsc{M-RmwS1}, \textsc{M-RmwF1} and \textsc{M-RmwF2} additionally ensures that there are no pending \texttt{flush}_{\text{opt}} entries by $\tau$ on any location in the persistence buffer: $\forall y. \texttt{fo}(\tau) \notin PB(y)$. Note that \textsc{M-RmwS1} captures executing a successful RMW on a non-\texttt{wb} location $(x \notin \text{Loc}_{\text{wb}})$, where the update is written directly to memory. By contrast, \textsc{M-RmwS2} denotes executing a successful RMW on a \texttt{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 $\texttt{rd}(M, PB(x), \epsilon, x)$. Analogously, \textsc{M-RmwF1} and \textsc{M-RmFS2} capture executing a failed RMW on a non-\texttt{wb} and \texttt{wb} location, respectively.

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

\textsc{M-PropFL} describes debuffering (\texttt{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 $x$: $\forall y. (x, y) \in \text{sc} \Rightarrow PB(y) = \epsilon$. Analogously, \textsc{M-PropFO} describes debuffering (\texttt{fo}, $x$) by thread $\tau$, appending $\texttt{fl}(\tau)$ to $PB(y)$ for each $y$ in the cache line of $x$, as denoted by the standard ternary conditional operator $(\sim\sim\sim)$ as $PB' = \lambda y. (y, x) \in \text{sc} \Rightarrow PB(y).fo(\tau) : PB(y)$. \textsc{M-PropSF} describes the debuffering of \texttt{sfence} entries by $\tau$, ensuring that there are no pending $\texttt{fo}(\tau)$ on any location in the persistence buffer.

Lastly, \textsc{M-PersistW} and \textsc{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 \texttt{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.

**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

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. [2020b] 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. 2020a; 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.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their feedback. This work was supported in part by a UKRI Future Leaders Fellowship [grant number MR/V024299/1], and by a European Research Council (ERC) Consolidator Grant for the project “PERSIST” under the European Union’s Horizon 2020 research and innovation programme (grant agreement No. 101003349).

REFERENCES


Extending Intel-x86 Consistency and Persistency


Machinery, New York, NY, USA, 362–376. https://doi.org/10.1145/3385412.3386010
LWN. 2007. https://lwn.net/Articles/255364/
LWN. 2008. https://lwn.net/Articles/282250/
LWN. 2016. https://lwn.net/Articles/698014/