Persistency Semantics of the Intel-x86 Architecture

AZALEA RAAD, MPI-SWS, Germany
JOHN WICKERSON, Imperial College London, UK
GIL NEIGER, Intel Labs, US
VIKTOR VAFEIADIS, MPI-SWS, Germany

Emerging non-volatile memory (NVM) technologies promise the durability of disks with the performance of RAM. To describe the persistency guarantees of NVM, several memory persistency models have been proposed in the literature. However, the persistency semantics of the ubiquitous x86 architecture remains unexplored to date. To close this gap, we develop the *Px86* ('*persistent x86*') model, formalising the persistency semantics of Intel-x86 for the *first time*. We formulate Px86 both operationally and declaratively, and prove that the two characterisations are *equivalent*. To demonstrate the application of Px86, we develop two *persistent libraries* over Px86: a persistent transactional library, and a persistent variant of the Michael–Scott queue. Finally, we encode our declarative Px86 model in Alloy and use it to generate persistency litmus tests *automatically*.

CCS Concepts: • **Theory of computation → Concurrency**; **Semantics and reasoning**.

Additional Key Words and Phrases: weak memory, memory persistency, non-volatile memory, Intel-x86

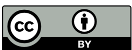
ACM Reference Format:
Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis. 2020. Persistency Semantics of the Intel-x86 Architecture. *Proc. ACM Program. Lang.* 4, POPL, Article 11 (January 2020), 31 pages. https://doi.org/10.1145/3371079

1 INTRODUCTION

Emerging non-volatile memory (NVM) technologies [Kawahara et al. 2012; Lee et al. 2009; Strukov et al. 2008] provide byte-level access to data guaranteed to persist beyond a power failure at performance comparable to regular (volatile) RAM. It is widely believed that NVM (a.k.a. persistent memory) will eventually supplant volatile memory, allowing efficient access to persistent data [Intel 2014; ITRS 2011; Pelley et al. 2014]. This has led to a surge in NVM research in recent years [Boehm and Chakrabarti 2016; Chakrabarti et al. 2014; Chatzistergiou et al. 2015; Coburn et al. 2011; Gogte et al. 2018; Izraelevitz et al. 2016a; Kolli et al. 2017, 2016a,b; Nawab et al. 2017; Raad and Vafeiadis 2018; Raad et al. 2019b; Volos et al. 2011; Wu and Reddy 2011; Zhao et al. 2013; Zuriel et al. 2019].

Using persistent memory *correctly*, however, is not easy. A key challenge is ensuring correct recovery after a crash (e.g. a power failure) by maintaining the consistency of data in memory, which requires an understanding of the order in which writes are propagated to memory. The problem is that CPUs are not directly connected to memory; instead there are multiple volatile caches in between. As such, writes may not propagate to memory at the time and in the order that the processor issues them, but rather at a later time and in the order decided by the cache coherence protocol. This can lead to surprising outcomes. For instance, consider the simple sequential program

Authors' addresses: Azalea Raad, MPI-SWS, Saarland Informatics Campus, Germany, azalea@mpi-sws.org; John Wickerson, Imperial College London, UK, j.wickerson@imperial.ac.uk; Gil Neiger, Intel Labs, US, gil.neiger@intel.com; Viktor Vafeiadis, MPI-SWS, Saarland Informatics Campus, Germany, viktor@mpi-sws.org.



This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2020 Copyright held by the owner/author(s).
2475-1421/2020/1-ART11
https://doi.org/10.1145/3371079

$x := 1; y := 1$, running to completion and crashing thereafter. On restarting the computer, the memory may contain $y{=}1, x{=}0$; i.e. the $x := 1$ write may not have propagated to memory before the crash.

To ensure correct recovery, one must understand and control the order in which writes are committed to persistent memory. To this end, Pelley et al. [2014] introduced *persistency models* to define the persistency semantics of programs (i.e. their permitted behaviours upon recovery) by prescribing the order in which writes are persisted to memory. Existing literature includes several proposals of persistency models varying in strength and performance [Condit et al. 2009; Gogte et al. 2018; Izraelevitz et al. 2016b; Joshi et al. 2015; Kolli et al. 2017, 2016b; Raad and Vafeiadis 2018; Raad et al. 2019b]. However, none of these works explore the persistency semantics of the mainstream Intel-x86 architecture [Intel 2019]. This is a significant gap not only because Intel-x86 hardware is ubiquitous, but also because Intel are one of the forerunners in supporting and developing NVM technologies and infrastructures. In particular, Intel-x86 processors provide extensive support for persistent programming via numerous persistence primitives [Intel 2019]. Further, Intel have recently manufactured their own line of NVM technology: Optane 3D [Intel 2019]. Intel are also behind large-scale projects such as PMDK (the persistent memory development kit), which provides a collection of libraries and tools for persistent programming [Intel 2015].

To close this gap, we formalise the persistency semantics of the Intel-x86 architecture. More concretely, we develop the *Px86* ('*persistent x86*') model by extending the x86-TSO (weak) memory model [Sewell et al. 2010] with the Intel-x86 persistency semantics as described informally in the Intel reference manual [Intel 2019]. We developed Px86 in close collaboration with research engineers at Intel. During our formal development, we discovered several ambiguities in the Intel reference manual. Following our discussions with Intel engineers, we were informed that the manual text is under-specified in that it allows for more behaviours than intended, underlining the ever-growing need for *formal* hardware specifications immune to misinterpretation. To remain faithful to the manual text and to capture the behaviour intended by the Intel engineers, we thus develop two models: (1) the weaker Px86_{man} model which is faithful to the manual; and (2) the stronger Px86_{sim} model which is a simplification of Px86_{man} and reflects the intended behaviour. We write Px86 when the distinction between Px86_{sim} and Px86_{man} is immaterial. We formally describe the Px86_{sim} and Px86_{man} models both *operationally* and *declaratively*, and prove that the two Px86_{sim} (resp. Px86_{man}) characterisations are equivalent. Whilst the operational models provide a more intuitive account of the hardware guarantees, the declarative models streamline correctness proofs such as those by Herlihy and Wing [1990] and Owens [2010]. We have encoded our Px86_{sim} and Px86_{man} models in the Alloy modelling language [Jackson 2012].

To showcase the application of our Px86 semantics, we present two persistent libraries implemented in Px86. First, we present a library for *persistent transactions*. Transactions are a powerful high-level abstraction for concurrency control, readily available to programmers in the volatile setting. Persistent transactions extend volatile transactions by additionally providing persistency control, thus streamlining the task of writing correct persistent programs. Our library makes persistent programming on Px86 more accessible to the uninitiated programmer as it liberates them from understanding the low-level details of Px86. We then show that our library is correct in that it guarantees *persistent serialisability* [Raad et al. 2019b], which is a strong transactional consistency guarantee in the NVM context. Second, we present a persistent variant of the Michael–Scott queue library [Michael and Scott 1996], and show that our implementation is correct in that it ensures *persistent linearisability* [Izraelevitz et al. 2016b], which is the persistent extension of the well-known notion of library correctness: linearisability [Herlihy and Wing 1990].

Finally, we use our Alloy encoding of Px86 to generate persistency litmus tests *automatically*. This process guided our formalism of Px86, allowing us to identify its corner cases. We believe that our persistency litmus tests are beneficial to those seeking to understand and program over Px86.

Contributions and Outline. Our contributions (detailed in §2) are as follows: (1) in §3 and §4 we respectively develop the operational $Px86_{sim}$ and $Px86_{man}$ models as the first formal models of the Intel-x86 persistency semantics; (2) in §5 we present the declarative $Px86_{sim}$/$Px86_{man}$ models and show that the two $Px86_{sim}$ (resp. $Px86_{man}$) characterisations are equivalent; (3) in §6 we discuss our persistent transactional library and our persistent Michael–Scott queue library implemented in Px86, and show that our implementations are correct; (4) in §7 we describe how we encode Px86 in Alloy to generate persistency litmus tests automatically. We discuss related and future work in §8.

Additional Material. The full proofs of all stated theorems in the paper are given in the accompanying technical appendix available at http://plv.mpi-sws.org/pmem/. We provide machine-readable versions of $Px86_{sim}$ and $Px86_{man}$ models in the Alloy modelling language [Jackson 2012].

2 OVERVIEW

2.1 Persistency Semantics

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 [Pelley et al. 2014] describe the permitted behaviours of programs upon recovering from a crash (e.g. due to a power failure) 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. Existing persistency models [Gogte et al. 2018; Izraelevitz et al. 2016b; Kolli et al. 2017; Pelley et al. 2014; Raad and Vafeiadis 2018; Raad et al. 2019b] can be categorised along two axes: (1) strict versus relaxed; (2) unbuffered versus buffered.

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. This makes reasoning about persistency simpler, but hinders performance as it introduces unnecessary dependencies between persists. To remedy this, *relaxed persistency* models allow for volatile and persistent memory orders to disagree. As we describe shortly, the Intel-x86 architecture follows a relaxed persistency model.

The second dichotomy concerns when persists occur. Under unbuffered persistency, persists occur *synchronously*: when executing an instruction, its effects are immediately committed to persistent memory; i.e. execution is stalled by persists. To improve performance, *persist buffering* allows memory persists to occur *asynchronously* [Condit et al. 2009; Izraelevitz et al. 2016b; Joshi et al. 2015], where memory persists are buffered in a queue to be committed to persistent memory at a future 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, after recovering from a crash, only a *prefix* of the persistent memory order may have successfully persisted. As we describe shortly, the Intel-x86 architecture follows a buffered persistency model [Intel 2019]. When it is necessary to control the committing of buffered persists explicitly (e.g. before performing I/O), buffered models typically offer explicit *persist* instructions (with varying granularity) that drain the relevant pending persists from the persistent buffer and commit them to persistent memory. For instance, the epoch persistency model provides a *sync* instruction which commits *all* pending writes, while Intel-x86 provides *per-cache-line persist* instructions which commit all pending writes on a *given cache line* (a set of memory locations). Moreover, explicit persist may themselves execute *synchronously* or *asynchronously*. For instance, the sync instruction in epoch persistency executes synchronously: it blocks until all pending writes are persisted. By

contrast, the per-cache-line persist instructions of Intel-x86 execute asynchronously: they do not stall execution and merely guarantee that the cache line will be persisted at a future time.

2.2 The Px86 Model: An Intuitive Account

As discussed in §1, we formalise the persistency semantics of the Intel-x86 architecture via two models: the weaker Px86_{man} model which is faithful to the manual, and the stronger Px86_{sim} model which is a simplification of Px86_{man} and captures the behaviour intended by Intel engineers. As we discuss shortly, the Px86_{sim} and Px86_{man} models coincide in most cases and only diverge on their ordering constraints between read and persist instructions. As such, we write Px86 when the distinction between Px86_{sim} and Px86_{man} is inconsequential and the described behaviour holds under both Px86_{man} and Px86_{sim}. We proceed with an intuitive description of Px86; we then discuss the differences between Px86_{sim} and Px86_{man} in §2.3. Later we describe the Px86_{sim} and Px86_{man} semantics formally both *operationally* (§3 and §4) and *declaratively* (§5), and prove that the two Px86_{sim} (resp. Px86_{man}) characterisations yield *equivalent* semantics.

The Px86 Model. The Intel-x86 architecture follows a relaxed, buffered persistency model. The buffered persistency of Px86 is reflected in the example of Fig. 1a. Due to the buffered model of Px86, 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. However, the relaxed nature of the Px86 model allows for 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 writes cannot be reordered under Intel-x86. Nevertheless, in case of a crash it is possible under Px86 to observe $x{=}0$, $y{=}1$ after recovery. This is due to the relaxed persistency of Px86: 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). Under the Px86 model the 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 location, its store and persist orders coincide.

In order to afford more control over when pending writes are persisted, Intel-x86 provides explicit *persist* instructions, **flush** x, $\textbf{flush}_{\text{opt}}$ x and **wb** x, that *asynchronously* persist all pending writes on all locations in the cache line of x [Intel 2019].[1] That is, when location x is in cache line X, written $x \in X$, an explicit persist on x persists all pending writes on all locations $x' \in X$. Persist instructions vary in strength (in terms of their constraints on instruction reordering) and performance. More concretely, **flush** is the strongest of the three enforcing additional ordering constraints. On the other hand, $\textbf{flush}_{\text{opt}}$ and **wb** are *equally* weak with varying performance. In other words, $\textbf{flush}_{\text{opt}}$ and **wb** have the same specification and exhibit equivalent behaviour; however, **wb** provides better performance than $\textbf{flush}_{\text{opt}}$. This is because when $x \in X$, although executing both instructions persists the X cache line, $\textbf{flush}_{\text{opt}}$ x invalidates X, while **wb** x may retain X. Note that retaining X is *not guaranteed* when executing **wb** x; rather, X *may* be retained. As such, both **wb** and $\textbf{flush}_{\text{opt}}$ follow the same specification, while **wb** may improve performance in certain cases:

> 'For usages that require only writing back modified data from cache lines to memory (do not require the line to be invalidated), and expect to subsequently access the data, software is recommended to use **wb** (with appropriate fencing) instead of $\textbf{flush}_{\text{opt}}$ or **flush** for improved performance.' [Intel 2019, p. 3-149]

[1] In [Intel 2019] **flush** is referred to as **CLFLUSH** or 'cache line flush'; $\textbf{flush}_{\text{opt}}$ is referred to as **CLFLUSHOPT** or 'cache line flush optimized' and **wb** is referred to as **CLWB** or 'cache line write back'.

$x := 1;$ $y := 1;$ (a)	$x := 1;$ $\textbf{flush}_{\text{opt}}\ x';\dagger$ $\textbf{FAA}(y, 1);$ (b)	$x := 1;$ $\textbf{flush}_{\text{opt}} x';$ $y := 1;$ (c)	$x := 1;$ $\textbf{flush}\ x';$ $y := 1;$ (d)
rec: $x, y \in \{0, 1\}$	rec: $y{=}1 \Rightarrow x{=}1$	rec: $x, y \in \{0, 1\}$	rec: $y{=}1 \Rightarrow x{=}1$
$x := 1;$ $\textbf{flush}_{\text{opt}}\ x';\dagger$ $\textbf{sfence};$ $y := 1;$ (e)	$x := 1;$ $x := 2;$ $\textbf{flush}_{\text{opt}}\ x';\dagger$ $\textbf{sfence};$ $y := 1;$ (f)	$x := 1;$ $\textbf{flush}_{\text{opt}}\ x';\dagger$ $\textbf{sfence};$ $y := 1;$ ‖ $\mathtt{a} := y;$ $\textbf{if}\ \mathtt{a}\ \textbf{then}$ $z := 1;$ (g)	$x := 1;$ $\textbf{flush}_{\text{opt}}\ x;$ $x := 2;$ ‖ $\mathtt{a} := x;$ $\textbf{if}\ \mathtt{a}{=}2\ \textbf{then}$ $y := 1;$ (h)
rec: $y{=}1 \Rightarrow x{=}1$	rec: $y{=}1 \Rightarrow x{=}2$	rec: $z{=}1 \Rightarrow (x{=}1 \wedge y \in \{0, 1\})$	rec: $y{=}1 \Rightarrow x \in \{0, 1, 2\}$

Fig. 1. Examples of Px86 programs and possible values of x, y, z upon recovery; in all examples x, y, z are locations in persistent memory, a is a (local) register, $x, x' \in X$ (x, x' are in cache line X), $y, z \notin X$, and initially $x{=}y{=}z{=}0$. The $\dagger$ annotation denotes that replacing $\textbf{flush}_{\text{opt}}$ with **flush** yields the same result after recovery.

As $\textbf{flush}_{\text{opt}}$ and **wb** have equivalent specifications, in the remainder of this article and in our formal development we only model **flush** and $\textbf{flush}_{\text{opt}}$. However, all behaviours and specifications ascribed to $\textbf{flush}_{\text{opt}}$ are also attributed to **wb**. We next describe the behaviour of **flush** and $\textbf{flush}_{\text{opt}}$ and their ordering constraints via several examples.

The persist behaviour of $\textbf{flush}_{\text{opt}}$ and **flush** is illustrated in Fig. 1b (where **FAA** denotes an atomic 'fetch-and-add' instruction): executing $\textbf{flush}_{\text{opt}}\ x'$ persists the earlier write on X (i.e. $x := 1$) to memory. As such, if a crash occurs during the execution of this program and upon recovery $y{=}1$, then $x{=}1$. That is, if $\textbf{FAA}(y, 1)$ has executed and persisted before the crash, then so must the earlier $x := 1; \textbf{flush}_{\text{opt}}\ x'$. Note that this behaviour is guaranteed thanks to the ordering constraints on $\textbf{flush}_{\text{opt}}$ and **flush** instructions. More concretely, both $\textbf{flush}_{\text{opt}}$ and **flush** instructions are ordered with respect to *earlier* (in program order) write instructions on the same cache line, as well as all (both earlier and later in program order) atomic read-modify-write (RMW) instructions regardless of the cache line. Hence, $\textbf{flush}_{\text{opt}}\ x'$ in Fig. 1b cannot be reordered with respect to the $x := 1$ write or the RMW instruction **FAA**. As such, upon recovery $y{=}1 \Rightarrow x{=}1$. Note that $\textbf{flush}_{\text{opt}}\ x$/**flush** x instructions persist the X cache line *asynchronously*: their execution does not block until X is persisted; rather, the execution may proceed and X is made persistent at a future point.

Note that although persist instructions execute asynchronously, together with ordering constraints they impose a particular *persist order*. Given $x \in X$ and C=$\textbf{flush}_{\text{opt}}\ x$ or C=**flush** x, all writes on X ordered before C persist before all instructions ordered after C, regardless of their cache line. For instance, since $x := 1$ in Fig. 1b is ordered before $\textbf{flush}_{\text{opt}}\ x$ and $\textbf{flush}_{\text{opt}}\ x$ is ordered before $\textbf{FAA}(y, 1)$, $x := 1$ is guaranteed to persist before $\textbf{FAA}(y, 1)$.

However, certain instruction reorderings under Px86 mean that persist instructions may not execute at the intended program point and thus may not guarantee the intended persist ordering. For instance, when $x' \in X$, the execution of $\textbf{flush}_{\text{opt}}\ x'$ is only ordered with respect to earlier writes on the same cache line X and thus may be reordered with respect to writes on locations in different cache lines, i.e. those not in X. This is illustrated in the example of Fig. 1c. Note that the program in Fig. 1c is obtained from Fig. 1b by replacing the RMW instruction **FAA** on y with a plain write. Since $y \notin X$, the execution of $\textbf{flush}_{\text{opt}}\ x'$ in Fig. 1c 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

$\textbf{flush}_{\text{opt}}\ x'$ has executed, then upon recovery it is possible to observe $x{=}0$ and $y{=}1$. That is, there is no guarantee that $x := 1$ persists before $y := 1$, *despite* the intervening persist instruction $\textbf{flush}_{\text{opt}}\ x'$.

The main difference between $\textbf{flush}_{\text{opt}}$ and **flush** lies in their ordering constraints with respect to writes. More concretely, the execution of $\textbf{flush}_{\text{opt}}\ x'$ is ordered *only* with respect to earlier writes on the same cache line, whereas the execution of $\textbf{flush}\ x'$ is ordered with respect to *all* (both earlier and later) writes regardless of the cache line. This is illustrated in the example of Fig. 1d, obtained from Fig. 1c by replacing $\textbf{flush}_{\text{opt}}$ with **flush**. Unlike in Fig. 1c, $\textbf{flush}\ x'$ in Fig. 1d is ordered with respect to $y := 1$ and cannot be reordered after it. Hence, if after recovery $y{=}1$ ($y := 1$ has executed and persisted), then $x{=}1$ ($x := 1; \textbf{flush}\ x'$ must have also executed and persisted). In particular, while $x{=}0 \land y{=}1$ is not possible upon recovery in Fig. 1d, it is possible in Fig. 1c.

In order to afford more control over the order in which writes on different locations are persisted, 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*. That is, $\textbf{flush}_{\text{opt}}$ and **flush** cannot be reordered with respect to fences. This is illustrated in the program of Fig. 1e, obtained from that of Fig. 1c by introducing an **sfence** after the persist instruction. Although in the case of Fig. 1c it is possible under Px86 to observe $y{=}1 \land x{=}0$ upon recovery as discussed above, the introduction of **sfence** in Fig. 1e ensures that the write on x persists before that on y and thus upon recovery $y{=}1 \Rightarrow x{=}1$. More concretely, **sfence** cannot be reordered before or after any of the instructions, and $\textbf{flush}_{\text{opt}}\ x'$ cannot be reordered before the earlier (in program order) write on x. As such, if upon recovery $y{=}1$ (i.e. $y := 1$ has executed and persisted prior to the crash), then $x{=}1$ (i.e. $x := 1; \textbf{flush}_{\text{opt}}\ x'$ has also executed and persisted).

Note that the writes on the same location are persisted in the execution order. That is, for each location x, the store and persist orders on x coincide. This is illustrated in the example of Fig. 1f, obtained from Fig. 1e by inserting a second write on x before the persist instruction. As in Fig. 1e, the instructions in Fig. 1f cannot be reordered. As such, if upon recovery $y{=}1$ and thus $y := 1$ has executed and persisted before the crash, then the earlier $x := 1; x := 2; \textbf{flush}_{\text{opt}}\ x'$ must have also executed and persisted. Moreover, $x := 1$ and $x := 2$ are executed and persisted in the same order (i.e. $x := 1$ before $x := 2$) and thus $y{=}1 \Rightarrow x{=}2$ upon recovery, and it is not possible to observe $y{=}1, x{=}1$.

The examples discussed thus far all concern sequential programs and the persist orderings on the writes in the *same* thread. The example in Fig. 1g illustrates how persist orderings can be imposed on the writes of *different* threads. Note that the program in the left thread of Fig. 1g is that of Fig. 1e. Under Px86 one can use *message-passing* between threads to ensure a certain persist ordering. A message is passed from thread τ_1 to τ_2 when τ_2 reads a value written by τ_1. For instance, if the right thread in Fig. 1g reads 1 from y (written by the left thread), then the left thread passes a message to the right thread. Under the Intel-x86 architecture message passing ensures that the instruction writing the message (e.g. $y := 1$) is executed (ordered) before the instruction reading it (e.g. $\mathtt{a} := y$). As such, since $x := 1; \textbf{flush}_{\text{opt}}\ x'$ is executed before $y := 1$ (as in Fig. 1e), $y := 1$ is executed before $\mathtt{a} := y$, and $z := 1$ is executed after $\mathtt{a} := y$ when $\mathtt{a}{=}1$, we know $x := 1; \textbf{flush}_{\text{opt}}\ x'$ is executed before $z := 1$. Consequently, if upon recovery $z{=}1$ (i.e. $z := 1$ has persisted before the crash), then $x{=}1$ ($x := 1; \textbf{flush}_{\text{opt}}\ x'$ must have also persisted before the crash). Note that by contrast $z{=}1 \Rightarrow y \in \{0, 1\}$. This is because $y := 1$ may persist after $z := 1$. As such, if a crash occurs after $z := 1$ has executed and persisted but before $y := 1$ has persisted, it is possible to observe $y{=}0, z{=}1$ after recovery, *even though* $y{=}0, z{=}1$ is never possible during normal (non-crashing) executions.

Recall that $\textbf{flush}_{\text{opt}}$ instructions are ordered only with respect to earlier (in program order) writes on the same cache line and thus may be reordered with respect to later writes on any cache line. This is illustrated in the example of Fig. 1h: $\textbf{flush}_{\text{opt}}\ x$ in the left thread may be reordered after $x := 2$. As such, even though at no point during the normal execution of the program the $y{=}1, x{\neq}2$

	(a)	(b)	(c)
	$x := 1;$ $y := 1;$ ‖ $\mathtt{a} := y;$ **flush** x; **if** a **then** $z := 1;$	$x := 1;$ $y := 1;$ ‖ $\mathtt{a} := y;$ **sfence**; **flush** x; **if** a **then** $z := 1;$	$x := 1;$ $y := 1;$ ‖ $\mathtt{a} := y;$ **mfence**; **flush** x; **if** a **then** $z := 1;$
Px86_{man}	rec: $z{=}1 \Rightarrow x, y \in \{0, 1\}$	rec: $z{=}1 \Rightarrow x, y \in \{0, 1\}$	rec: $z{=}1 \Rightarrow x{=}1 \land y \in \{0, 1\}$
Px86_{sim}	rec: $z{=}1 \Rightarrow x{=}1 \land y \in \{0, 1\}$	rec: $z{=}1 \Rightarrow x{=}1 \land y \in \{0, 1\}$	rec: $z{=}1 \Rightarrow x{=}1 \land y \in \{0, 1\}$

Fig. 2. Examples programs and possible values of x, y, z upon recovery under Px86_{man} and Px86_{sim}; in all examples x, y, z are locations in persistent memory, a is a (local) register, initially $x{=}y{=}z{=}0$, $x \in X$ and $y, z \notin X$.

behaviour is observable, it is possible to observe $y{=}1, x{\neq}2$ after recovery as follows: first $x := 1$ and $x := 2$ in the left thread are executed (but not persisted), then the instructions in the right thread are executed and $y := 1$ is persisted, and finally the program crashes before $\textbf{flush}_{\text{opt}}$ x is executed. By contrast, since **flush** cannot be reordered with respect to any writes, replacing $\textbf{flush}_{\text{opt}}$ in Fig. 1h with the stronger **flush** prohibits this behaviour and ensures $y{=}1 \Rightarrow x{\neq}0$ upon recovery.

2.3 Px86_{man} vs. Px86_{sim}: Read and $\textbf{flush}_{\text{opt}}$/flush Ordering Constraints

The ordering constraints discussed above are described as follows in [Intel 2019], where *italicised text* in square-brackets denotes our added clarification:

(FL) 'Executions of the **flush** instruction are ordered with respect to each other and with respect to writes, locked read-modify-write *[RMW]* instructions, fence instructions *[***mfence** *and* **sfence***]*, and executions of $\textbf{flush}_{\text{opt}}$ to the same cache line. They are not ordered with respect to executions of $\textbf{flush}_{\text{opt}}$ to different cache lines.' [Intel 2019, p. 3-142]

(FO) 'Executions of the $\textbf{flush}_{\text{opt}}$ instruction are ordered with respect to fence instructions *[***mfence** *and* **sfence***]* and to locked read-modify-write *[RMW]* instructions; they are also ordered with respect to the following accesses to the cache line being invalidated: older *[earlier in program order]* writes and older *[earlier in program order]* executions of **flush**. They are not ordered with respect to writes, executions of **flush** that access other cache lines, or executions of $\textbf{flush}_{\text{opt}}$ regardless of cache line; to enforce $\textbf{flush}_{\text{opt}}$ ordering with any write, **flush**, or $\textbf{flush}_{\text{opt}}$ operation, software can insert an **sfence** instruction between $\textbf{flush}_{\text{opt}}$ and that operation.' [Intel 2019, p. 3-144]

(SF) 'The **sfence** instruction is ordered with respect to memory stores *[writes]*, other **sfence** instructions, **mfence** instructions, and any serializing instructions... It is not ordered with respect to memory loads *[reads]*...' [Intel 2019, p. 4-597]

Observe that (FL)-(FO) do not constrain the order between $\textbf{flush}_{\text{opt}}$/**flush** and read instructions: persist instructions may also be reordered with respect to read instructions. However, our extensive discussions with engineers at Intel revealed that the intended behaviour is more constrained:

(SIM) Executions of $\textbf{flush}_{\text{opt}}$/**flush** are ordered with respect to older *[earlier in program order]* reads regardless of their cache line. Executions of $\textbf{flush}_{\text{opt}}$/**flush** are not ordered with respect to younger *[later in program order]* reads regardless of their cache line.

At the time of publishing this article, the additional constraints in (SIM) are not reflected in the recently revised manual text [Intel 2019]. As such, in order to remain faithful to the manual text

Later in Program Order

Earlier in Program Order		1 Read	2 Write	3 RMW	4 **mfence**	5 **sfence**	6 **flush**$_{opt}$	7 **flush**
A	Read	✓	✓	✓	✓	$Px86_{man}$ ✗ $Px86_{sim}$ ✓*	$Px86_{man}$ ✗ $Px86_{sim}$ ✓	$Px86_{man}$ ✗ $Px86_{sim}$ ✓
B	Write	✗	✓	✓	✓	✓	CL	✓
C	RMW	✓	✓	✓	✓	✓	✓	✓
D	**mfence**	✓	✓	✓	✓	✓	✓	✓
E	**sfence**	✗	✓	✓	✓	✓	✓	✓
F	**flush**$_{opt}$	✗	✗	✓	✓	✓	✗	CL
G	**flush**	✗	✓	✓	✓	✓	CL	✓

Fig. 3. A summary of ordering constraints in $Px86_{sim}$ and $Px86_{man}$, where ✓denotes that two instructions are ordered, ✗ denotes that they are not ordered (and thus may be reordered), and CL denotes that they are ordered if and only if they are on the same cache line; see page 9 for a clarification of ✓*. $Px86_{sim}$ and $Px86_{man}$ agree on all constraints but those between earlier reads and later **flush**$_{opt}$/**flush**/**sfence**. The highlighted cells denote the $Px86_{sim}$/$Px86_{man}$ extensions from the original x86-TSO model by Sewell et al. [2010].

and to account for the intended behaviour enforced by (SIM), we develop two formal persistency models for Intel-x86: (1) the weaker $Px86_{man}$ model which is faithful to the manual [Intel 2019] and excludes the constraints in (SIM); and (2) the stronger $Px86_{sim}$ model which is a simplification of $Px86_{man}$ obtained from extending $Px86_{man}$ with (SIM). We formulate both specifications as an extension of the original x86-TSO model by Sewell et al. [2010].

The examples in Fig. 2 illustrate the difference between $Px86_{man}$ and $Px86_{sim}$. Under $Px86_{man}$, the $\mathtt{a} := y$ read in Fig. 2a may be reordered after **flush** x and thus when **flush** x is executed, it may not necessarily persist the $x := 1$ write in the left thread. As such, upon recovery it is possible to observe x=0, z=1, even though x=0, z=1 is not possible during normal (non-crashing) executions.

Note that inserting an **sfence** after $\mathtt{a} := y$ in Fig. 2b does not alter this behaviour since read instructions may be reordered with respect to **sfence** (see (SF) above). Consequently, as in Fig. 2a the $\mathtt{a} := y$ can be reordered after **sfence**; **flush** x, and once again it is possible to observe z=1, x=0.

However, inserting an **mfence** after $\mathtt{a} := y$ in Fig. 2c prohibits this behaviour: **mfence** ensures that $\mathtt{a} := y$ cannot be reordered after **flush**. As such, if upon recovery z=1 (i.e. $z := 1$ has executed and persisted), then x=1 (i.e. **flush** x must have executed, thus persisting $x := 1$).

Observe that by contrast, due to the extra constraint in (SIM), under $Px86_{sim}$ the $\mathtt{a} := y$ reads in Fig. 2a, Fig. 2b and Fig. 2c are ordered with respect to the later (in program order) **flush** instructions and thus cannot be reordered after it. As such, in all three examples, regardless of the presence of additional fences z=1 $\Rightarrow$ x=1 holds, and thus z=1, x=0 is not possible after recovery.

Ordering Constraints in $Px86_{man}$/$Px86_{sim}$. Fig. 3 presents a summary of ordering constraints between earlier (in program order) instructions (rows) and later instructions (columns) under $Px86_{sim}$ and $Px86_{man}$. Intuitively, if two instructions are ordered (denoted by ✓), then 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 CL entries denote that two instructions are ordered if and only if they access the same cache line.

The top left corner of the table (not highlighted) corresponds to the original x86-TSO model by Sewell et al. [2010]. We develop $Px86_{sim}$ and $Px86_{man}$ by extending x86-TSO with **sfence**, **flush**$_{opt}$

and **flush** instructions. The ordering constraints on the $Px86_{sim}$/$Px86_{man}$ extensions are denoted by the highlighted cells. As shown, $Px86_{sim}$ and $Px86_{man}$ agree on all constraints except for those between earlier reads and later $\mathbf{flush}_{opt}$/**flush** instructions (cells A6-A7) as mandated by (SIM), as well as those between earlier reads and later **sfence** instructions (cell A5) as discussed below.

***Px86 and* sfence *Orderings* (✓*).** Recall from (SF) that **sfence** instructions are not ordered with respect to reads and may be reordered. Under $Px86_{sim}$, however, reordering a *later* **sfence** before an *earlier read* does not affect the program behaviour. In particular, as earlier read instructions are ordered with respect to all other later instructions under $Px86_{sim}$ (see row A in Fig. 3), reordering them after a later **sfence** instruction does not alter the program behaviour. That is, given a program $\mathsf{P} \triangleq \mathtt{a} := x; \mathbf{sfence}; \mathsf{C}$ with $\mathsf{C} \neq \mathbf{sfence}$, the $\mathtt{a} := x$ read cannot be reordered after C, and reordering it after **sfence** alone does not affect the behaviour of P. An example of this is illustrated in Fig. 2b: although $\mathtt{a} := y$ may be reordered after **sfence**, as it cannot be reordered after **flush** x, the program behaviour is unaltered regardless of whether $\mathtt{a} := y$ and **sfence** are reordered. As such, for simplicity we opt to order earlier reads and later **sfence** under $Px86_{sim}$, as denoted by ✓* (cell A5).

Note that by contrast, under the $Px86_{man}$ model earlier reads are not ordered with respect to other later instructions (e.g. later $\mathbf{flush}_{opt}$/**flush** – see cells A6 and A7), and thus reordering an earlier read after a later **sfence** may alter the program behaviour. For instance, as shown in Fig. 2b, the $\mathtt{a} := y$ read is not ordered with respect to the later **sfence** and **flush** x, and thus may be reordered after both, allowing us to observe $z{=}1 \wedge x{=}0$ upon recovery under $Px86_{man}$. However, had we not allowed the reordering of $\mathtt{a} := y$ after **sfence**, it could subsequently not be ordered after **flush** x, and thus as in the case of $Px86_{sim}$, we could no longer observe $z{=}1 \wedge x{=}0$ upon recovery.

Similarly, as later reads are not always ordered with respect to earlier instructions (see column 1), reordering them before earlier **sfence** may alter the behaviour; they are thus unordered (cell E1).

2.4 The Operational $Px86_{sim}$ and $Px86_{man}$ Models

We briefly describe the operational x86-TSO model and how we extend it to model $Px86_{sim}$/$Px86_{man}$.

The x86-TSO Model with Store Buffers. The Intel-x86 architecture follows the *total store ordering* (x86-TSO) model [Sewell et al. 2010] first introduced by the SPARC architecture [SPARC 1992]. Under this model, each thread is connected to the main (volatile) memory via a FIFO *store buffer*, as illustrated in Fig. 4a. The execution of writes is *delayed*: when a thread issues a write to a location, the write is recorded only in its buffer. The delayed writes in the buffer are *debuffered* and propagated (in FIFO order) to the memory at non-deterministic points in time. 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, the thread reads the value of the *last* buffered write to x; otherwise, it consults the memory. In other words, one can model the reordering of a write w after a later read r (cell B1 of Fig. 3) by delaying the debuffering of w until after r has executed in real time. Programmers can use **mfence** instructions to control this debuffering: executing an **mfence** flushes the store buffer of the executing thread to the memory, debuffering all its delayed writes.

Alternative x86-TSO Model with Load Buffers. Abdulla et al. [2015] propose an alternative operational x86-TSO model where each thread is connected to the memory via a FIFO *load buffer* rather than a store buffer. Under this model the execution of reads can be *promoted*: at any point a thread may *pre-fetch* the value of a location from memory and append it to its load buffer. That is, each buffer entry is of the form $\langle x, v \rangle$, denoting that value v was read from x in memory. By contrast, the writes are executed in *real time*: when a thread issues a write $x := v$, value v is written directly to memory at x, updating all x entries of the form $\langle x, - \rangle$ in its load buffer to $\langle x, v \rangle$. When a thread issues a read $\mathtt{a} := x$, it may proceed only if the first entry in its buffer is of the form $\langle x, v \rangle$, in

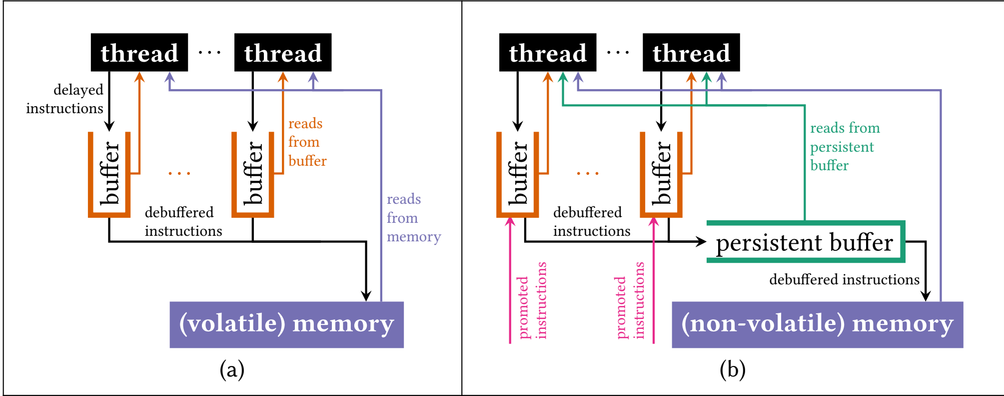


Fig. 4. The storage subsystems of the x86-TSO (left) and Px86 (right) memory models

which case it returns v. To make sure that the execution of reads is not stuck, a thread may always discard the promoted entries from its buffer. Before executing $\mathtt{a} := x$, a thread may thus inspect the entries in its load buffer in turn until it arrives at the first entry for x, discarding the non-x entries along the way. Intuitively, one can model the reordering of a write w after a later read r by promoting (pre-fetching) r before w is executed in real time. As before, **mfence** can be used to prevent such instruction reorderings: executing **mfence** drains the load buffer of the thread.

$Px86_{man}$: Modelling the Ordering Constraints. Note that in the x86-TSO models discussed above there is only one type of entries in the buffer: either delayed entries in the store buffer, or promoted entries in the load buffer, but not both. However, to model the ordering constraints in $Px86_{man}$, we need *both* delayed and promoted entries. To see this, let us attempt to extend the store buffering model with delayed entries where reads are executed in real time. As shown in Fig. 3, the order between read and **sfence**/**flush**/$\mathbf{flush}_{opt}$ instructions is not preserved under $Px86_{man}$ and they may be reordered in either direction. As in the store buffering model, we can model the reordering of **sfence**/**flush**/$\mathbf{flush}_{opt}$ after later reads by delaying their executions in the buffer. However, as reads are executed in real time, we cannot model the reordering of **sfence**/**flush**/$\mathbf{flush}_{opt}$ *before* earlier reads without *promoting* their executions. Similarly, let us attempt to extend the load buffering model with promoted entries where writes are executed in real time. Note that the order between write and $\mathbf{flush}_{opt}$ instructions on different cache lines is not preserved and they may be reordered in either direction. We can then model the ordering of $\mathbf{flush}_{opt}$ before earlier writes by promoting the execution of $\mathbf{flush}_{opt}$. However, since writes are executed in real time, we cannot model the reordering of $\mathbf{flush}_{opt}$ *after* later writes without *delaying* the $\mathbf{flush}_{opt}$ execution.

By allowing both types of entries in the buffer we can extend either model to $Px86_{man}$. Here we extend the store buffering model of Sewell et al. [2010] as it is more widely known. More concretely, (1) the execution of writes is delayed in the buffer while reads are executed in real time, as before; and (2) the execution of **sfence**/**flush**/$\mathbf{flush}_{opt}$ may be delayed or promoted in the buffer.

$Px86_{sim}$: Modelling the Ordering Constraints. As shown in Fig. 3, unlike in $Px86_{man}$, the order between earlier reads and later **sfence**/$\mathbf{flush}_{opt}$/**flush** is preserved in $Px86_{sim}$ and they cannot be reordered. As such, in the $Px86_{sim}$ model the execution of **sfence**/**flush**/$\mathbf{flush}_{opt}$ is no longer required to be promoted. More concretely, (1) the execution of writes is delayed in the buffer while reads are executed in real time, as in $Px86_{man}$; (2) the execution of **sfence**/**flush**/$\mathbf{flush}_{opt}$ may be delayed (but not promoted) in the buffer. The reordering of **sfence**/$\mathbf{flush}_{opt}$/**flush** after later reads can be modelled by delaying **sfence**/$\mathbf{flush}_{opt}$/**flush**. The remaining bidirectional reorderings

Basic domains		**Expressions and sequential commands**
$\mathsf{a} \in \textsc{Reg}$	Registers	$\textsc{Exp} \ni \mathsf{e} ::= v \mid \mathsf{a} \mid \mathsf{e{+}e} \mid \cdots$
$v \in \textsc{Val}$	Values	$\textsc{PCom} \ni \mathsf{c} ::= \mathbf{load}(x) \mid \mathbf{store}(x, \mathsf{e}) \mid \mathbf{CAS}(x, \mathsf{e}, \mathsf{e}') \mid \mathbf{FAA}(x, \mathsf{e})$
$\tau \in \textsc{TId}$	Thread IDs	$\mid \mathbf{mfence} \mid \mathbf{sfence} \mid \mathbf{flush}_{\text{opt}}\ x \mid \mathbf{flush}\ x$
Programs		$\textsc{Com} \ni \mathsf{C} ::= \mathsf{e} \mid \mathsf{c} \mid \mathbf{let}\ \mathsf{a}{:=}\mathsf{C}\ \mathbf{in}\ \mathsf{C}$
$\mathsf{P} \in \textsc{Prog} \triangleq \textsc{TId} \xrightarrow{\text{fin}} \textsc{Com}$		$\mid \mathbf{if}\ (\mathsf{C})\ \mathbf{then}\ \mathsf{C}\ \mathbf{else}\ \mathsf{C} \mid \mathbf{repeat}\ \mathsf{C}$

Fig. 5. A simple concurrent programming language

amongst write, $\mathbf{flush}_{\text{opt}}$ and **flush** instructions can be modelled by debuffering their associated delayed entries from the store buffer in the desired order.

Px86: Modelling Buffered and Relaxed Persistency. In order to allow for buffered persists, the Px86 storage system has an additional layer compared to its x86-TSO counterpart: a *persistent buffer*, as illustrated in Fig. 4b. The persistent buffer contains those writes that are pending to be persisted to the (non-volatile) memory. As with the memory, the persistent buffer is accessible by all threads. However, while the memory is non-volatile, the persistent buffer is volatile and its contents are lost upon a crash. When delayed writes in the thread-local buffer are debuffered, they are propagated to the persistent buffer; this debuffering denotes the *store* associated with the write, i.e. when the write is made visible to other threads. Pending writes in the persistent buffer are in turn debuffered and propagated to the memory at non-deterministic points in time; this debuffering denotes the *persist* associated with the write, i.e. when the write 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 persistent buffer for the value of the last buffered store to x if such a store exists; otherwise, it reads x from the memory.

Lastly, recall that under Px86 the writes on distinct locations may persist in any order (see Fig. 1a), while the writes on the same location persist in the store order (see Fig. 1f). We thus model the persistent buffer as a queue, where the pending writes on each location x are propagated in the FIFO queue order, while those on different locations are propagated in an arbitrary order.

3 THE OPERATIONAL Px86$_{\text{sim}}$ MODEL

Locations and Cache Lines. We assume a set of *locations*, $\textsc{Loc}$, and a set of cache lines $\text{CL} \triangleq \mathcal{P}(\textsc{Loc})$. We typically use $x, y, \ldots$ as meta-variables for locations, and $X, Y, \ldots$ for cache lines.

Programming Language. To keep our presentation concise, we employ a simple concurrent programming language as given in Fig. 5. We assume a finite set $\textsc{Reg}$ of registers (local variables); a finite set $\textsc{Val}$ of values; a finite set $\textsc{TId} \subseteq \mathbb{N}^+$ of thread identifiers; and any standard interpreted language for expressions, $\textsc{Exp}$, containing registers and values. We use v as a metavariable for values, τ for thread identifiers, and e for expressions. We model a multi-threaded program P as a function mapping each thread to its (sequential) program. We write $\mathsf{P}{=}\mathsf{C}_1 || \cdots || \mathsf{C}_n$ when $dom(\mathsf{P}){=}\{\tau_1 \cdots \tau_n\}$ and $\mathsf{P}(\tau_i){=}\mathsf{C}_i$. Sequential programs are described by the $\textsc{Com}$ grammar and include *primitives* (c), as well as the standard constructs of expressions, assignments, conditionals and loops.

The highlighted primitives denote the extensions of the original x86-TSO model by Sewell et al. [2010]. The $\mathbf{load}(x)$ denotes an atomic *read* from location x; similarly, the $\mathbf{store}(x, \mathsf{e})$ denotes an atomic *write* to location x. The $\mathbf{CAS}(x, \mathsf{e}, \mathsf{e}')$ 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

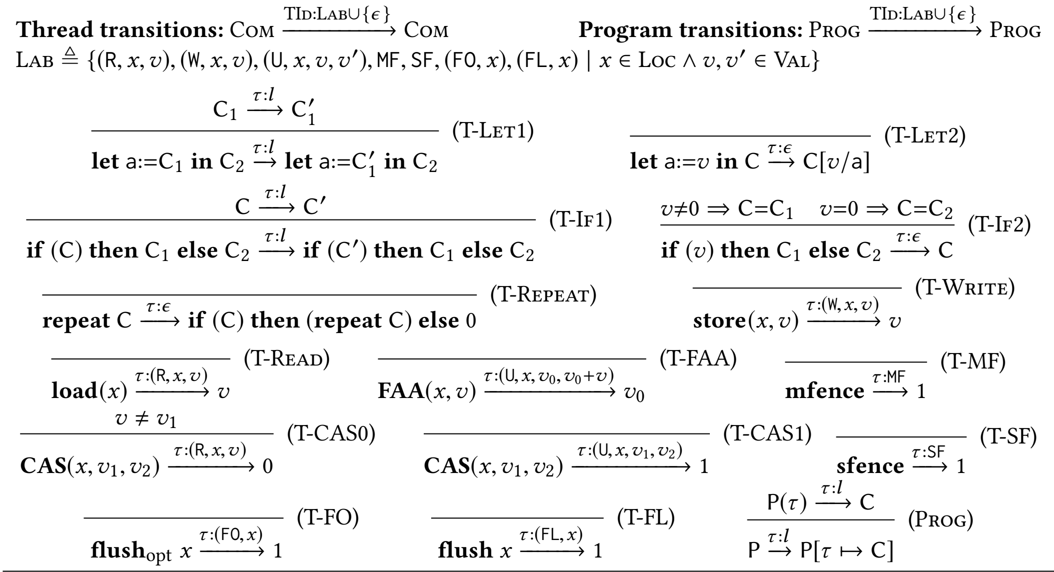


Fig. 6. Program transitions in Px86

is returned; otherwise x is left 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, $\textbf{flush}_{\text{opt}}$ and **flush** denote persist instructions, persisting a given cache line as discussed in §2.

For better readability, we do not always follow syntactic conventions and write e.g. $\mathsf{a} := \mathsf{C}$ for **let** $\mathsf{a}{:=}\mathsf{C}$ **in** a, and $\mathsf{C}_1; \mathsf{C}_2$ for **let** $\mathsf{a}{:=}\mathsf{C}_1$ **in** C_2, where a is a fresh local variable. We write $a := x$ for **let** a:=**load**(x) **in** a (where a is fresh); similarly for **CAS** and **FAA**. We write $x := \mathsf{e}$ for **store**(x, e).

3.1 The Operational $\mathbf{Px86_{sim}}$ Model

We describe the Px86_{sim} 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 (the non-volatile memory, persistent buffer and thread-local buffers in Fig. 4b) determine the execution steps. The Px86_{sim} operational semantics is then defined by combining the transitions of its program and storage subsystems.

3.1.1 Program Transitions. The Px86_{sim} program transitions are given in Fig. 6. Program transitions are defined via the transitions of their constituent threads. Thread transitions are of the form: $\mathsf{C} \xrightarrow{\tau:l} \mathsf{C}'$, where $\mathsf{C}, \mathsf{C}' \in \textsc{Com}$ denote sequential commands (Fig. 5). The $\tau{:}l$ marks the transition by recording the identifier of the executing thread τ, as well as the transition *label* l. A label may be ϵ, to denote silent transitions of no-ops; $(\mathtt{R}, x, v)$ to denote reading v from location x; $(\mathtt{W}, x, v)$ to denote writing v to location x; $(\mathtt{U}, x, v, v')$ to denote a successful update (RMW) instruction modifying x to v' when its value matches v; MF or SF to denote the execution of **mfence** or **sfence**, respectively; and $(\mathtt{FO}, x)$ or $(\mathtt{FL}, x)$ to denote the execution of $\textbf{flush}_{\text{opt}}$ x or **flush** x, respectively.

Most thread transitions are standard. The (T-CAS0) transition describes the reduction of the **CAS**(x, v_1, v_2) instruction when unsuccessful; i.e. when the value read (v) is different from v_1. The (T-CAS1) transition dually describes the reduction of a **CAS** when successful. Note that in the failure

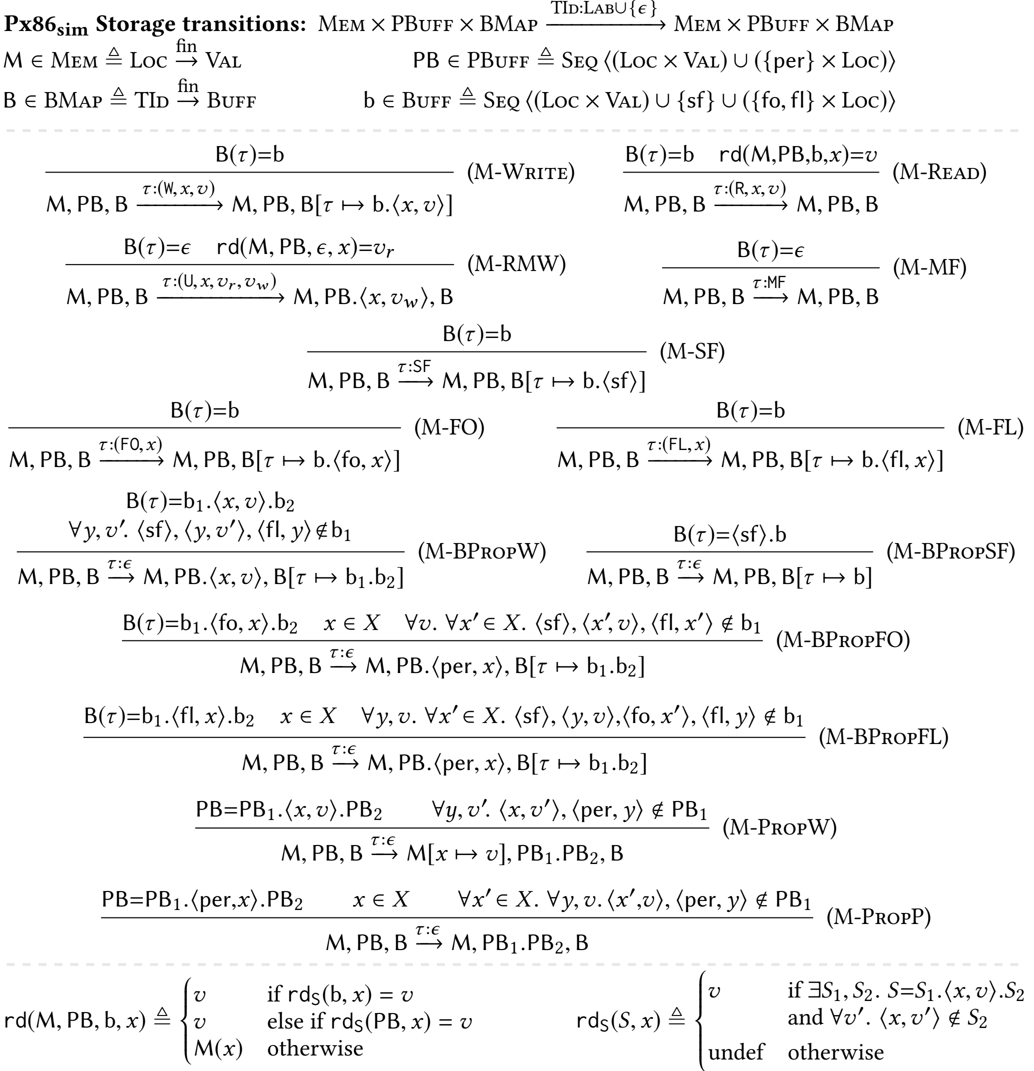


Fig. 7. Storage transitions in Px86_{sim}

case, as no update takes place, the transition is marked with a read label (R, x, v) and not an update label as in the success case. The (T-FAA) transition behaves analogously. The (T-MF) and (T-SF) transitions respectively describe executing an **mfence** and **sfence**, reducing to value 1 (successful termination). Analogously, (T-FO) and (T-FL) describe the execution of persist instructions.

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

3.1.2 *Storage Transitions.* The Px86_{sim} storage transitions in Fig. 7 are of the form: $\mathsf{M}, \mathsf{PB}, \mathsf{B} \xrightarrow{\tau:l} \mathsf{M}', \mathsf{PB}', \mathsf{B}'$, where $\mathsf{M}, \mathsf{M}' \in \textsc{Mem}$ denote the *memory*, modelled as a (finite) map from locations to values; $\mathsf{PB}, \mathsf{PB}' \in \textsc{PBuff}$ denote the *persistent buffer*, represented as a sequence of persist-pending instructions; and $\mathsf{B}, \mathsf{B}' \in \textsc{BMap}$ denote the *buffer map*, associating each thread with its *buffer*.

Recall from §2.4 that under the Px86_{sim} model the buffer contains delayed entries only. More concretely, each entry in the buffer may be of the form: (1) $\langle x, v \rangle$, denoting a delayed write $x := v$; (2) $\langle \mathsf{sf} \rangle$, $\langle \mathsf{fo}, x \rangle$ or $\langle \mathsf{fl}, x \rangle$, denoting a delayed **sfence**, $\textbf{flush}_{\text{opt}}\ x$ or **flush** x, respectively.

Px86_{sim}: Thread Buffers and Ordering Constraints. A delayed entry (e.g. $\langle x, v \rangle$) is added to the buffer when a corresponding instruction (e.g. $x := v$) is encountered in the program, and is debuffered at arbitrary times (subject to ordering constraints), as we describe shortly. Intuitively, the order in which delayed entries are added to the buffer corresponds to the program order, while the order in which they are debuffered denotes their store order (i.e. the order in which they are made visible to other threads). That is, a delayed entry reaches its *program* (resp. *store*) *point* when it is added to (resp. removed from) the buffer. We use this intuition to encode the Px86_{sim} ordering constraints in Fig. 3. In particular, we check the constraints between delayed entries when debuffering them by comparing their program order (the FIFO buffer order) and store order (the debuffering order). Note that despite allowing only delayed entries in the thread buffers, we can still model the reordering of later instructions before earlier instructions allowed under Px86_{sim}. For instance, we can model the reordering of $\textbf{flush}_{\text{opt}}$ before earlier writes on different cache lines by debuffering their associated delayed entries in the reverse order.

The Persistent Buffer. As discussed in §2.4, the persistent buffer entries are propagated to the memory at arbitrary times. Intuitively, the effect of an entry is committed durably to memory (i.e. its persist takes place) once it leaves the persistent buffer. Therefore, since **sfence** instructions have no durable effect and merely constrain the execution order, they are not retained in the persistent buffer: when an $\langle \mathsf{sf} \rangle$ entry is eventually debuffered from the thread buffer, it is simply discarded. Each entry in the persistent buffer may be thus of the form: (1) $\langle x, v \rangle$, denoting a persist-pending write; or (2) $\langle \mathsf{per}, x \rangle$, denoting a persist-pending $\textbf{flush}_{\text{opt}}\ x$/**flush** x. This is because the fo, fl tags determine the constraints on the execution order and do not affect the durable effect of $\textbf{flush}_{\text{opt}}$/**flush**. That is, $\textbf{flush}_{\text{opt}}\ x$ and **flush** x have the same durable effect: when made durable, they both persist the cache line containing x. As such, distinguishing the persist type in the persistent buffer is unnecessary and we simply represent them by the 'per' tag.

Px86_{sim} Storage Transitions. When a thread writes v to x, this is recorded in its buffer as the $\langle x, v \rangle$ entry, as described by (M-Write). Recall that when a thread reads from x, it first consults its own buffer, followed by the persistent buffer (if no write to x is found in its buffer), and finally the non-volatile memory (if no store to x is found in the thread or persistent buffer). This lookup chain is captured by $\mathsf{rd}(\mathsf{M}, \mathsf{PB}, \mathsf{b}, x)$ in the premise of (M-Read), defined at the bottom of Fig. 7.

The (M-MF) rule ensures that an **mfence** proceeds only when the buffer of the executing thread is drained, as stipulated by the $\mathsf{B}(\tau)=\epsilon$ premise. In the (M-RMW) rule, when executing an RMW instruction (i.e. a **CAS** or **FAA**) on x, a similar lookup chain is followed to determine the value of x, as with a read. To ensure their atomicity, RMW instructions act as memory fences and may only proceed when the buffer of the executing thread is drained. Moreover, the resulting update is committed directly to the persistent buffer, bypassing the thread buffer. This is to ensure that the resulting update is immediately visible to other threads. Note that the behaviour of RMW instructions with respect to thread buffers differs from that of write instructions: writes are added to the thread buffer; updates bypass the thread buffer and flush it of pending entries.

The (M-SF) rule describes the (delayed) **sfence** execution by adding $\langle \mathsf{sf} \rangle$ to b. Similarly, (M-FO) and (M-FL) describe the (delayed) execution of $\textbf{flush}_{\text{opt}}$ and **flush**, respectively.

The (M-BPropW) describes the debuffering of a delayed write and propagating it to the persistent buffer, i.e. when its store finally takes place. Let $d = \langle x, v \rangle$ denote the write debuffered from b in (M-BPropW). As discussed above, when debuffering d we must check the constraints between d

Operational semantics: $\textsc{Prog} \times \textbf{Rec} \vdash \textsc{Conf} \Rightarrow \textsc{Conf}$ $\qquad \textsc{Conf} \triangleq \textsc{Prog} \times \textsc{Mem} \times \textsc{PBuff} \times \textsc{BMap}$

$\textbf{rec} \in \textbf{Rec} \triangleq \textsc{Prog} \times \textsc{Mem} \rightarrow \textsc{Prog}$

$$\frac{\mathsf{P} \xrightarrow{\tau:\epsilon} \mathsf{P}'}{\Delta \vdash \mathsf{P}, \mathsf{M}, \mathsf{PB}, \mathsf{B} \Rightarrow \mathsf{P}', \mathsf{M}, \mathsf{PB}, \mathsf{B}} \ (\textsc{SilentP}) \qquad \frac{\mathsf{P} \xrightarrow{\tau:l} \mathsf{P}' \quad \mathsf{M}, \mathsf{PB}, \mathsf{B} \xrightarrow{\tau:l} \mathsf{M}', \mathsf{PB}', \mathsf{B}'}{\Delta \vdash \mathsf{P}, \mathsf{M}, \mathsf{PB}, \mathsf{B} \Rightarrow \mathsf{P}', \mathsf{M}', \mathsf{PB}', \mathsf{B}'} \ (\textsc{Step})$$

$$\frac{\mathsf{M}, \mathsf{PB}, \mathsf{B} \xrightarrow{\tau:\epsilon} \mathsf{M}', \mathsf{PB}', \mathsf{B}'}{\Delta \vdash \mathsf{P}, \mathsf{M}, \mathsf{PB}, \mathsf{B} \Rightarrow \mathsf{P}, \mathsf{M}', \mathsf{PB}', \mathsf{B}'} \ (\textsc{SilentS}) \qquad \frac{\Delta{=}(\mathsf{P}_0, \textbf{rec}) \quad \mathsf{B}_0 \triangleq \lambda\tau.\epsilon \quad \mathsf{PB}_0 \triangleq \epsilon}{\Delta \vdash \mathsf{P}, \mathsf{M}, \mathsf{PB}, \mathsf{B} \Rightarrow \textbf{rec}(\mathsf{P}_0, \mathsf{M}), \mathsf{M}, \mathsf{PB}_0, \mathsf{B}_0} \ (\textsc{Crash})$$

Fig. 8. The Px86 operational semantics

and other delayed entries in b. This is captured by the second premise of the rule. For instance, let $d'{=}\langle y, v'\rangle$ denote a delayed write preceding d in b; i.e. d' is program-ordered before d. Note that since two writes cannot be reordered, the store and program order between d and d' must agree: d' must be debuffered before d. Hence when d is debuffered, no delayed write may preceded it in b: $\forall y, v'.\ \langle y, v'\rangle \notin \mathsf{b}_1$. Analogously, (M-BPropSF), (M-BPropFO) and (M-BPropFL) describe the debuffering of delayed **sfence**, $\textbf{flush}_{\text{opt}}$ and **flush** instructions, respectively.

The (M-PropW) rule describes the debuffering of a write from the persistent buffer PB, i.e. when its persist finally takes place. Recall that the writes on the *same* location are persisted in the store order. That is, the writes on each location in PB are debuffered (persisted) in the order in which they were added to PB. This is ensured by requiring that the debuffered $\langle x, v\rangle$ entry be the first x entry in PB. Moreover, as discussed in §2.2, given $x \in X$ and a persist instruction C on x, all writes on X (store-) ordered before C persist before all instructions (store-) ordered after C. This is captured by ensuring that no persist entry precedes the debuffered $\langle x, v\rangle$. Similarly, (M-PropP) describes the debuffering of a persist entry from the persistent buffer.

3.1.3 Combined Transitions. The Px86_{sim} operational semantics is defined by combining the transitions of the program and storage subsystems under a *recovery context*, as presented in Fig. 8. The recovery context records the information necessary for correct recovery after a crash. A recovery context is a pair $\Delta{=}(\mathsf{P}, \textbf{rec})$, where P denotes the original program, and **rec** denotes its associated *recovery program*. A naive recovery program may always choose to *restart* executing P from the beginning. However, a more sophisticated recovery may *resume* executing P upon recovery by determining the progress made prior to the crash. To do this, the recovery program may inspect the memory to determine the operations whose effects have persisted to memory. To capture this, we model a recovery program as a function $\textbf{rec} : \textsc{Prog} \times \textsc{Mem} \rightarrow \textsc{Prog}$. That is, $\textbf{rec}(\mathsf{P}, \mathsf{M})$ describes the recovery program of P when the memory upon recovery is described by M.

The (SilentP) rule describes the case when the program subsystem takes a silent step and thus the storage subsystem is left unchanged; *mutatis mutandis* for (SilentS). The (Step) rule describes the case when 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. Lastly, the (Crash) rule describes the case when the program crashes: the execution is restarted with the recovery program $\textbf{rec}(\mathsf{P}, \mathsf{M})$; the memory is left unchanged as it is non-volatile; the thread-local buffers and the persistent buffer are lost (as they are volatile) and are thus reset to empty.

4 THE OPERATIONAL Px86_{man} MODEL

As with Px86_{sim}, we describe the Px86_{man} operational model by separating the transitions of its program and storage subsystems. While the Px86_{man} program transitions are those of Px86_{sim} in Fig. 6, the Px86_{man} storage transitions are more complex than those of Px86_{sim}, and are presented

in Fig. 9. As with Px86_{sim}, the Px86_{man} operational semantics is then defined by combining the transitions of its program and storage subsystems, as described in Fig. 8.

The Px86_{man} storage transitions in Fig. 9 are of the form: $\mathsf{M}, \mathsf{PB}, \mathsf{B} \xrightarrow{\tau:l} \mathsf{M}', \mathsf{PB}', \mathsf{B}'$, where $\mathsf{M}, \mathsf{M}', \mathsf{PB}, \mathsf{PB}'$ are defined as before, and highlighted sections denote extensions from Px86_{sim}. Recall that Px86_{man} buffers contains both delayed and promoted entries. As such, the buffer maps B, B' associate each thread with its *$Px86_{man}$ buffer*, extended from Px86_{sim} buffers to include promoted entries. More concretely, each Px86_{man} buffer entry may be of the form: (1) $\langle x, v \rangle$, denoting a delayed write $x := v$; (2) $\langle \mathsf{sf} \rangle$, $\langle \mathsf{fo}, x \rangle$ or $\langle \mathsf{fl}, x \rangle$, denoting a delayed **sfence**, $\textbf{flush}_{\text{opt}}$ x or **flush** x, respectively; or (3) $\langle \mathsf{psf} \rangle$, $\langle \mathsf{pfo}, x \rangle$ or $\langle \mathsf{pfl}, x \rangle$, denoting a promoted **sfence**, $\textbf{flush}_{\text{opt}}$ x or **flush** x, respectively.

$Px86_{man}$: Thread Buffers and Ordering Constraints. As in Px86_{sim}, a delayed entry (e.g. $\langle x, v \rangle$) is added to the buffer when a corresponding instruction (e.g. $x := v$) is encountered in the program, and is debuffered at arbitrary times (subject to ordering constraints). Conversely, a promoted entry may be added to the buffer at arbitrary times (subject to constraints), and is debuffered when a corresponding instruction is encountered. Additionally, a promoted entry may always be discarded to ensure progress. Recall that the order in which delayed entries are added to the buffer corresponds to the program order, while the order in which they are debuffered denotes their store order. Dually, the order in which promoted entries are added denotes their store order, while the order in which they are debuffered (not discarded) corresponds to the program order. That is, a promoted entry reaches its *program* (resp. *store*) *point* when it is removed from (resp. added to) the buffer. As such, given promoted entries p, p', a delayed entry d and a buffer $\mathsf{b}{=}p.d.p'$, then: (1) p' is store-ordered after p (p' is added after p); d is program-ordered before p, p' (d has reached its program point while p, p' are yet to reach theirs); and (2) d is store-ordered after p, p' (d is yet to reach its store point while p, p' have reached theirs). As the program and store orders between d and p, p' disagree, p, p' are *reordered* before d, or (equivalently) d is reordered after p, p'.

We use these intuitions to encode the Px86_{man} ordering constraints in Fig. 3. In particular, the constraints between delayed entries are checked when debuffering them by comparing their program order (the FIFO buffer order) and store order (the debuffering order). The constraints between promoted entries are checked when debuffering them by comparing their program order (the debuffering order) and store order (the FIFO buffer order). The constraints between delayed and promoted entries are checked when adding either. For instance, in the example above, when d is added yielding $p.d$, we compare the program order (d before p) against the store order (p before d) to ensure that ordering constraints are respected. Similarly, when p' is later added yielding $p.d.p'$, we compare the program order (d before p') against the store order (p' before d).

Debuffering promoted entries *justifies* their execution: it ensures that their execution speculated beforehand corresponds to a later instruction in program. This prohibits the promoted execution of non-existent instructions as unjustifiable promoted entries must be discarded to ensure progress.

$Px86_{man}$ Storage Transitions. Given a thread τ and its buffer b, the (M-ProFO) rule describes the promoted execution of $\textbf{flush}_{\text{opt}}$ x by τ, where $\langle \mathsf{pfo}, x \rangle$ is added to b. As discussed above, we must check the constraints between the promoted entry $\langle \mathsf{pfo}, x \rangle$ and existing delayed entries in b. Recall from Fig. 3 that $\textbf{flush}_{\text{opt}}$ instructions cannot be reordered with respect to **sfence**. As such, when adding $\langle \mathsf{pfo}, x \rangle$, the buffer cannot contain a delayed **sfence** entry. This is captured by requiring $\langle \mathsf{sf} \rangle \notin \mathsf{b}$ in the last premise. Note that otherwise the $\textbf{flush}_{\text{opt}}$ associated with $\langle \mathsf{pfo}, x \rangle$ is reordered before **sfence** which is prohibited. The last premise additionally checks the ordering constraints between $\langle \mathsf{pfo}, x \rangle$ and delayed write/**flush** entries in b. Lastly, recall that the persist of each instruction occurs after its associated store. As the store of a promoted persist entry is reached

Px86$_{\text{man}}$ Storage transitions: $\textsc{Mem} \times \textsc{PBuff} \times \textsc{BMap} \xrightarrow{\textsc{TId}:\textsc{Lab} \cup \{\epsilon\}} \textsc{Mem} \times \textsc{PBuff} \times \textsc{BMap}$

$\mathsf{B} \in \textsc{BMap} \triangleq \textsc{TId} \xrightarrow{\text{fin}} \textsc{Buff}$ $\quad$ $\mathsf{b} \in \textsc{Buff} \triangleq \textsc{Seq}\langle (\textsc{Loc} \times \textsc{Val}) \cup \{\mathsf{sf}, \mathsf{psf}\} \cup (\{\mathsf{fo}, \mathsf{fl}, \mathsf{pfo}, \mathsf{psf}\} \times \textsc{Loc})\rangle$

$$\frac{\mathsf{B}(\tau)=\mathsf{b} \qquad x \in X \qquad \forall v.\, \forall x' \in X.\, \langle \mathsf{sf} \rangle, \langle x', v \rangle, \langle \mathsf{fl}, x' \rangle \notin \mathsf{b}}{\mathsf{M}, \mathsf{PB}, \mathsf{B} \xrightarrow{\tau:\epsilon} \mathsf{M}, \mathsf{PB}.\langle \mathsf{per}, x \rangle, \mathsf{B}[\tau \mapsto \mathsf{b}.\langle \mathsf{pfo}, x \rangle]}\ (\textsc{M-ProFO})$$

$$\frac{\mathsf{B}(\tau)=\mathsf{b} \qquad x \in X \qquad \forall y, v.\, \forall x' \in X.\, \langle \mathsf{sf} \rangle, \langle y, v \rangle, \langle \mathsf{fo}, x' \rangle, \langle \mathsf{fl}, y \rangle \notin \mathsf{b}}{\mathsf{M}, \mathsf{PB}, \mathsf{B} \xrightarrow{\tau:\epsilon} \mathsf{M}, \mathsf{PB}.\langle \mathsf{per}, x \rangle, \mathsf{B}[\tau \mapsto \mathsf{b}.\langle \mathsf{pfl}, x \rangle]}\ (\textsc{M-ProFL})$$

$$\frac{\forall x, v.\, \langle \mathsf{sf} \rangle, \langle x, v \rangle, \langle \mathsf{fo}, x \rangle, \langle \mathsf{fl}, x \rangle \notin \mathsf{B}(\tau)}{\mathsf{M}, \mathsf{PB}, \mathsf{B} \xrightarrow{\tau:\epsilon} \mathsf{M}, \mathsf{PB}, \mathsf{B}[\tau \mapsto \mathsf{B}(\tau).\langle \mathsf{psf} \rangle]}\ (\textsc{M-ProSF})$$

$$\frac{\mathsf{B}(\tau)=\mathsf{b}_1.o.\mathsf{b}_2 \quad o \in \{\langle \mathsf{psf} \rangle, \langle \mathsf{pfo}, - \rangle, \langle \mathsf{pfl}, - \rangle\}}{\mathsf{M}, \mathsf{PB}, \mathsf{B} \xrightarrow{\tau:\epsilon} \mathsf{M}, \mathsf{PB}, \mathsf{B}[\tau \mapsto \mathsf{b}_1.\mathsf{b}_2]}\ (\textsc{M-Drop})$$

$$\frac{\mathsf{B}(\tau)=\mathsf{b} \quad x \in X \quad \forall y. \forall x' \in X. \langle \mathsf{psf} \rangle, \langle \mathsf{pfl}, y \rangle, \langle \mathsf{pfo}, x' \rangle \notin \mathsf{b}}{\mathsf{M}, \mathsf{PB}, \mathsf{B} \xrightarrow{\tau:(\mathtt{W}, x, v)} \mathsf{M}, \mathsf{PB}, \mathsf{B}[\tau \mapsto \mathsf{b}.\langle x, v \rangle]}\ (\textsc{M-Write})$$

$$\frac{\mathsf{B}(\tau)=\mathsf{b} \quad \mathsf{rd}(\mathsf{M}, \mathsf{PB}, \mathsf{b}, x)=v}{\mathsf{M}, \mathsf{PB}, \mathsf{B} \xrightarrow{\tau:(\mathtt{R}, x, v)} \mathsf{M}, \mathsf{PB}, \mathsf{B}}\ (\textsc{M-Read})$$

$$\frac{\mathsf{B}(\tau)=\epsilon \quad \mathsf{rd}(\mathsf{M}, \mathsf{PB}, \epsilon, x)=v_r}{\mathsf{M}, \mathsf{PB}, \mathsf{B} \xrightarrow{\tau:(\mathtt{U}, x, v_r, v_w)} \mathsf{M}, \mathsf{PB}.\langle x, v_w \rangle, \mathsf{B}}\ (\textsc{M-RMW})$$

$$\frac{\mathsf{B}(\tau)=\epsilon}{\mathsf{M}, \mathsf{PB}, \mathsf{B} \xrightarrow{\tau:\mathtt{MF}} \mathsf{M}, \mathsf{PB}, \mathsf{B}}\ (\textsc{M-MF})$$

$$\frac{\mathsf{B}(\tau)=\mathsf{b} \qquad \forall x.\ \langle \mathsf{psf} \rangle, \langle \mathsf{pfo}, x \rangle, \langle \mathsf{pfl}, x \rangle \notin \mathsf{b}}{\mathsf{M}, \mathsf{PB}, \mathsf{B} \xrightarrow{\tau:\mathtt{SF}} \mathsf{M}, \mathsf{PB}, \mathsf{B}[\tau \mapsto \mathsf{b}.\langle \mathsf{sf} \rangle]}\ (\textsc{M-SF})$$

$$\frac{\mathsf{B}(\tau)=\langle \mathsf{psf} \rangle.\mathsf{b}'}{\mathsf{M}, \mathsf{PB}, \mathsf{B} \xrightarrow{\tau:\mathtt{SF}} \mathsf{M}, \mathsf{PB}, \mathsf{B}[\tau \mapsto \mathsf{b}']}\ (\textsc{M-SF2})$$

$$\frac{\begin{array}{c}\mathsf{B}(\tau)=\mathsf{b} \qquad x \in X \\ \forall x' \in X. \langle \mathsf{psf} \rangle, \langle \mathsf{pfl}, x' \rangle \notin \mathsf{b}\end{array}}{\mathsf{M}, \mathsf{PB}, \mathsf{B} \xrightarrow{\tau:(\mathtt{FO}, x)} \mathsf{M}, \mathsf{PB}, \mathsf{B}[\tau \mapsto \mathsf{b}.\langle \mathsf{fo}, x \rangle]}\ (\textsc{M-FO})$$

$$\frac{\begin{array}{c}\mathsf{B}(\tau)=\mathsf{b}_1.\langle \mathsf{pfo}, x \rangle.\mathsf{b}_2 \quad x \in X \quad \langle \mathsf{psf} \rangle \notin \mathsf{b}_1 \\ \forall x' \in X.\ \langle \mathsf{pfl}, x' \rangle \notin \mathsf{b}_1\end{array}}{\mathsf{M}, \mathsf{PB}, \mathsf{B} \xrightarrow{\tau:(\mathtt{FO}, x)} \mathsf{M}, \mathsf{PB}, \mathsf{B}[\tau \mapsto \mathsf{b}_1.\mathsf{b}_2]}\ (\textsc{M-FO2})$$

$$\frac{\begin{array}{c}\mathsf{B}(\tau)=\mathsf{b} \qquad x \in X \\ \forall y. \forall x' \in X. \langle \mathsf{psf} \rangle, \langle \mathsf{pfl}, y \rangle, \langle \mathsf{pfo}, x' \rangle \notin \mathsf{b}\end{array}}{\mathsf{M}, \mathsf{PB}, \mathsf{B} \xrightarrow{\tau:(\mathtt{FL}, x)} \mathsf{M}, \mathsf{PB}, \mathsf{B}[\tau \mapsto \mathsf{b}.\langle \mathsf{fl}, x \rangle]}\ (\textsc{M-FL})$$

$$\frac{\begin{array}{c}\mathsf{B}(\tau)=\mathsf{b}_1.\langle \mathsf{pfl}, x \rangle.\mathsf{b}_2 \quad x \in X \quad \langle \mathsf{psf} \rangle \notin \mathsf{b}_1 \\ \forall x' \in X.\ \langle \mathsf{pfo}, x' \rangle \notin \mathsf{b}_1 \quad \forall y.\ \langle \mathsf{pfl}, y \rangle \notin \mathsf{b}_1\end{array}}{\mathsf{M}, \mathsf{PB}, \mathsf{B} \xrightarrow{\tau:(\mathtt{FL}, x)} \mathsf{M}, \mathsf{PB}, \mathsf{B}[\tau \mapsto \mathsf{b}_1.\mathsf{b}_2]}\ (\textsc{M-FL2})$$

$$\frac{\begin{array}{c}\mathsf{B}(\tau)=\mathsf{b}_1.\langle x, v \rangle.\mathsf{b}_2 \\ \forall y, v'.\ \langle \mathsf{sf} \rangle, \langle y, v' \rangle, \langle \mathsf{fl}, y \rangle \notin \mathsf{b}_1\end{array}}{\mathsf{M}, \mathsf{PB}, \mathsf{B} \xrightarrow{\tau:\epsilon} \mathsf{M}, \mathsf{PB}.\langle x, v \rangle, \mathsf{B}[\tau \mapsto \mathsf{b}_1.\mathsf{b}_2]}\ (\textsc{M-BPropW})$$

$$\frac{\mathsf{B}(\tau)=\langle \mathsf{sf} \rangle.\mathsf{b}}{\mathsf{M}, \mathsf{PB}, \mathsf{B} \xrightarrow{\tau:\epsilon} \mathsf{M}, \mathsf{PB}, \mathsf{B}[\tau \mapsto \mathsf{b}]}\ (\textsc{M-BPropSF})$$

$$\frac{\mathsf{B}(\tau)=\mathsf{b}_1.\langle \mathsf{fo}, x \rangle.\mathsf{b}_2 \quad x \in X \quad \forall v.\ \forall x' \in X.\ \langle \mathsf{sf} \rangle, \langle x', v \rangle, \langle \mathsf{fl}, x' \rangle \notin \mathsf{b}_1}{\mathsf{M}, \mathsf{PB}, \mathsf{B} \xrightarrow{\tau:\epsilon} \mathsf{M}, \mathsf{PB}.\langle \mathsf{per}, x \rangle, \mathsf{B}[\tau \mapsto \mathsf{b}_1.\mathsf{b}_2]}\ (\textsc{M-BPropFO})$$

$$\frac{\mathsf{B}(\tau)=\mathsf{b}_1.\langle \mathsf{fl}, x \rangle.\mathsf{b}_2 \quad x \in X \quad \forall y, v.\ \forall x' \in X.\ \langle \mathsf{sf} \rangle, \langle y, v \rangle, \langle \mathsf{fo}, x' \rangle, \langle \mathsf{fl}, y \rangle \notin \mathsf{b}_1}{\mathsf{M}, \mathsf{PB}, \mathsf{B} \xrightarrow{\tau:\epsilon} \mathsf{M}, \mathsf{PB}.\langle \mathsf{per}, x \rangle, \mathsf{B}[\tau \mapsto \mathsf{b}_1.\mathsf{b}_2]}\ (\textsc{M-BPropFL})$$

$$\frac{\mathsf{PB}=\mathsf{PB}_1.\langle x, v \rangle.\mathsf{PB}_2 \qquad \forall y, v'.\ \langle x, v' \rangle, \langle \mathsf{per}, y \rangle \notin \mathsf{PB}_1}{\mathsf{M}, \mathsf{PB}, \mathsf{B} \xrightarrow{\tau:\epsilon} \mathsf{M}[x \mapsto v], \mathsf{PB}_1.\mathsf{PB}_2, \mathsf{B}}\ (\textsc{M-PropW})$$

$$\frac{\mathsf{PB}=\mathsf{PB}_1.\langle \mathsf{per}, x \rangle.\mathsf{PB}_2 \qquad x \in X \qquad \forall x' \in X. \forall y, v.\ \langle x', v \rangle, \langle \mathsf{per}, y \rangle \notin \mathsf{PB}_1}{\mathsf{M}, \mathsf{PB}, \mathsf{B} \xrightarrow{\tau:\epsilon} \mathsf{M}, \mathsf{PB}_1.\mathsf{PB}_2, \mathsf{B}}\ (\textsc{M-PropP})$$

Fig. 9. Storage transitions in Px86$_{\text{man}}$, where the highlighted sections denote extensions from Px86$_{\text{sim}}$

upon adding it to the buffer, we can then add an associated $\langle \mathsf{per}, x \rangle$ entry to the persistent buffer, which when eventually debuffered, carries out the persist of the associated $\mathbf{flush}_{\mathrm{opt}}$.

The (M-ProFL) and (M-ProSF) rules analogously describe the promoted execution of **flush** x and **sfence**, respectively. The (M-Drop) rule describes the discarding of promoted entries at will to ensure that transitions do not get stuck.

As in $\mathrm{Px86}_{\mathrm{sim}}$, when a thread writes v to x, this is recorded in its buffer as the delayed $\langle x, v \rangle$ entry, as described by (M-Write). Under $\mathrm{Px86}_{\mathrm{man}}$ we must additionally check the ordering constraints between $\langle x, v \rangle$ and existing promoted entries. This is captured by the last premise of (M-Write), ensuring that later $\mathbf{flush}_{\mathrm{opt}}$ on the same cache line as well as later **sfence** and **flush** cannot be ordered before $\langle x, v \rangle$, as prescribed in Fig. 3. The (M-Read), (M-RMW) and (M-MF) rules remain unchanged from their $\mathrm{Px86}_{\mathrm{sim}}$ counterparts in Fig. 7.

As in $\mathrm{Px86}_{\mathrm{sim}}$, the (M-SF) rule describes the delayed **sfence** execution by adding $\langle \mathsf{sf} \rangle$ to b. Under $\mathrm{Px86}_{\mathrm{man}}$ we must also check the constraints between the promoted entries in b and $\langle \mathsf{sf} \rangle$, as captured by the last premise. The premises of (M-FO) and (M-FL) are analogously extended to check the constraints between the promoted entries in b and the newly added $\langle \mathsf{fo}, x \rangle$ and $\langle \mathsf{fl}, x \rangle$, respectively.

The (M-SF2) rule describes the *justification* of a promoted **sfence** execution by debuffering its entry. Let $p{=}\langle \mathsf{psf} \rangle$ denote the promoted **sfence** entry to be removed by (M-SF2). As discussed, we must check the constraints between p and other promoted entries. Let p' be a promoted entry preceding p in the buffer; i.e. p' is store-ordered before p. Note that since no instruction can be reordered with respect to **sfence** (except reads), the store order and program order between p and p' must agree: p' must be debuffered before p. That is, when p is debuffered, no promoted entry may precede it in b. As such, since by construction no delayed entry may precede p in b (see (M-ProSF)), p must be at the head of b when debuffered, as stipulated by $\mathsf{b}{=}\langle \mathsf{psf} \rangle.\mathsf{b}'$. Analogously, (M-FO2) and (M-FL2) justify the promoted execution of $\mathbf{flush}_{\mathrm{opt}}$ and **flush**, respectively.

The (M-BPropW), (M-BPropSF), (M-BPropFO), (M-BPropFL), (M-PropW) and (M-PropP) rules remain unchanged from their $\mathrm{Px86}_{\mathrm{sim}}$ counterparts.

5 THE DECLARATIVE $\mathrm{Px86}_{\mathrm{man}}$ AND $\mathrm{Px86}_{\mathrm{sim}}$ MODELS

We develop our declarative $\mathrm{Px86}_{\mathrm{sim}}$ and $\mathrm{Px86}_{\mathrm{man}}$ models as instances of the declarative persistency framework proposed by Raad et al. [2019b]. We then prove that the two $\mathrm{Px86}_{\mathrm{sim}}$ (resp. $\mathrm{Px86}_{\mathrm{man}}$) characterisations are equivalent, with the full proof given in the accompanying technical appendix.

Executions and Events. In the literature of declarative models, the traces of shared memory accesses generated by a program are commonly represented as a set of *executions*, where each execution G 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 ($\mathsf{c} \in \mathrm{PCom}$ in Fig. 5) and is a tuple of the form $e{=}\langle n, \tau, l \rangle$, where $n \in \mathbb{N}$ is the *event identifier* uniquely identifying e; $\tau \in \mathrm{TId}$ is the thread identifier of the executing thread; and $l \in \mathrm{Lab}$ is the event *label*, as defined in Fig. 6. The functions $\mathtt{loc}$, $\mathtt{val}_{\mathsf{r}}$ and $\mathtt{val}_{\mathsf{w}}$ respectively project the location, the read value and the written value of a label, where applicable. For instance, $\mathtt{loc}(l){=}x$ and $\mathtt{val}_{\mathsf{w}}(l){=}v$ for $l{=}(\mathtt{W}, x, v)$.

Definition 1 (Events). An *event* is a tuple $\langle n, \tau, l \rangle$, where $n \in \mathbb{N}$ is an event identifier, $\tau \in \mathrm{TId}$ is a thread identifier, and $l \in \mathrm{Lab}$ is an event label.

We typically use a, b and e to range over events. The functions $\mathtt{tid}$ and $\mathtt{lab}$ respectively project the thread identifier and the label of an event. We lift the label functions $\mathtt{loc}$, $\mathtt{val}_{\mathsf{r}}$ and $\mathtt{val}_{\mathsf{w}}$ to events, and given an event e, we write e.g. $\mathtt{loc}(e)$ for $\mathtt{loc}(\mathtt{lab}(e))$.

Notation. Given a relation r on a set A, we write $\mathsf{r}^?$, r^+ and r^* for the reflexive, transitive and reflexive-transitive closures of r, respectively. We write r^{-1} for the inverse of r; $\mathsf{r}|_A$ for $\mathsf{r} \cap (A \times A)$;

$[A]$ for the identity relation on A, i.e. $\{(a, a) \mid a \in A\}$; irreflexive(r) for $\nexists a.\ (a, a) \in r$; and acyclic(r) for irreflexive(r^+). We write $r_1; r_2$ for the relational composition of r_1 and r_2, i.e. $\{(a, b) \mid \exists c.\ (a, c) \in r_1 \wedge (c, b) \in r_2\}$. When A is a set of events, we write A_x for $\{a \in A \mid \texttt{loc}(a){=}x\}$, and write A_X for $\{a \in A \mid \texttt{loc}(a){\in}X\}$. Similarly, we write r_x for $r \cap (A_x \times A_x)$, and write r_X for $r \cap (A_X \times A_X)$.

Definition 2 (Executions)**.** An *execution*, $G \in \textsc{Exec}$, is a tuple $(E, I, P, \text{po}, \text{rf}, \text{mo}, \text{nvo})$, where:

- E denotes a set of *events*. The set of *read* events in E is: $R \triangleq \left\{e \in E \,\middle|\, \exists x, v.\ \texttt{lab}(e){=}(\mathsf{R}, x, v)\right\}$; the sets of *write* ($W$), *RMW* ($U$), *memory fence* ($MF$), *store fence* ($SF$), *optimised flush* ($FO$) and *flush* ($FL$) events are defined analogously. The set of *durable* events in E is: $D \triangleq W \cup U \cup FO \cup FL$.
- I is a set of *initialisation events*, comprising a single write event $w \in W_x$ for each location x.
- P is a set of *persisted events* such that $I \subseteq P \subseteq D$.
- $\text{po} \subseteq E \times E$ denotes the *'program-order' relation*, defined as a disjoint union of strict total orders, each ordering the events of one thread, with $I \times (E \setminus I) \subseteq \text{po}$.
- $\text{rf} \subseteq (W \cup U) \times (R \cup U)$ denotes the *'reads-from' relation* between events of the same location with matching values; i.e. $(a, b) \in \text{rf} \Rightarrow \texttt{loc}(a){=}\texttt{loc}(b) \wedge \texttt{val}_\texttt{w}(a){=}\texttt{val}_\texttt{r}(b)$. Moreover, rf is total and functional on its range, i.e. every read or update is related to exactly one write or update.
- $\text{mo} \subseteq E \times E$ is the *'modification-order'*, defined as the disjoint union of relations $\{\text{mo}_x\}_{x \in \text{Loc}}$, such that each mo_x is a strict total order on $W_x \cup U_x$, and $I_x \times ((W_x \cup U_x) \setminus I_x) \subseteq \text{mo}_x$.
- $\text{nvo} \subseteq D \times D$ is the *'non-volatile-order'*, defined as a strict total order on D, such that $I \times (D \setminus I) \subseteq \text{nvo}$ and $dom(\text{nvo}; [P]) \subseteq P$.

In the context of an execution graph G (we often use the "G." prefix to make this explicit), durable events are those whose effects *may* be observed when recovering from a crash. For instance, the effects of $x := v$ may be observed upon recovery if the write of v on x has persisted before the crash. As such, write events are durable. Note that durability does not reflect whether the effects of the instruction *do persist*; rather that its effect *could persist*. That is, regardless of whether the effects of $x := v$ persist, its associated label is deemed durable. By contrast, **mfence**, **sfence** and read instructions have no durable effects and their events are thus not durable.

Intuitively, the persisted events P include those durable events ($P \subseteq D$) whose effects have reached the persistent memory, and thus persisted events include initialisation writes ($I \subseteq P$). The 'modification-order' mo describes the store order on the writes and updates of each location. Analogously, the 'non-volatile-order' nvo prescribes the persist order. As such, we require that the persisted events in P be downward-closed with respect to nvo: $dom(\text{nvo}; [P]) \subseteq P$. That is, let $e_1 \cdots e_n$ denote the enumeration of D according to nvo (nvo is total on D); as P is downward-closed with respect to nvo, we know there exists $0 \leq i \leq n$ such that $e_1, \cdots, e_i \in P$ and $e_{i+1}, \cdots, e_n \in D \setminus P$.

Note that in this initial stage, executions are unrestricted in that there are few constraints on rf, mo and nvo. Such restrictions are determined by the set of model-specific *consistent* executions. We shortly define execution consistency for the Px86_{sim} and Px86_{man} models.

Chains. The traces of shared memory accesses generated by a program are commonly represented as a set of *complete* executions, i.e. those that do not crash. However, this assumption renders this model unsuitable for capturing the crashing behaviour of executions in the presence of persistent memory. Instead, Raad et al. [2019b] model an *execution chain* C as a sequence $G_1, \cdots, G_n$, with each G_i describing an execution *era* between two adjacent crashes. More concretely, when an execution of program P crashes $n{-}1$ times, this is modelled as the chain $C{=}G_1, \cdots, G_n$, where (1) G_1 describes the initial era between the start of execution up to the first crash; (2) for all $i \in \{2, \cdots, n{-}1\}$, G_i denotes the ith execution era, recovering from the $(i{-}1)^{\text{st}}$ crash; and (3) G_n describes the final execution era terminating successfully.

Definition 3 (Chains). A *chain* C is a sequence $G_1, \cdots, G_n$ of executions such that for $1 \leq i < n$ and $G_i{=}(E_i, I_i, P_i, \text{po}_i, \text{rf}_i, \text{mo}_i, \text{nvo}_i)$:

- $\forall x \in \text{Loc}.\ \exists w.\ w \in I_1 \wedge \texttt{loc}(w){=}x \wedge \texttt{val}_\texttt{w}(w){=}0$;
- $\forall x \in \text{Loc}.\ \exists w, e.\ w \in I_{i+1} \wedge \texttt{loc}(w){=}x \wedge e{=}\max\left(\text{nvo}_i|_{P_i \cap (W_x \cup U_x)}\right) \wedge \texttt{val}_\texttt{w}(w){=}\texttt{val}_\texttt{w}(e)$;
- $P_n = D_n$.

Given a memory model $\mathcal{M}$, a chain $C{=}G_1, \cdots, G_n$ is $\mathcal{M}$-*valid* if each G_i is $\mathcal{M}$-consistent.

The first axiom ensures that in the first era all locations are initialised with 0. The second axiom ensures that in each subsequent $(i{+}1)^{\text{st}}$ era all locations are initialised with a value persisted by a write (or update) in the previous (i^{th}) era maximally (in nvo_i). The last axiom ensures that the final era executes completely (does not crash) by stipulating that all its durable events be persisted. That is, in the absence of a crash, all durable events are eventually persisted.

Persistent Programs. In the persistent setting each program P is associated with a *recovery mechanism* describing the code executed upon recovery from a crash. Raad et al. [2019b] thus model a *persistent program*, $\mathbb{P} \in \text{PProg}$, as a pair $\langle \mathsf{P}, \texttt{rec}\rangle$, where $\mathsf{P} \in \text{Prog}$ denotes the original program, and rec denotes its recovery mechanism. A recovery mechanism is analogous to a recovery program (see §4), except that a recovery mechanism operates on an execution $G \in \text{Exec}$, rather than a memory $\mathsf{M} \in \text{Mem}$. That is, a recovery mechanism is a function $\texttt{rec} : \text{Prog} \times \text{Exec} \rightarrow \text{Prog}$, where $\texttt{rec}(\mathsf{P}, G)$ describes the recovery mechanism associated with P when the memory upon recovery is that obtained after execution G. Intuitively, G corresponds to the previous execution era and thus the state of memory upon recovery can be ascertained by inspecting G.

From Programs to Chains. The *semantics* of a program P is typically defined as a set of consistent executions associated with P and is defined by straightforward induction on the structure of P. Analogously, Raad et al. [2019b] define the semantics of a persistent program $\mathbb{P}$ as a set of valid chains associated with $\mathbb{P}$. More concretely, a persistent program $\mathbb{P}{=}\langle \mathsf{P}, \texttt{rec}\rangle$ is associated with a valid chain $C{=}G_1, \cdots, G_n$ if: (1) G_1 is a partial execution of P; (2) G_i is a partial execution of $\texttt{rec}(\mathsf{P}, G_{i-1})$, for all $2 \leq i \leq n{-}1$; and (3) G_n is an execution of $\texttt{rec}(\mathsf{P}, G_{n-1})$. Note that executions of all but the last era are partial in that they have failed to run to completion due to a crash. We refer the reader to [Raad et al. 2019b] for the formal definition of the semantics of persistent programs.

Definition 4 (Px86_{sim} and Px86_{man} consistency). An execution $(E, I, P, \text{po}, \text{rf}, \text{mo}, \text{nvo})$ is *Px86_{sim}-consistent* iff there exists a strict order, $\text{tso} \subseteq E \times E$, such that the Px86_{sim} axioms in Fig. 10 hold.

An execution $(E, I, P, \text{po}, \text{rf}, \text{mo}, \text{nvo})$ is *Px86_{man}-consistent* iff there exists a strict order, $\text{tso} \subseteq E{\times}E$, such that the Px86_{man} axioms in Fig. 10 hold.

The Px86-consistency axioms are given in Fig. 10, where the right column describes how each axiom corresponds to the ordering constraints in Fig. 3. The (tso-mo)–(tso-mf) axioms are those of the *original* x86-TSO model defined by Sewell et al. [2010], which requires the existence of a strict order tso ('total store order') on events. Intuitively, tso describes the store order, i.e. the order in which events are made visible to other threads, which corresponds the order in which events reach the persistent buffer in the operational Px86 model. As such, tso subsumes the modification order mo (tso-mo). The (tso-po) and (tso-mf) axioms capture the ordering constraint in the original x86-TSO model, as illustrated in the top-left corner of Fig. 3. For instance, the $[E]; \text{po}; [MF] \subseteq \text{tso}$ requirement of (tso-mf) describes column 4 of Fig. 3, while $[MF]; \text{po}; [E] \subseteq \text{tso}$ describes row D.

Note that these original axioms do not account for store fences or persists (highlighted in Fig. 3); nor have they any bearing on the Px86 persistency semantics as they impose no constraints on nvo. We capture the constraints of store fences and persists via (tso-sf)–(tso-w-fo). We describe the Px86 persistency semantics via (nvo-loc)–(nvo-fofl-d).

$\text{Px86}_{\text{sim}}/\text{Px86}_{\text{man}}$ **Axiom**	**Label**	✓ **in Fig. 3**
$\text{mo} \subseteq \text{tso}$	(TSO-MO)	
tso is total on $E \setminus R$	(TSO-TOTAL)	
$\text{rf} \subseteq \text{tso} \cup \text{po}$	(TSO-RF1)	
$\forall x \in \text{Loc}. \forall (w, r) \in \text{rf}_x. \forall w' \in W_x \cup U_x. (w', r) \in \text{tso} \cup \text{po} \Rightarrow (w, w') \notin \text{tso}$	(TSO-RF2)	
$([W \cup U \cup R]; \text{po}; [W \cup U \cup R]) \setminus (W \times R) \subseteq \text{tso}$	(TSO-PO)	A1–C4
$([E]; \text{po}; [MF]) \cup ([MF]; \text{po}; [E]) \subseteq \text{tso}$	(TSO-MF)	Col. 4, Row D
$([E \setminus R]; \text{po}; [SF]) \cup ([SF]; \text{po}; [E \setminus R]) \subseteq \text{tso}$	(TSO-SF)	Col. 5, Row E
$([W \cup U \cup FL]; \text{po}; [FL]) \cup ([FL]; \text{po}; [W \cup U \cup FL]) \subseteq \text{tso}$	(TSO-FL-WUFL)	B7,C7,G2,G3,G7
$\forall X \in \text{CL}. ([FL_X]; \text{po}; [FO_X]) \cup ([FO_X]; \text{po}; [FL_X]) \subseteq \text{tso}$	(TSO-FL-FO)	G6, F7
$([U]; \text{po}; [FO]) \cup ([FO]; \text{po}; [U]) \subseteq \text{tso}$	(TSO-FO-U)	C6, F3
$\forall X \in \text{CL}. ([W_X]; \text{po}; [FO_X]) \subseteq \text{tso}$	(TSO-W-FO)	B6
$\forall x \in \text{Loc}. \text{tso}\vert_{D_x} \subseteq \text{nvo}$	(NVO-LOC)	
$\forall X \in \text{CL}. [W_X \cup U_X]; \text{tso}; [FO_X \cup FL_X] \subseteq \text{nvo}$	(NVO-WU-FOFL)	
$[FO \cup FL]; \text{tso}; [D] \subseteq \text{nvo}$	(NVO-FOFL-D)	
Px86_{sim} **Axiom**	**Label**	✓ **in Fig. 3**
$[R]; \text{po}; [SF] \subseteq \text{tso}$	(TSO-R-SF)	A5
$[R]; \text{po}; [FO \cup FL] \subseteq \text{tso}$	(TSO-SIM)	A6, A7

Fig. 10. Declarative Px86_{sim} and Px86_{man} axioms

Axiom (TSO-SF) states that store fences are ordered (in both directions) with respect to all events but reads. Similarly, (TSO-FL-WUFL)–(TSO-FL-FO) ensure that **flush** is ordered (in both directions) with respect to writes, RMWs, flushes, as well as $\textbf{flush}_{\text{opt}}$ on the same cache line. Analogously, (TSO-FO-U) states that $\textbf{flush}_{\text{opt}}$ is ordered (in both directions) with respect to RMWs; (TSO-W-FO) ensures that $\textbf{flush}_{\text{opt}}$ is ordered with respect to po-earlier writes on the same cache line.

Recall from §2 that for each location x, its store and persist orders coincide. This is captured by (NVO-LOC). Moreover, as discussed in §2, given $x \in X$ and $\text{C}=\textbf{flush}_{\text{opt}}\ x$ or $\text{C}=\textbf{flush}\ x$, all writes on X (store-) ordered before C persist before all instructions (store-) ordered after C, regardless of their cache line. This is captured by (NVO-WU-FOFL) and (NVO-FOFL-D).

Lastly, recall that in addition to the Px86_{man} constraints, Px86_{sim} further requires that the following orders be preserved: (1) the order between earlier reads and later **sfence** instructions; and (2) the order between earlier reads and later $\textbf{flush}_{\text{opt}}/\textbf{flush}$ instructions as stipulated by (SIM). This is captured by the (TSO-R-SF) and (TSO-SIM) axioms at the bottom of Fig. 10.

***x86-TSO and* sfence.** Observe that the original x86-TSO model by Sewell et al. [2010] does not include the (TSO-SF) axiom. This is because: (1) the x86-TSO model simply does not model **sfence** instructions; and (2) extending x86-TSO with **sfence** is trivial in that in the absence of $\textbf{flush}_{\text{opt}}/\textbf{flush}$ operations, **sfence** instructions behave as *no-ops*. This is because as stated in (TSO-SF), **sfence** instructions enforce a tso order between po-earlier and po-later *non-read* instructions. As such, in the absence of $\textbf{flush}_{\text{opt}}/\textbf{flush}$ (i.e. only considering read, write, RMW, **mfence** and **sfence** instructions), this enforces a tso order between po-earlier and po-later write/RMW/**mfence** instructions. However, this constraint is already captured by (TSO-PO) and (TSO-MF); i.e. **sfence** instructions introduce no additional ordering constraints. This is no longer the case when considering $\textbf{flush}_{\text{opt}}/\textbf{flush}$ instructions, and we thus explicitly capture the **sfence** constraints in (TSO-SF).

5.1 Equivalence of the Px86 Operational and Declarative Semantics

The Px86_{sim} and Px86_{man} operational models are *equivalent* to their counterpart declarative models, as stated in Thm. 1 and Thm. 2 below. We proceed with an intuitive account of the equivalence proofs; we refer the reader to the accompanying technical appendix for the full proof.

Px86_{sim} Equivalence. Let $\mathsf{P}_{\mathbf{skip}} \triangleq \lambda\tau.v \in \textsc{Val}$ denote a terminated program. To establish the equivalence of the two Px86_{sim} models, we must show that for all programs P, if $\mathsf{P}, \mathsf{M}_0, \mathsf{PB}_0, \mathsf{B}_0 \Rightarrow^* \mathsf{P}_{\mathbf{skip}}, \mathsf{M}, \mathsf{PB}, \mathsf{B}$, then we can construct a corresponding Px86_{sim}-valid chain; and vice versa.

To this end, we develop an *intermediate* Px86_{sim} semantics as an *event-annotated* transition system. More concretely, we describe the intermediate semantics by separating the transitions of its program and storage subsystems, as before. The transitions of the *annotated program subsystem* are of the form $\mathsf{P} \xrightarrow{\lambda} \mathsf{P}'$, where λ is an *annotated label*, recording the memory event $e \in E$ (Def. 1) making the transition. For instance, when executing an **sfence** instruction the annotated label is $\lambda{=}\mathsf{SF}\langle sf \rangle$, where $sf \in SF$ is a store fence event. Such explicit tracking of events in the operational semantics allows for a simple construction of the constituent events in the corresponding Px86_{sim} executions. Similarly, when executing a read instruction $a := x$, the annotated label is $\lambda{=}\mathsf{R}\langle r, e \rangle$, where $r \in R$ is a *read* event, and $e \in W \cup U$ is a write/update event, denoting the event responsible for writing the value read by r. That is, e denotes the write/update event that r *reads from*. Tracking the write/update events this way allows us to construct the rf relation when constructing the corresponding Px86_{sim} executions. Moreover, when $\mathsf{P} \xrightarrow{\lambda_1} \cdots \xrightarrow{\lambda_2} -$, we can determine the *program order* between the events associated with λ_1 and λ_2. For instance, when $\lambda_1{=}\mathsf{SF}\langle sf \rangle$, $\lambda_2{=}\mathsf{R}\langle r, - \rangle$ and $\mathtt{tid}(sf){=}\mathtt{tid}(r)$, then sf is po-before r in the corresponding Px86_{sim} execution.

Similarly, the transitions of the storage subsystem are of the form $M, PB, B \xrightarrow{\lambda} M', PB', B'$, where M, M' are the *event-annotated memory*; PB, PB' are the *event-annotated persistent buffer*; and B, B' are the *event-annotated buffer maps*. An annotated memory M is a map from locations to *write/update* events. That is, for each location x, rather than recording its value, we record the write/update event responsible for setting x to its current value. An annotated persistent buffer PB is analogously augmented to record the write/update/$\mathbf{flush}_{\text{opt}}$/**flush** *events* to be propagated; similarly for B. Additionally, when a delayed event e is debuffered from a thread buffer in B, we mark this transition with the label $\lambda{=}\mathsf{B}\langle e \rangle$. Recall that the order in which delayed entries are debuffered amounts to their store order (the order they become visible to other threads). As such, we can track the debuffering order via $\mathsf{B}\langle . \rangle$ transitions, allowing us to construct the tso relation of the corresponding Px86_{sim} execution. That is, when $M, PB, B \xrightarrow{\lambda_1} \cdots \xrightarrow{\lambda_2} -$ with $\lambda_1{=}\mathsf{B}\langle e_1 \rangle$ and $\lambda_2{=}\mathsf{B}\langle e_2 \rangle$, then e_1 is tso-before e_2 in the corresponding execution. Analogously, when an event e is debuffered from PB, we mark this transition with $\lambda{=}\mathsf{PB}\langle e \rangle$. This in turn allows us to construct the nvo relation of the execution.

The intermediate semantics is obtained by combining its program and storage transitions. The combined transitions are of the form: $\mathsf{P}, M, PB, B, \mathcal{H}, \pi \Rightarrow \mathsf{P}', M', PB', B', \mathcal{H}', \pi'$. The π denotes the *execution path* in the current execution era, modelled as a sequence of annotated labels of the form $\lambda_1.\cdots.\lambda_n$. That is, each time the combined system takes a λ step, the current execution path is extended by appending λ at the end. Recording the execution path in π allows us to construct the po, tso and nvo relations of the current execution era, as discussed above. The *execution history*, $\mathcal{H}$, tracks the execution paths of the previous eras. That is, if the execution has encountered n crashes thus far, then $\mathcal{H}$ contains n entries, $\pi_1, \cdots, \pi_n$, with each π_i tracking the execution path in the ith era. Recording the history $\mathcal{H}$ allows us to construct the execution graphs of the previous eras.

To prove the equivalence of our two Px86_{sim} semantics we show: (i) the Px86_{sim} operational semantics is equivalent to the Px86_{sim} intermediate semantics; and (ii) the Px86_{sim} intermediate

semantics is equivalent to the Px86_{sim} declarative semantics. Proof of part (i) is by straightforward induction on the structure of $\Rightarrow$ transitions. Proof of part (ii) is more involved. As discussed above, to construct Px86_{sim}-valid execution chains from the intermediate semantics, we appeal to the events in the storage subsystem, as well as the order of annotated labels in execution paths and histories. Dually, to construct the intermediate transitions given a Px86_{sim}-valid chain C, we construct the relevant execution path and histories from the executions in C and their sundry relations.

Theorem 1 (Px86_{sim} Equivalence)**.** *The declarative and operational $Px86_{sim}$ models are equivalent.*

$Px86_{man}$ Equivalence. As with Px86_{sim}, the Px86_{man} equivalence proof is done in two steps via an intermediate event-annotated operational semantics. The Px86_{man} intermediate semantics is an extension of Px86_{sim} where the annotated thread buffers may additionally contain promoted events. For instance, upon promoted execution of a **flush** instruction, the intermediate semantics executes a transition with the annotated label $\lambda{=}\mathtt{PFL}\langle fl \rangle$, where $fl \in FL$ is a **flush** event. Analogously, when a promoted event e is debuffered, the transition is labelled with $\lambda{=}\mathtt{J}\langle e \rangle$, justifying the promoted execution of e (see §4). Recall that the order in which promoted entries are added (resp. removed) amounts to their store (resp. program) order. As such, we can construct the tso/po of promoted events by tracking their addition/removal orders. For instance, when $M, PB, B \xrightarrow{\lambda_1} \cdots \xrightarrow{\lambda_2} -$, then in the corresponding execution: (1) e_1 is tso-before e_2 when e.g. $\lambda_1{=}\mathtt{PFL}\langle e_1 \rangle$ and $\lambda_2 \in \{\mathtt{PFL}\langle e_2 \rangle, \mathtt{B}\langle e_2 \rangle\}$; and (2) e_1 is po-before e_2 when e.g. $\lambda_1{=}\mathtt{J}\langle e_1 \rangle$, $\lambda_2 \in \{\mathtt{J}\langle e_2 \rangle, \mathtt{SF}\langle e_2 \rangle\}$ and $\mathtt{tid}(e_1){=}\mathtt{tid}(e_2)$. The construction of the remaining relations (i.e. nvo and rf), as well as execution chains is as in Px86_{sim}.

Theorem 2 (Px86_{man} Equivalence)**.** *The declarative and operational $Px86_{man}$ models are equivalent.*

6 PERSISTENT LIBRARIES IN Px86

As discussed thus far, *correct* persistent programming is difficult as it requires accounting for the crashing behaviour of programs. This difficulty is further compounded when programming on top of low-level models such as Px86. This is because developing at this low-level does not afford high-level abstractions such as concurrency control, and requires an understanding of hardware-specific guarantees. In an effort to make persistent programming accessible to the uninitiated programmer, recent NVM literature explores simpler notions of persistency via *high-level persistent libraries*, which may be implemented over existing hardware models (e.g. Px86), while shielding their clients from low-level hardware details. In what follows we present two such libraries we implement over Px86: a persistent transactional library, and a persistent variant of the Michael–Scott queue.

6.1 A Persistent Transactional Library in Px86

Transactions are a concurrency control mechanism used widely to ensure the consistency of shared data, both in the distributed setting (in databases), and the shared-memory concurrency setting (as transactional memory, e.g. [Raad et al. 2018, 2019a]). Transactions ensure that a given block of code executes *atomically*: at all times during the execution of a transaction **Tx** either all or none of the writes in **Tx** are visible to other threads. This coarse-grained atomicity abstraction simplifies correct concurrent programming. As such, NVM researchers have sought to extend atomicity guarantees of transactions to persistency: upon recovery, either all or none of the writes of a transactions may have persisted [Avni et al. 2015; Intel 2015; Kolli et al. 2016a; Shu et al. 2018; Tavakkol et al. 2018].

To formalise the semantics of such persistent transactions, Raad et al. [2019b] proposed the PSER (persistent serialisability) model by extending the well-known transactional model, serialisability, to the NVM setting. PSER is a simple model with strong and intuitive guarantees. As such, we present an implementation of PSER and its recovery mechanism in Px86. We show that our implementation is *sound*, i.e. it provides PSER guarantees, thereby demonstrating correct Px86-to-PSER compilation.

We proceed with a brief account of PSER, and then describe our PSER implementation. We refer the reader to the technical appendix for the details of our implementation and its correctness proof.

The PSER Model. A *transaction* typically describes a block of code that executes *atomically*, ensuring that the transactional writes exhibit an all-or-nothing behaviour. For instance, consider the transaction: **Tx** $[x := 1;\ y := 1]$. If initially $x{=}y{=}0$, at all points during the execution of **Tx** either $x{=}y{=}0$ or $x{=}y{=}1$. Under *serialisability* (SER), all concurrent transactions appear to execute one after another in a total sequential order. Consider the transactional program (PTx) below (left):

$$\textbf{Tx1:}\begin{bmatrix} x := 1; \\ \mathsf{b} := y; \end{bmatrix} \Bigg\| \ \textbf{Tx2:}\begin{bmatrix} y := 1; \\ \mathsf{a} := x; \end{bmatrix} \quad \text{(PTx)} \qquad \textbf{Tx3:}\begin{bmatrix} x := 1; \\ y := 1; \end{bmatrix} \Bigg\| \ \textbf{Tx4:}\begin{bmatrix} \mathsf{a} := x; \\ \textbf{if } \mathsf{a} > 0 \textbf{ then } z := 1; \end{bmatrix} \quad \text{(PTx2)}$$

Assuming $x{=}y{=}0$ initially, under SER either **Tx1** executes before **Tx2** and thus $\mathsf{a}{=}1, b{=}0$; or **Tx2** executes before **Tx1** and thus $\mathsf{a}{=}0, b{=}1$. Serialisability is the gold standard of transactional models as it provides strong guarantees with simple intuitive semantics. Raad et al. [2019b] develop PSER by extending the atomicity and ordering guarantees of SER to persistency. That is, PSER provides (1) *persist atomicity*, ensuring that the persists in a transaction exhibit an all-or-nothing behaviour. For instance, if a crash occurs during the execution of **Tx** in the example above, upon recovery either $x{=}y{=}0$ or $x{=}y{=}1$. Moreover, PSER guarantees (2) *strict, buffered persistency* in that all concurrent transactions appear to persist one after another in the *same* total sequential order in which they (appear to) have executed. As such, upon recovery a *prefix* of the transactions in the total order may have persisted. For example, consider the (PTx2) program above and assume that **Tx3** executes before **Tx4**. If the execution of (PTx2) crashes, under PSER $z{=}1 \Rightarrow x{=}y{=}1$ upon recovery. That is, if **Tx4** has persisted ($z{=}1$), then the earlier transaction **Tx3** must have also persisted ($x{=}y{=}1$).

Our PSER Implementation in Px86. A transactional implementation in the NVM context comprises two orthogonal components: persistency control and concurrency control. The former choice is tied to the hardware platform over which transactions are implemented. As we implement PSER over Px86, we thus use the Px86 persistency primitives ($\mathbf{flush}_{\text{opt}}$ and **sfence**).

For the latter we can use either (1) *hardware* concurrency control, e.g. hardware transactional memory (HTM); or (2) *software* concurrency control, e.g. locks. The Intel-x86 architecture provides HTM support via *Transactional Synchronization Extensions* (TSX), and can thus be used to implement persistent transactions. However, a well-known HTM limitation in general is that transactions may arbitrarily *abort*: HTM systems can only commit transactions whose memory footprint do not exceed the cache capacity. Moreover, even if the cache capacity is not exceeded, HTM systems typically provide no guarantee that transactions will commit. As such, applications using HTM must rely on a (software) fallback mechanism in cases of transactional aborts. One such mechanism is to rerun the aborted transaction by acquiring a designated single global lock (SGL). In the Intel-x86 hardware the likelihood of aborts is further increased when using persistency primitives:

> 'The $\mathbf{flush}_{\text{opt}}$ *[resp.* **flush** *[Intel 2019, p. 3-142]]* instruction may always cause transactional abort with Transactional Synchronization Extensions (TSX). The $\mathbf{flush}_{\text{opt}}$ instruction is not expected to be commonly used inside typical transactional regions.' [Intel 2019, p. 3-144]

As we need to use **flush**/$\mathbf{flush}_{\text{opt}}$ within transactional regions to ensure persist atomicity, our implementation must account for the high probability of aborts by providing a software contingency (e.g. SGL). As such, we forgo the Intel-x86 HTM (TSX) altogether, and instead resort to software concurrency control via locks. In particular, as serialisability allows concurrent transactions to read the same memory location simultaneously, for better performance we use multiple-readers-single-writer locks implemented using Intel-x86 atomic primitives (**FAA** and **CAS**).

Intuitively, given a transaction T of thread τ, our PSER implementation of T adheres to the following pattern: (1) it updates and persists (using $\mathbf{flush}_{\text{opt}}$) the *metadata* tracking the progress

of τ in a designated log; (2) executes an **sfence**; (3) executes and persists T; and (4) executes an **sfence**. The first two steps ensure that the recovery metadata of τ does not lag behind its progress; conversely, the last two steps ensure that the progress of τ does not lag behind its recovery metadata. Therefore, in case of a crash, the persisted progress of τ may be at most one step behind its persisted metadata. Upon a crash, the recovery mechanism can then inspect the τ metadata to infer its progress and complete the execution of its last transaction, if necessary.

6.2 A Persistent Michael–Scott Queue Library in Px86

Implementing and verifying concurrent libraries is a challenging undertaking. Traditionally, library *implementers* are tasked with ensuring the underlying library state (e.g. a queue) remains *consistent* (e.g. the queue maintains its FIFO property), when accessed concurrently. Library *verifiers* are tasked with identifying the appropriate proof techniques to establish the desired consistency guarantees. One well-known such technique is that of *linearisability* proofs by Herlihy and Wing [1990], and has been used extensively in the verification literature. Both tasks of implementing and verifying concurrent libraries are exacerbated in the presence of NVM. Library implementers must ensure the library state remains both consistent *and* persistent in case of crashes; library verifiers must accordingly adapt their formal techniques to establish the desired persistence properties. While persistent libraries have been the subject of recent research [Chatzistergiou et al. 2015; Coburn et al. 2011; Friedman et al. 2018], few implementations have been formally verified. In order to verify the correctness of persistent library implementations, Izraelevitz et al. [2016b] introduced the notion of *persistent linearisability* as an extension of linearisability in the NVM context.

Our Persistent MSQ Implementation. We implement a *persistent* variant of the wait-free Michael–Scott queue (MSQ) [Michael and Scott 1996] and its recovery mechanism over Px86. Our MSQ implementation is instrumented with additional metadata to track the progress of queue operations, and follows a similar pattern to that discussed above. That is, when thread τ executes enq(v) (resp. deq(v)), our MSQ implementation (1) updates and persists the *metadata* of τ to reflect its intention to enqueue (resp. dequeue) v; (2) executes an **sfence**; (3) enqueues (resp. dequeues) v; and (4) executes an **sfence**. As such, as with our PSER implementation, upon recovering from a crash the persisted progress of each thread may be at most one step behind its persisted metadata and thus its execution can be resumed accordingly. We prove that our MSQ implementation (together with its recovery program) is *sound* in that it is persistently linearisable. We refer the reader to the accompanying technical appendix for the details of our implementation and its correctness proof.

7 AUTOMATED LITMUS TEST GENERATION

We describe how we use our declarative Px86 axioms in Def. 4 to generate litmus tests such as those in Fig. 1 *automatically*. This enables us to explore program behaviours that are forbidden by our declarative axioms, and indeed guided our process of formalising Px86 by identifying its corner cases, and subsequently discussing them with research engineers at Intel. We believe that this is beneficial to those seeking to understand and program over Px86.

Test generation from declarative memory *consistency* axioms is well understood in the literature [Chong et al. 2018; Lustig et al. 2017]. The key observation is that one can use a solver such as Alloy [Jackson 2012] to search for executions that violate the given axioms, and then to construct from each execution a litmus test that passes only when that particular execution has occurred. The main challenge is to ensure that every component that appears in the execution is reflected in the litmus test. To this end, write/read events in the execution become **store**/**load** instructions in the litmus test, the desired rf edges are checked in the postcondition of the litmus test, and the desired mo edges are checked by polling the relevant memory locations.

We extend these techniques to handle memory *persistency* axioms. Given an execution (E, I, P, po, rf, mo, nvo), our main challenge is constructing a litmus test that checks for the desired nvo relation. This is difficult because nvo cannot be observed directly. In the absence of hardware customised for monitoring the memory system, the only opportunity to observe nvo is in the post-recovery state; even then, all one can infer is that certain events have persisted (i.e. are in P) and others have not.

To address this, we rewrite our persistency axioms such that they constrain P rather than nvo. That is, our executions need not mention nvo at all as it cannot be observed. Moreover, excluding nvo has the added benefit of reducing the number of relations over which our constraints need to quantify. More concretely, recall from Def. 4 that the (NVO-LOC), (NVO-WU-FOFL), and (NVO-FOFL-D) axioms constrain nvo, and that nvo and P are related by the $dom(\text{nvo}; [P]) \subseteq P$ constraint in Def. 2. As such, we can rewrite (NVO-LOC), (NVO-WU-FOFL), and (NVO-FOFL-D) as follows:

- $dom((r_1 \cup r_2 \cup r_3); [P]) \subseteq P$ (P-TSO)
  with $r_1 = \bigcup_{x \in \text{Loc}} \text{tso}|_{D_x}$ $\quad r_2 = \bigcup_{X \in \text{CL}} [W_X \cup U_X]; \text{tso}; [FO_X \cup FL_X]$ $\quad r_3 = [FO \cup FL]; \text{tso}; [D]$

The (P-TSO) axiom states that if event e has persisted, and e' precedes e in r_1, r_2, or r_3 (borrowed from (NVO-LOC), (NVO-WU-FOFL) and (NVO-FOFL-D), respectively) then e' must also have persisted.

However, we must still weaken (P-TSO) to make it more suitable for test generation by removing constraints that are *not observable*, as follows. First, we cannot observe a (P-TSO) violation when two persist instructions (e.g. two flushes) persist in the wrong order. Second, we cannot observe a (P-TSO) violation when two writes or RMWs on the *same* location persist in the wrong order: when observing the post-recovery state we cannot infer whether the first write persisted and was then overwritten by the second, or that the first write did not persist at all. Therefore, we weaken (P-TSO) so that it only constrains writes and updates (and not $\textbf{flush}_{\text{opt}}$ and **flush**) on *different* locations:

- $dom((r_1 \cup r_2 \cup r_3)^+|_{W \cup U} \setminus \text{sloc}); [P]) \subseteq P$ (P-TSO-WU)

where $\text{sloc} \triangleq \{(e, e') \mid \texttt{loc}(e) = \texttt{loc}(e')\}$ is an equivalence relation between events on the same location. Note that weakening (P-TSO) to (P-TSO-WU) compromises *completeness* in that we may not generate all litmus tests. However, this is to be expected as our test generation method aims to be useful (informative) rather than complete (exhaustive). As we discuss in §8, in the future we intend to investigate complete verification techniques such as model checking.

We can now encode our constraints in Alloy. Our aim is to capture executions that witness the *difference* between the original x86-TSO model [Sewell et al. 2010] and our Px86 extensions. To this end, we define an execution to be *indicative* if it satisfies *all* original x86-TSO axioms and violates *at least one* Px86 axiom. Recall that both x86-TSO and Px86 models require the existence of a 'total-store-order' relation tso (see Def. 4). Note that quantifying the tso relation correctly requires additional care: while our first instinct may be to quantify tso existentially, we must rather quantify it *universally* to ensure that the result of the litmus test is indicative regardless of how tso is resolved at runtime. For Px86_{man} we thus define indicative(G) such that *for all* tso:

if (G, tso) satisfies all of (TSO-MO, TSO-TOTAL, TSO-RF1, TSO-RF2, TSO-PO, TSO-MF)
then (G, tso) violates some of (TSO-SF, TSO-FL-WUFL, TSO-FL-FO, TSO-FO-U, TSO-W-FO, P-TSO-WU)

We then ask Alloy to find executions G such that:

- indicative(G) holds; and
- for all G', if G' is obtained from G by either (1) removing an event (and all its incident edges), or (2) weakening an RMW to a write, or (3) weakening a **flush** to a $\textbf{flush}_{\text{opt}}$, or (4) weakening an **mfence** to an **sfence**, then indicative(G') does not hold.

The second constraint above captures a notion of *minimality*: if Alloy reports G as indicative then no smaller or weaker execution G' is indicative. We encode minimality in Alloy using the 'relation perturbations' method of Lustig et al. [2017]. The universal quantification over tso necessitates the

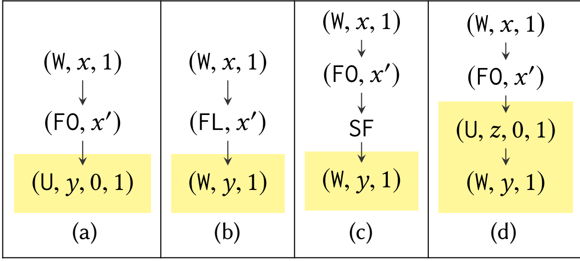


Fig. 11. Example indicative executions found by Alloy where ↓ denote po edges, highlighted events denote persisted events (in P), and the forbidden post-recovery state is $y=1 \land x=0$.

#Events	#Generated Execs.	Time Taken
2	0	3 seconds
3	20	45 minutes
4	118	> 2 hours
5	1143	> 2 hours
6	1173	> 2 hours

Table 1. Execution generation statistics

higher-order solver Alloy* [Milicevic et al. 2015], which is only feasible for very small executions; we explored executions with up to six events.

Fig. 11 presents several indicative executions discovered by Alloy. Observe that there are no executions with two events. This is in line with our examples in Fig. 1; in particular, note that the program in Fig. 1a has no indicative executions. With an upper bound of three events, Alloy reports executions such as those in Fig. 11a and Fig. 11b. These executions correspond precisely to the litmus tests of Fig. 1b and Fig. 1d. The only other three-event executions reported by Alloy are stronger variants of the above, in which x and y are the same location or are in the same cache line—we considered imposing further minimality constraints to exclude these executions, but doing so hampers Alloy's performance. With an upper bound of four events, Alloy reports executions such as those in Fig. 11c, and Fig. 11d. Fig. 11c corresponds to the litmus test in Fig. 1e, and Fig. 11d demonstrates that RMWs can replace store fences.

Results and Experimental Setup. Table 1 shows the results of generating indicative executions with event bounds between two and six. We used the Glucose SAT solver as the backend of Alloy, on a machine with a 3.1 GHz Intel Core processor and 8 GB of RAM. Execution generation timed out at two hours; i.e. for event bounds of four or higher we could generate only a sample of executions.

The generated executions were useful for validating our Px86 axioms, and we believe they will prove even more valuable as a conformance suite once a systematic way is developed to execute persistent litmus tests on NVM technology (see §8).

8 RELATED AND FUTURE WORK

Although the existing literature on non-volatile memory has grown rapidly in the recent years, *formalising* persistency models has largely remained unexplored.

Hardware Persistency Models. Existing literature includes several hardware persistency models. Pelley et al. [2014] describe *epoch persistency* under sequential consistency [Lamport 1979], whilst Condit et al. [2009]; Joshi et al. [2015] describe epoch persistency under x86-TSO [Sewell et al. 2010]. However, neither work provides a *formal* description of the studied persistency semantics. Liu et al. [2019] develop the PMTest testing framework for finding persistency bugs in software running over hardware models. However, they do not formalise the persistency semantics of the underlying hardware. Izraelevitz et al. [2016b] give a formal declarative semantics of epoch persistency under release consistency[Gharachorloo et al. 1990]; Raad and Vafeiadis [2018] propose the PTSO model by formalising epoch persistency under x86-TSO (see below); Raad et al. [2019b] develop the PARMv8 model, formalising the persistency semantics of the ARMv8 architecture. However, neither work formalises the persistency semantics of the ubiquitous Intel-x86 architecture.

The PTSO Model. The PTSO model of Raad and Vafeiadis [2018] is a persistency *proposal* for Intel-x86, and is rather different from the existing Intel-x86 model described by Intel [2019]. More concretely, PTSO does not support the fine-grained primitives of Intel-x86 for *selectively* persisting cache lines (**wb**, $\textbf{flush}_{\text{opt}}$ and **flush**), and instead proposes two coarse-grained instructions for persisting *all* locations (**pfence** and **psync**) which do not exist in Intel-x86. Moreover, all Intel-x86 persistence primitives are asynchronous, while the **psync** primitive of PTSO is synchronous. That is, executing an Intel-x86 **wb**/$\textbf{flush}_{\text{opt}}$/**flush** primitive does not block execution until the corresponding cache line is persisted, and merely guarantees that it will be persisted at a future time. By contrast, executing a **psync** blocks until all pending writes on all locations have persisted.

The PARMv8 Model. The work of Raad et al. [2019b] on the PARMv8 model differs from ours in two ways. First, PARMv8 is described *only* declaratively and *not operationally*. Second, persistency support in ARMv8 is more limited than in Intel-x86: ARMv8 provides a single persistence primitive, **DC CVAP**, which persists a given cache line to memory, whilst Intel-x86 provides three persistence primitives, **wb**, $\textbf{flush}_{\text{opt}}$ and **flush**, varying both in performance and in strength (their ordering constraints). As such, the Px86 persistency axioms are more sophisticated than those of PARMv8.

Software Persistency Models. The existing literature on software persistency is more limited. Kolli et al. [2017] propose *acquire-release persistency* (ARP), an analogue to release-acquire consistency in C/C++. Gogte et al. [2018] propose *synchronisation-free regions* (regions delimited by synchronisation operations or system calls). Both approaches enjoy good performance and can be efficiently used by seasoned persistent programmers. Nevertheless, their semantic models are low-level, rendering them too complex for the inexperienced developers. The NVM community has thus moved towards high-level transactional approaches [Avni et al. 2015; Intel 2015; Kolli et al. 2016a; Raad et al. 2019b; Shu et al. 2018; Tavakkol et al. 2018], such as our PSER library in Px86.

Future Work. We plan to build on top of this work in several ways. First, we plan to explore *language-level* persistency by researching persistency extensions of high-level languages such as C/C++. This would make persistent programming more accessible as it liberates developers from understanding hardware-specific persistency guarantees. Second, we plan to specify and verify existing persistent libraries, including more elaborate variants of our implementations in §6 (e.g. PMDK transactions of Intel [2015]), as well as other persistent data structures (e.g. sets in [Cooper 2008; PCJ 2016; Zuriel et al. 2019]). Recall from §7 that a key challenge of testing persistency is that nvo (the persist order) is not directly observable. To address this, as a third direction of future work we intend to build custom hardware that allows us to monitor the traffic to persistent memory, and thus to observe nvo directly. This can be achieved when the processor under test is a component of a system-on-chip (SoC) FPGA [Jain et al. 2018]. We can then use our Alloy-generated litmus tests as a conformance suite to test for nvo violations. Lastly, we plan to develop *complete* verification techniques, such as stateless model checking (SMC), for NVM. We plan to do this by extending our existing SMC techniques [Kokologiannakis et al. 2019a,b] for verifying concurrent libraries under weak consistency [Raad et al. 2019]. This would allow us to verify a persistent program by exhaustively generating its executions and inspecting them for persistency violations.

ACKNOWLEDGMENTS

We thank Nathan Chong, Stephen Dolan, William Wang, and the POPL 2020 reviewers for helpful discussions and valuable feedback. The first author was supported in part by a European Research Council (ERC) Consolidator Grant for the project "RustBelt", under the European Union Horizon 2020 Framework Programme (grant agreement number 683289). The second author was supported in part by the EPSRC grant EP/R006865/1.

REFERENCES

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

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

Hans-J. Boehm and Dhruva R. Chakrabarti. 2016. Persistence Programming Models for Non-volatile Memory. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management (ISMM 2016)*. ACM, New York, NY, USA, 55–67. https://doi.org/10.1145/2926697.2926704

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

Andreas Chatzistergiou, Marcelo Cintra, and Stratis D. Viglas. 2015. REWIND: Recovery Write-ahead system for In-memory Non-volatile Data-structures. *Proc. VLDB Endow.* 8, 5 (Jan. 2015), 497–508. https://doi.org/10.14778/2735479.2735483

Nathan Chong, Tyler Sorensen, and John Wickerson. 2018. The Semantics of Transactions and Weak Memory in x86, Power, ARM, and C++. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, New York, NY, USA, 211–225. https://doi.org/10.1145/3192366.3192373

Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. *SIGPLAN Not.* 46, 3 (March 2011), 105–118. https://doi.org/10.1145/1961296.1950380

Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O Through Byte-addressable, Persistent Memory. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09)*. ACM, New York, NY, USA, 133–146. https://doi.org/10.1145/1629575.1629589

Harold Cooper. 2008. Persistent Collections. (2008). https://pcollections.org/

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

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

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

Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (July 1990), 463–492. https://doi.org/10.1145/78969.78972

Intel. 2014. Intel architecture instruction set extensions programming reference. (2014). https://software.intel.com/sites/default/files/managed/07/b7/319433-023.pdf

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

Intel. 2019. 3D XPoint. (2019). https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html

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

ITRS. 2011. Process Integration, devices, and structures. (2011). http://www.maltiel-consulting.com/ITRS_2011-Process-Integration-Devices-Structures.pdf International technology roadmap for semiconductors.

Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. 2016a. Failure-Atomic Persistent Memory Updates via JUSTDO Logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, New York, NY, USA, 427–442. https://doi.org/10.1145/2872362.2872410

Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. 2016b. Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model. In *Distributed Computing*, Cyril Gavoille and David Ilcinkas (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 313–327.

Daniel Jackson. 2012. *Software Abstractions – Logic, Language, and Analysis* (revised edition ed.). MIT Press.

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

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

T. Kawahara, K. Ito, R. Takemura, and H. Ohno. 2012. Spin-transfer torque RAM technology: Review and prospect. *Microelectronics Reliability* 52, 4 (2012), 613 – 627. https://doi.org/10.1016/j.microrel.2011.09.028 Advances in non-volatile memory technology.

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

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

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

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

Aasheesh Kolli, Jeff Rosen, Stephan Diestelhorst, Ali Saidi, Steven Pelley, Sihang Liu, Peter M. Chen, and Thomas F. Wenisch. 2016b. Delegated Persist Ordering. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-49)*. IEEE Press, Piscataway, NJ, USA, Article 58, 13 pages. http://dl.acm.org/citation.cfm?id=3195638.3195709

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

Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting Phase Change Memory As a Scalable Dram Alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*. ACM, New York, NY, USA, 2–13. https://doi.org/10.1145/1555754.1555758

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

Daniel Lustig, Andrew Wright, Alexandros Papakonstantinou, and Olivier Giroux. 2017. Automated Synthesis of Comprehensive Memory Model Litmus Test Suites. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. https://doi.org/10.1145/3037697.3037723

Maged M. Michael and Michael L. Scott. 1996. Simple, Fast, and Practical Non-blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing (PODC '96)*. ACM, New York, NY, USA, 267–275. https://doi.org/10.1145/248052.248106

Aleksandar Milicevic, Joseph P. Near, Eunsuk Kang, and Daniel Jackson. 2015. Alloy*: A General-Purpose Higher-Order Relational Constraint Solver. In *International Conference on Software Engineering (ICSE)*.

Faisal Nawab, Joseph Izraelevitz, Terence Kelly, Charles B. Morrey, Dhruva R. Chakrabarti, and Michael James Scott. 2017. Dalí: A Periodically Persistent Hash Map. In *DISC*.

Scott Owens. 2010. Reasoning About the Implementation of Concurrency Abstractions on x86-TSO. In *Proceedings of the 24th European Conference on Object-oriented Programming (ECOOP'10)*. Springer-Verlag, Berlin, Heidelberg, 478–503. http://dl.acm.org/citation.cfm?id=1883978.1884011

PCJ. 2016. Persistent Collections for Java. (2016). https://github.com/pmem/pcj

Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. 2014. Memory Persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA '14)*. IEEE Press, Piscataway, NJ, USA, 265–276. http://dl.acm.org/citation.cfm?id=2665671.2665712

Azalea Raad, Marko Doko, Lovro Rožić, Ori Lahav, and Viktor Vafeiadis. 2019. On Library Correctness Under Weak Memory Consistency: Specifying and Verifying Concurrent Libraries Under Declarative Consistency Models. *Proc. ACM Program. Lang.* 3, POPL, Article 68 (Jan. 2019), 31 pages. https://doi.org/10.1145/3290381

Azalea Raad, Ori Lahav, and Viktor Vafeiadis. 2018. On Parallel Snapshot Isolation and Release/Acquire Consistency. In *Programming Languages and Systems*, Amal Ahmed (Ed.). Springer International Publishing, Cham, 940–967.

Azalea Raad, Ori Lahav, and Viktor Vafeiadis. 2019a. On the Semantics of Snapshot Isolation. In *Verification, Model Checking, and Abstract Interpretation*, Constantin Enea and Ruzica Piskac (Eds.). Springer International Publishing, Cham, 1–23.

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

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

Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. X86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors. *Commun. ACM* 53, 7 (July 2010), 89–97. https://doi.org/10.1145/1785414.1785443

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

SPARC. 1992. *The SPARC Architecture Manual: Version 8.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams. 2008. The missing memristor found. *Nature* 453 (2008), 80 – 83.

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

Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. *SIGPLAN Not.* 47, 4 (March 2011), 91–104. https://doi.org/10.1145/2248487.1950379

Xiaojian Wu and A. L. Narasimha Reddy. 2011. SCMFS: A File System for Storage Class Memory. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC '11).* ACM, New York, NY, USA, Article 39, 11 pages. https://doi.org/10.1145/2063384.2063436

Jishen Zhao, Sheng Li, Doe Hyun Yoon, Yuan Xie, and Norman P. Jouppi. 2013. Kiln: Closing the Performance Gap Between Systems with and Without Persistence Support. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46).* ACM, New York, NY, USA, 421–432. https://doi.org/10.1145/2540708.2540744

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