Persistent Owicki-Gries Reasoning

A Program Logic for Reasoning about Persistent Programs on Intel-x86

AZALEA RAAD, MPI-SWS, Germany and Imperial College London, United Kingdom
ORI LAHAV, Tel Aviv University, Israel
VIKTOR VAFEIADIS, MPI-SWS, Germany

The advent of non-volatile memory (NVM) technologies is expected to transform how software systems are structured fundamentally, making the task of *correct* programming significantly harder. This is because ensuring that memory stores persist in the correct order is challenging, and requires low-level programming to flush the cache at appropriate points. This has in turn resulted in a noticeable *verification gap*.

To address this, we study the verification of NVM programs, and present *Persistent Owicki-Gries* (POG), the first program logic for reasoning about such programs. We prove the soundness of POG over the recent Intel-x86 model, which formalises the out-of-order persistence of memory stores and the semantics of the Intel cache line flush instructions. We then use POG to verify several programs that interact with NVM.

CCS Concepts: • **Theory of computation → Concurrency**; **Semantics and reasoning**; • **Software and its engineering → General programming languages**.

Additional Key Words and Phrases: non-volatile memory, program logic, x86-TSO, consistency, persistency

ACM Reference Format:
Azalea Raad, Ori Lahav, and Viktor Vafeiadis. 2020. Persistent Owicki-Gries Reasoning: A Program Logic for Reasoning about Persistent Programs on Intel-x86. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 151 (November 2020), 28 pages. https://doi.org/10.1145/3428219

1 INTRODUCTION

The emergence of *non-volatile memory* (NVM) technologies [Kawahara et al. 2012; Lee et al. 2009; Strukov et al. 2008] is expected to revamp the structure of modern software: NVM provides storage persistency across power failures with performance close to that of traditional (volatile) memory. As such, programs that require persistency of their data (e.g. databases) can achieve orders of magnitude lower latency by storing their data on NVM rather than on hard disks. It is therefore believed that NVM (a.k.a. persistent memory) will supplant RAM in the near future, thanks to its durable yet competitive performance. This belief is backed by widespread industrial support. Specifically, the two major architectures, ARMv8 and Intel-x86 which together account for almost 100% of the desktop and mobile market, have extended their official specifications to support persistent programming [Arm 2018; Intel 2019]. Intel has further (1) manufactured its own line of NVM, Optane technology [Intel 2019], with an extended academic study evaluating its performance [Izraelevitz et al. 2019]; and (2) released open-source NVM libraries such as PMDK [Intel 2015].

To describe the behaviour of programs under NVM, Intel has introduced a *persistency model* for their x86 architecture [Intel 2019], describing the order in which memory stores may persist to

Authors' addresses: Azalea Raad, MPI-SWS, Saarland Informatics Campus, Germany , Imperial College London, South Kensington Campus, United Kingdom, azalea@imperial.ac.uk; Ori Lahav, Tel Aviv University, School of Computer Science, 69978, Israel, orilahav@tau.ac.il; Viktor Vafeiadis, MPI-SWS, Saarland Informatics Campus, Germany, viktor@mpi-sws.org.

2475-1421/2020/11-ART151
https://doi.org/10.1145/3428219

NVM. This model was formalised by Raad et al. [2020], wherein they extended the x86-TSO model [Sewell et al. 2010] (with thread-local buffers to model the delayed propagation of writes to other threads) with a global persistent buffer to model the out-of-order propagation of stores to NVM.

Although NVM research has grown rapidly in recent years in both persistency semantics [Condit et al. 2009; Gogte et al. 2018; Izraelevitz et al. 2016; Joshi et al. 2015; Kolli et al. 2017, 2016; Raad and Vafeiadis 2018; Raad et al. 2020, 2019] and algorithms/libraries that exploit NVM [Friedman et al. 2018; Nawab et al. 2017; Zuriel et al. 2019], there has been little work on *verifying* such artifacts. To our knowledge, the existing work [Friedman et al. 2018; Nawab et al. 2017; Raad and Vafeiadis 2018; Raad et al. 2020, 2019; Zuriel et al. 2019] offer low-level proofs about the correctness of small persistent algorithms and often make simplifying assumptions. In particular, they all work at the level of *traces* rather than at the level of *program syntax*, while [Friedman et al. 2018; Nawab et al. 2017; Zuriel et al. 2019] further assume sequential consistency as their concurrency model.

This is a significant omission because developing correct persistent data structures is rather error-prone. Since memory stores are typically persisted out of order, one has to use special low-level instructions for flushing the cache in order to ensure a correct persist ordering. Moreover, such persistent implementations are virtually impossible to test and debug, as one would have to use custom hardware to simulate crashes and check correct recovery from them.

To close this gap, we consider the *formal verification* of (multi-threaded) programs running over NVM. To this end, we adapt the well-known Owicki-Gries (OG) proof system [Owicki and Gries 1976], and develop *POG* (Persistent OG), the *first* program logic for reasoning about NVM programs. We show that the POG proof system is *sound* with respect to the Intel-x86 persistency semantics.

We develop POG over Intel-x86 for several reasons. First, Intel-x86 is a ubiquitous architecture with a formally-defined persistency model [Raad et al. 2020]. Indeed, excluding academic models proposed as proofs of concept, the only real-world persistency models currently available are those of low-level architectures, i.e. those of ARMv8 and Intel-x86 [Raad et al. 2020, 2019], and no existing mainstream programming language such as C/C++ currently has a formal persistency model. Second, existing research on language-level persistency models (e.g. [Gogte et al. 2020; Kolli et al. 2017]) suggests that similar persistency primitives to those of Intel-x86 are considered at the language level, and will likely be lifted to higher-levels. As such, the reasoning principles of POG will be useful in the future higher-level persistency models. Lastly, POG presents the first formal framework for reasoning about persistency primitives and their behaviour *abstractly*, i.e. at the program syntax level, rather than delving into all possible program executions and reasoning at the trace level. Given the complexity of the Intel-x86 persistency model, even verifying simple examples is non-trivial, especially when done at the trace level, and we believe that our syntax-level proof rules in POG help simplify such proofs significantly.

Challenges. Developing the reasoning principles of POG over the Intel-x86 persistency model involves two main challenges: (1) dealing with weak memory consistency, i.e. with the thread-local FIFO buffers of Intel-x86; and (2) dealing with weak persistency, i.e. the persistent buffer of Intel-x86, which allows for stores to persist *asynchronously* and *out of order*.

To address the first challenge, we base our program logic on a variant of OG, named OGRA [Lahav and Vafeiadis 2015], proved sound under release-acquire consistency (a memory model weaker than x86-TSO) and thus also sound under the Intel-x86 model as far as consistency is concerned.

To address the second challenge, we develop an *intermediate* operational model of Intel-x86 persistency, Ix86_{sim}, which forgoes the persistent buffer altogether and operates on the original x86-TSO model (i.e. with only the thread-local buffers). We show that our Ix86_{sim} model correctly captures the effect of the Intel-x86's persistency buffer. That is, the possible outcomes of a program under the Intel-x86 persistency model with two types of buffers are the same as those under Ix86_{sim}.

Our next challenge in designing the POG reasoning principles is modelling the behaviour of explicit persist instructions on Intel-x86, **flush** and $\textbf{flush}_{\text{opt}}$, which when executed persist all pending writes on a given cache line to the memory. As we describe shortly in §2, $\textbf{flush}_{\text{opt}}$ instructions offer weaker ordering constraints and may be reordered with respect to other instructions more freely. As such, their effect may not take place at the intended program point, making it more difficult to reason about their persistency behaviour. To keep the POG reasoning principles simple, we devise POG to focus only on the stronger **flush** instructions. We then provide a mechanism to extend our POG reasoning to weaker $\textbf{flush}_{\text{opt}}$ instructions. More concretely, we present a transformation that allows us in most cases to rewrite a program using $\textbf{flush}_{\text{opt}}$ to an *equivalent* program using **flush**. We can then use POG to reason about such programs by first using our transformation to replace $\textbf{flush}_{\text{opt}}$ with **flush**, and then using POG to verify the transformed program.

Although our main contribution is POG, we remark that our Ix86_{sim} model and our transformation are also valuable contributions *per se*. Specifically, Ix86_{sim} may serve as the input to automated verification tools for concurrency, e.g. model checkers, especially those that already support x86-TSO [Abdulla et al. 2015; Clarke et al. 2004; Huang and Huang 2016]. Our persistency-preserving program transformation can be used to optimise code (e.g. at compile time) to replace **flush** with $\textbf{flush}_{\text{opt}}$.

Contributions and Outline. Our contributions (detailed in §2) are as follows: (1) in §3 we present POG, the *first* program logic for verifying persistency guarantees; (2) in §4 we use POG to verify several examples; (3) in §5 we present our Ix86_{sim} model and show that it faithfully captures Intel-x86 persistency; (4) in §6 we show the POG is sound with respect to the Ix86_{sim} model; (5) in §7 we present our transformation for rewriting programs with $\textbf{flush}_{\text{opt}}$ to equivalent ones with **flush**, and show that this transformation is sound. We discuss related and future work in §8.

Additional Material. The proofs of all theorems stated in this article are given in full in the accompanying technical appendix available at http://plv.mpi-sws.org/pog/.

2 OVERVIEW

2.1 Px86_{sim} at a Glance

Memory *consistency* models typically describe the permitted behaviours of programs by constraining the *volatile memory order*, i.e. the order in which memory 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. a power failure) by defining a *persistent memory order*, i.e. the order in which writes 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 a write visible to other threads, whilst the latter denotes the process of committing writes durably to persistent memory.

Raad et al. [2020] recently developed the Px86 ('persistent x86') models, formalising the persistency semantics of the Intel-x86 architecture. As they noted, the Intel manual [Intel 2019] is ambiguous at times and allows for weaker behaviours than originally intended. They thus formulated two persistency models: (1) Px86_{man}, which reflects the behaviour outlined in the manual; and (2) Px86_{sim}, which is a strengthening of Px86_{man} and captures the architectural intent. As Px86_{sim} reflects the architectural intent, in this article we focus on the Px86_{sim} model.

The Px86_{sim} model follows a *buffered*, *relaxed* persistency model. Under a buffered model, memory persists occur *asynchronously* [Condit et al. 2009; Izraelevitz et al. 2016; Joshi et al. 2015]: they 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, while allowing the execution to proceed ahead of persists. As such, after recovering from a crash, only a *prefix*

$x := 1;$ $y := 1;$	$x := 1;$ **flush** $x';$ $y := 1;$	$x := 1;$ **flush**$_{\text{opt}}$ $x';$ $y := 1;$	$x := 1;$ **flush**$_{\text{opt}}$ $x';$ **sfence**; $y := 1;$	$x := 1;$ $\Vert$ $\mathsf{a} := y;$ **flush** $x';$ $\Vert$ **if** $(\mathsf{a}{=}1)$ $y := 1;$ $\Vert$ $z := 1;$
(a)	(b)	(c)	(d)	(e)
$\lightning: x, y \in \{0,1\}$	$\lightning: y{=}1 \Rightarrow x{=}1$	$\lightning: x, y \in \{0,1\}$	$\lightning: y{=}1 \Rightarrow x{=}1$	$\lightning: z{=}1 \Rightarrow x{=}1$

Fig. 1. Example Px86$_{\text{sim}}$ programs and possible values 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 $x{=}y{=}z{=}0$. Replacing the **sfence** instruction in (d) with **mfence** or an atomic RMW yields the same result. Similarly, replacing **flush** x' in (e) with **flush**$_{\text{opt}}$ x'; c yields the same result when c is an **sfence**/**mfence** or an RMW.

of the persistent memory order may have successfully persisted. Under relaxed persistency, the volatile and persistent memory orders may disagree: the order in which the writes are made visible to other threads may differ from the order in which they are persisted.

The relaxed and buffered persistency of Px86$_{\text{sim}}$ is demonstrated in Fig. 1a. If a crash occurs during (or after) the execution of this program, at crash time either write may or may not have persisted and thus $x, y \in \{0,1\}$ upon recovery. The relaxed nature of Px86$_{\text{sim}}$ allows for surprising behaviours that are not possible during normal (non-crashing) executions. Specifically, the two writes cannot be reordered under Intel-x86 and thus at no point during the normal execution of this program $x{=}0$, $y{=}1$ is observable. Nevertheless, in case of a crash it is possible under Px86$_{\text{sim}}$ to observe $x{=}0$, $y{=}1$ after recovery. This is due to the relaxed persistency of Px86$_{\text{sim}}$: the store order (x before y) is separate from the persist order (y before x). Under the Px86$_{\text{sim}}$ model the writes may persist (1) in any order, when they are on distinct locations; or (2) in the volatile memory order, when they are on the same location. That is, for each location, its store and persist orders coincide.

Intel-x86 provides explicit *persist* instructions, **flush** x, **flush**$_{\text{opt}}$ x and **wb** x, in order to afford more control over when pending writes are persisted. When executed, these instructions *asynchronously* persist all pending writes on all locations in the cache line of x [Intel 2019]. 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 $x' \in X$. As noted by Raad et al. [2020], **flush** instructions are the strongest of the three in terms of their constraints on instruction reordering, whereas **flush**$_{\text{opt}}$ and **wb** are equally weak and have the same specification, with **wb** providing better performance than **flush**$_{\text{opt}}$. That is, **flush**$_{\text{opt}}$ and **wb** are indistinguishable under Px86$_{\text{sim}}$. As such, in the remainder of our discussion we focus on **flush** and **flush**$_{\text{opt}}$ instructions and describe their behaviour via several examples.

The persistency behaviour of **flush** is illustrated in Fig. 1b: given $x, x' \in X$, executing **flush** x' persists the earlier write on X (i.e. $x := 1$). As such, if a crash occurs during the execution of this program and $y{=}1$ upon recovery, then $x{=}1$. That is, if $y := 1$ has executed and persisted before the crash, then so must the earlier $x := 1$; **flush** x'. This is guaranteed by the ordering constraints on **flush**: **flush** instructions are ordered with respect to both earlier (in program order) and later writes. Hence, **flush** x' in Fig. 1b cannot be reordered with respect to $x := 1$ or $y := 1$. As such, upon recovery $y{=}1 \Rightarrow x{=}1$. Note that **flush** x persists X *asynchronously*: its execution does not block until X is persisted; rather, the execution proceeds and X is made persistent at a future point.

In contrast to **flush**, **flush**$_{\text{opt}}$ instructions provide weaker ordering guarantees in relation to writes, in that they are only ordered with respect to *earlier writes* on the *same* cache line. This is illustrated in the example of Fig. 1c obtained from that in Fig. 1b by replacing **flush** x' with **flush**$_{\text{opt}}$ x'. Unlike in Fig. 1b, the **flush**$_{\text{opt}}$ x' instruction is not ordered with respect to the later write ($y := 1$) and may thus be reordered after it. As such, **flush**$_{\text{opt}}$ x' may not execute at the intended program point (after $x := 1$ and before $y := 1$) and thus may not guarantee the intended persist

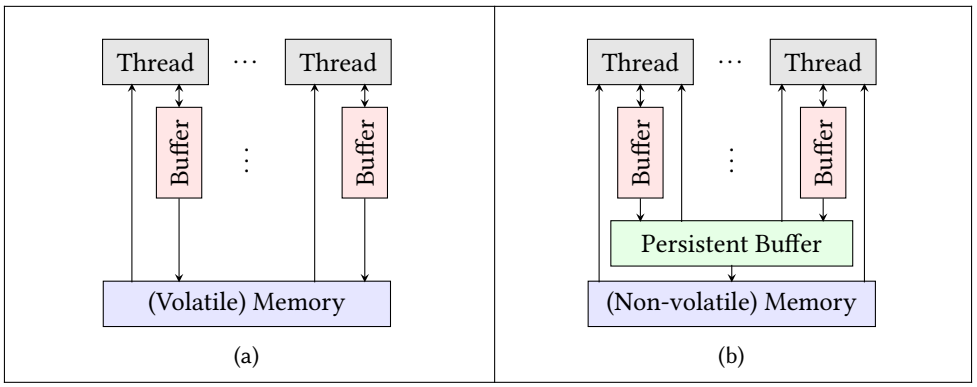

Fig. 2. Storage subsystems of x86-TSO (a) and Px86_{sim} (b) as depicted in [Raad et al. 2020]

ordering. That is, unlike in Fig. 1b, the $\textbf{flush}_{\text{opt}}\ x'$ instructions does not guarantee that the $x := 1$ write persists before $y := 1$, and is thus possible to observe $x{=}0 \wedge y{=}1$ upon recovery.

In order to prevent such reorderings and to strengthen the ordering constraints between $\textbf{flush}_{\text{opt}}$ and later instructions, one can use either *fence* instructions, namely **sfence** (store fence) and **mfence** (memory fence), or atomic *read-modify-write* (RMW) instructions such as compare-and-set (**CAS**) and fetch-and-add (**FAA**). More concretely, **sfence**, **mfence** and RMW instructions are ordered with respect to all (both earlier and later) $\textbf{flush}_{\text{opt}}$, **flush** and write instructions, and can be used to prevent reorderings such as that in Fig. 1c. This is illustrated in the example of Fig. 1d obtained from Fig. 1c by inserting an **sfence** after $\textbf{flush}_{\text{opt}}$. Unlike in Fig. 1c, the intervening **sfence** ensures that $\textbf{flush}_{\text{opt}}$ in Fig. 1d is ordered with respect to $y := 1$ and cannot be reordered after it, thus ensuring that $x := 1$ persists before $y := 1$ (i.e. $y{=}1 \Rightarrow x{=}1$ upon recovery), as in Fig. 1b.

The example in Fig. 1e illustrates how persist orderings can be imposed on the writes of different threads using *message passing*. Note that the program in the left thread of Fig. 1e is that of Fig. 1b. 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. 1e reads 1 from y (written by the left thread), then the left thread passes a message to the right thread. Under Intel-x86 message passing ensures that the instruction writing the message and all those ordered before it (e.g. $x := 1; \textbf{flush}\ x; y := 1$) are executed (ordered) before the instruction reading it (e.g. $\mathsf{a} := y$). As such, since $x := 1; \textbf{flush}\ x'$ is executed before $\mathsf{a} := y$, and $z := 1$ is executed after $\mathsf{a} := y$ when $\mathsf{a}{=}1$, we know $x := 1; \textbf{flush}\ x'$ is executed before $z := 1$. Consequently, if upon recovery $z{=}1$ (i.e. $z := 1$ has persisted before the crash), then $x{=}1$ ($x := 1; \textbf{flush}\ x'$ must have also persisted before the crash). As before, replacing **flush** x' in Fig. 1e with $\textbf{flush}_{\text{opt}}\ x'; \mathsf{c}$ yields the same result upon recovery when c is an **sfence**/**mfence** or an RMW.

Lastly, observe that $\textbf{flush}/\textbf{flush}_{\text{opt}}$ instructions impose a particular *persist ordering*. Given $x \in X$, all writes on X ordered before **flush** x/$\textbf{flush}_{\text{opt}}\ x$ persist before all instructions (regardless of their cache line) ordered after **flush** x. For instance, since $x := 1$ in Fig. 1b is ordered before **flush** x', and **flush** x' is ordered before $y := 1$, the $x := 1$ write is guaranteed to persist before $y := 1$. Similarly, as $x := 1$ in Fig. 1d is ordered before $\textbf{flush}_{\text{opt}}\ x'$ which is in turn ordered before $y := 1$ (thanks to the intervening **sfence**), the $x := 1$ write is guaranteed to persist before $y := 1$.

The Operational Px86_{sim} Model. Raad et al. [2020] developed their operational Px86_{sim} model as an extension of the x86-TSO model by Sewell et al. [2010]. As illustrated in Fig. 2a, each thread in x86-TSO is connected to the (volatile) memory via a FIFO buffer. When a thread writes value v to location x, it records it in its buffer as the $\langle x, v \rangle$ entry. When a thread reads from x, it first consults

its own buffer. If it contains buffered writes for x, the thread reads the last such buffered write; otherwise, it consults the memory. Threads can debuffer their writes by propagating them (in FIFO order) to the memory at non-deterministic times. Additionally, the execution of a memory fence **mfence** drains the buffer of the executing thread.

To model the buffered persists of Px86_{sim}, Raad et al. [2020] extended the x86-TSO storage subsystem with a *persistent buffer* as depicted in Fig. 2b, containing 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 writes in the thread-local buffer are debuffered, they are propagated to the persistent buffer, denoting the store of the write (i.e. when the write becomes visible to other threads). Pending writes in the persistent buffer are in turn debuffered and propagated to the memory at non-deterministic times, denoting the persist of the write (i.e. when the write is committed durably to memory). The execution of reads accordingly traverses this hierarchy: when reading from x, the thread first inspects its own local buffer for the last write to x when such a write exists; otherwise, it consults the persistent buffer for the last store to x if such a store exists; otherwise, it reads x from the memory. Recall that the writes on distinct locations may persist in any order, whereas the writes on the same location persist in the store order. To capture this, the persistent buffer is modelled as a queue, where the pending writes on each location are propagated in the FIFO queue order, while those on different locations are propagated in an arbitrary order.

2.2 Eliminating $\text{flush}_{\text{opt}}$ Instructions via Program Transformation

As discussed above, the $\textbf{flush}_{\text{opt}}$ instructions provide weaker ordering constraints and can be reordered e.g. with respect to writes on different cache lines. While this flexibility may in certain cases lead to better performance by affording the compiler more optimisation opportunities through reordering, it significantly complicates the task of reasoning about persistency behaviours. In particular, the weak ordering constraints on $\textbf{flush}_{\text{opt}}$ can lead to unintended persistency behaviours such as that in Fig. 1c (where it is possible to observe $x{=}0 \wedge y{=}1$ upon recovery), and to ensure correct persistency one must thus account for all possible such reorderings. It is therefore simpler to limit our reasoning to programs that solely use the stronger **flush** instructions.

As such, in order to support reasoning about the weaker $\textbf{flush}_{\text{opt}}$ instructions while simultaneously keeping our reasoning principles simple, we first (1) devise a mechanism that in most cases allows us to transform programs using $\textbf{flush}_{\text{opt}}$ to *equivalent* programs that *only* use **flush**; and then (2) design our POG reasoning principles for programs that only use **flush** instructions.

More concretely, for step (1) we note that the main use-case of $\textbf{flush}_{\text{opt}}$ (in which using $\textbf{flush}_{\text{opt}}$ rather than **flush** may prove advantageous for performance) prescribes a particular programming *pattern*. We then show that given a program C that uses $\textbf{flush}_{\text{opt}}$ in this pattern, one can transform C to a program C′ that uses *only* **flush** instructions, such that C and C′ have *equivalent persistency behaviours*, in that they yield the same values for all memory locations upon crash recovery.

Note that our intent through this transformation is not to forgo $\textbf{flush}_{\text{opt}}$ instructions altogether; rather, this transformation merely allows us to extend our reasoning to programs that use $\textbf{flush}_{\text{opt}}$ by considering equivalent programs using **flush**, while keeping our reasoning principles simple.

We present the formal details of this transformation in §7; in the remainder of this section and in §3-§6 we thus focus on programs using only **flush** (and not $\textbf{flush}_{\text{opt}}$) instructions.

2.3 An Intermediate Operational Model for Px86_{sim}

Our goal in this paper is to devise a program logic for reasoning about the persistency behaviour of programs under Px86_{sim}. Although program logics are typically built over operational models that manipulate the underlying state, such operational models often operate on states that comprise the

memory alone, without intermediate caches such as those of thread-local and persistent buffers in Px86_{sim}. This is because a large number of such logics operate under *sequential consistency* (SC) [Lamport 1979], while the presence of such caches introduces weak behaviours absent under SC. To remedy this, recent research [Lahav and Vafeiadis 2015; Sieczkowski et al. 2015; Svendsen et al. 2018; Turon et al. 2014; Vafeiadis and Narayan 2013] demonstrates how to reason about the weak behaviours introduced by e.g. thread-local buffers (see §2.4). However, no existing work currently supports the challenging task of reasoning about the persistency behaviour of programs. The difficulty of such reasoning is further compounded when considering the buffered behaviour of persists due to e.g. the persistent buffer of Px86_{sim}.

To streamline the task of devising a program logic for Px86_{sim}, we first (1) develop an *intermediate* operational semantics, Ix86_{sim}, that forgoes the persistent buffer, while emulating all valid Px86_{sim} behaviours; and then (2) devise a program logic for persistency reasoning over Ix86_{sim}. We proceed with an intuitive account of our Ix86_{sim} model; we briefly describe our program logic later in §2.4.

As discussed above, in our Ix86_{sim} model we forgo the persistent buffer altogether, thus operating on the x86-TSO storage system in Fig. 2a, with the volatile memory replaced with a non-volatile one. For simplicity, let us begin by assuming that **flush** instructions are executed *synchronously*. We later lift this assumption and describe how we handle the asynchronous behaviour of **flush**.

Recall that under Px86_{sim} the store and persist orders may disagree; i.e. the order in which writes in thread buffers are debuffered may differ from the order in which they are debuffered from the persistent buffer. As such, when forgoing the persistent buffer, additional care is required to preserve such weak behaviours. To see this, let us return to Fig. 1a, where $x := 1$ is always store-ordered before $y := 1$, while $y := 1$ may be persist-ordered before $x := 1$. We can then model the store order as in x86-TSO: upon executing each write the thread adds it to its local buffer, and non-deterministically debuffers its entries in the FIFO order, thus ensuring $x := 1$ is store-ordered before $y := 1$. However, without the additional persistent buffer, we can no longer model the out-of-order persists.

To remedy this, for each location x we record two versions: (1) the '*volatile*' version, written x_{v}, tracking the latest observable value of x; and (2) the '*synchronously-persisted*' ('synchronous') version, written x_{s}, tracking the latest persisted value of x provided that **flush** instructions are executed synchronously. Memory instructions (e.g. writes) are then carried out on volatile versions, leaving the synchronous versions untouched. Moreover, the volatile versions may non-deterministically propagate to the corresponding synchronous versions, modelling the notion that writes may be committed to memory at non-deterministic times. Similarly, since we assume **flush** instructions execute synchronously, given $x \in X$, executing **flush** x copies x'_{v} to x'_{s}, for all $x' \in X$.

Let us return to Fig. 1a and write $x{=}v$ to denote that the latest value observable for x is v; i.e. either the thread buffer contains no x entries and the value of x in memory is v, or the latest x entry in the thread buffer is $\langle x, v\rangle$. Similarly, let us write $x \in \{v_1, v_2\}$ for $x{=}v_1 \vee x{=}v_2$. Therefore:

(i) we assume that initially $x_{\mathsf{v}}{=}x_{\mathsf{s}}{=}y_{\mathsf{v}}{=}y_{\mathsf{s}}{=}0$;
(ii) after executing $x := 1$ we have: $x_{\mathsf{v}}{=}1 \wedge x_{\mathsf{s}} \in \{0,1\} \wedge y_{\mathsf{v}}{=}y_{\mathsf{s}}{=}0$;
(iii) upon subsequently executing $y := 1$ we have: $x_{\mathsf{v}}{=}1 \wedge x_{\mathsf{s}} \in \{0,1\} \wedge y_{\mathsf{v}}{=}1 \wedge y_{\mathsf{s}} \in \{0,1\}$.

Note that since initially $x_{\mathsf{s}}{=}0$ and the value of x_{v} may be copied to x_{s} non-deterministically, we must account for this propagation in (ii) and thus we have $x_{\mathsf{s}} \in \{0,1\}$; similarly for y_{s} in (iii). As such, at all program points ((i)–(iii)) we have $x_{\mathsf{s}}, y_{\mathsf{s}} \in \{0,1\}$. That is, if a crash occurs at any point, upon recovery we have $x_{\mathsf{s}}, y_{\mathsf{s}} \in \{0,1\}$, thus emulating the desired behaviour in Fig. 1a.

We can analogously emulate the behaviour of Fig. 1b:

(a) we assume that initially $x_{\mathsf{v}}{=}x_{\mathsf{s}}{=}y_{\mathsf{v}}{=}y_{\mathsf{s}}{=}0$;
(b) after executing $x := 1$ we have: $x_{\mathsf{v}}{=}1 \wedge x_{\mathsf{s}} \in \{0,1\} \wedge y_{\mathsf{v}}{=}y_{\mathsf{s}}{=}0$;

(c) after executing **flush** x we have: $x_{\mathsf{v}}{=}x_{\mathsf{s}}{=}1 \wedge y_{\mathsf{v}}{=}y_{\mathsf{s}}{=}0$;
(d) upon subsequently executing $y := 1$ we have: $x_{\mathsf{v}}{=}x_{\mathsf{s}}{=}1 \wedge y_{\mathsf{v}}{=}1 \wedge y_{\mathsf{s}} \in \{0,1\}$.

That is, executing **flush** x (c) copies x_{v} to x_{s}, and thus we have $y_{\mathsf{s}}{=}1 \Rightarrow x_{\mathsf{s}}{=}1$ at all program points.

***Modelling the Asynchronous Behaviour of* flush.** As we demonstrated above, tracking the synchronous version of locations (e.g. x_{s}) allows us to capture the necessary persist orderings (e.g. that x_{s} persists before y_{s}). However, as **flush** instructions execute asynchronously, synchronous versions do not accurately capture the memory state upon a crash. For instance, if a crash occurs after **flush** x' is executed (but not yet fully completed) in Fig. 1b, unlike what we wrote in (c) and (d) above, x may not necessarily contain 1. To address this, for each location x we record a third, *persisted* version, written x_{p}, denoting the latest persisted value of x (without assuming **flush** instructions are executed synchronously).

Intuitively, there is a chronological order on different versions of a location x: $x_{\mathsf{p}} \rightarrow x_{\mathsf{s}} \rightarrow x_{\mathsf{v}}$, in that while x_{p} reflects the last persisted value of x in memory, x_{s} and x_{v} denote later updates on x that are yet to be persisted, with x_{v} describing the latest such update. As discussed, x_{v} may non-deterministically be copied to x_{s}, allowing x_{s} to catch up with x_{v}. Intuitively, this amounts to a pending write on x (in the persistent buffer) being committed to memory. Analogously, the effect of asynchronous **flush** instructions may take effect at non-deterministic times. We may thus be inclined to copy x_{s} to x_{p} non-deterministically, allowing x_{p} to catch up with x_{s}. However, this is too weak. To see this, let us extend the (a)–(d) steps of Fig. 1b with x_{p} and y_{p} constraints (highlighted):

(a') $x_{\mathsf{v}}{=}x_{\mathsf{s}}{=}x_{\mathsf{p}}{=}y_{\mathsf{v}}{=}y_{\mathsf{s}}{=}y_{\mathsf{p}}{=}0$
(b') $x_{\mathsf{v}}{=}1 \wedge x_{\mathsf{s}}, x_{\mathsf{p}} \in \{0,1\} \wedge (x_{\mathsf{p}}{=}1 \Rightarrow x_{\mathsf{s}}{=}1) \wedge y_{\mathsf{v}}{=}y_{\mathsf{s}}{=}y_{\mathsf{p}}{=}0$
(c') $x_{\mathsf{v}}{=}x_{\mathsf{s}}{=}1 \wedge x_{\mathsf{p}} \in \{0,1\} \wedge y_{\mathsf{v}}{=}y_{\mathsf{s}}{=}y_{\mathsf{p}}{=}0$
(d') $x_{\mathsf{v}}{=}x_{\mathsf{s}}{=}1 \wedge y_{\mathsf{v}}{=}1 \wedge x_{\mathsf{p}}, y_{\mathsf{s}}, y_{\mathsf{p}} \in \{0,1\} \wedge (y_{\mathsf{p}}{=}1 \Rightarrow y_{\mathsf{s}}{=}1)$

First, note that $(x_{\mathsf{p}}{=}1 \Rightarrow x_{\mathsf{s}}{=}1)$ in (b') captures the chronological order between x_{s} and x_{p}: x_{s} may be copied to x_{p} and thus if $x_{\mathsf{p}}{=}1$ then $x_{\mathsf{s}}{=}1$; similarly for $(y_{\mathsf{p}}{=}1 \Rightarrow y_{\mathsf{s}}{=}1)$ in (d'). Second, note that x_{p} and y_{p} are our main interest as they denote the latest persisted values of x and y. We are hence interested in establishing $y_{\mathsf{p}}{=}1 \Rightarrow x_{\mathsf{p}}{=}1$ at all program points, thus modelling the desired behaviour in Fig. 1b. This is, however, not the case as (d') allows $y_{\mathsf{p}}{=}1 \wedge x_{\mathsf{p}}{=}0$.

To remedy this, we require that the non-deterministic copying of synchronous values to persisted ones be carried out *for all locations* at once, and not a single location. In doing so, we ensure that $y_{\mathsf{p}}{=}1 \Rightarrow x_{\mathsf{p}}{=}1$ holds at (d'), as desired. Intuitively, this captures the global persist orderings imposed by **flush**. In particular, recall that when $x \in X$, all X writes ordered before **flush** x persist before all instructions ordered after **flush** x. As such, upon propagating a write to the persistent memory, i.e. copying some y_{s} to y_{p}, we must ensure that the effects of each prior **flush** x is completed in that its preceding writes on X have also reached the memory. This amounts to (simultaneous) copying of x_{s} to x_{p} for $x \in X$ (since each prior **flush** on X has copied x_{v} to x_{s} for $x \in X$).

In summary, in our Ix86$_{\text{sim}}$ model: (1) memory operations on x (reads/writes) manipulate x_{v}; (2) when $x \in X$, **flush** x copies x'_{v} to x'_{s} for every $x' \in X$; (3) x_{v} may be copied to x_{s} non-deterministically; and (4) $\overline{x_{\mathsf{s}}}$ may be point-wise copied to $\overline{x_{\mathsf{p}}}$ non-deterministically, where $\overline{x_{\mathsf{s}}}$ denotes *all* synchronous locations and $\overline{x_{\mathsf{p}}}$ denotes the corresponding persistent ones.

2.4 POG: Persistent Owicki-Gries Reasoning

Having forgone the need for persistent buffers through our Ix86$_{\text{sim}}$ operational model, our next goal is to devise a program logic for persistency reasoning over Ix86$_{\text{sim}}$ which operates on the x86-TSO storage system in Fig. 2a. As mentioned in §2.3, our next challenge is accounting for the weak behaviours caused by the thread-local buffers in x86-TSO. Fortunately, existing work [Lahav and Vafeiadis 2015; Sieczkowski et al. 2015; Svendsen et al. 2018; Turon et al. 2014; Vafeiadis and

Narayan 2013] demonstrate how existing program logics for SC can be adapted to reason about such weak behaviours. Here, we follow the simple approach of [Lahav and Vafeiadis 2015], which demonstrated how Owicki-Gries (OG) reasoning [Owicki and Gries 1976] can be adapted to reason about the release-acquire fragment of the C11 memory model [Lahav et al. 2017].

OG Reasoning. As in Hoare logic, the basic constructs in OG are *Hoare triples* of the form $\{P\}\ \mathsf{C}\ \{Q\}$, where P and Q are assertions (sets of states) describing the pre- and post-condition of program C. OG reasoning extends the proof rules of Hoare logic with a rule to reason about concurrent programs of the form $\mathsf{C}_1 \,||\, \mathsf{C}_2$, which allows one to compose the verified programs C_1 and C_2 into a verified concurrent program, provided that the two proofs are *non-interfering*:

$$\frac{\{P_1\}\ \mathsf{C}_1\ \{Q_1\} \quad \{P_2\}\ \mathsf{C}_2\ \{Q_2\} \quad \text{two proofs are non-interfering}}{\{P_1 \wedge P_2\}\ \mathsf{C}_1 \,||\, \mathsf{C}_2\ \{Q_1 \wedge Q_2\}}$$

As such, OG is often deemed non-compositional as it refers to non-interference of proof outlines that cannot be checked based solely on the two input triples. However, as demonstrated in [Lahav and Vafeiadis 2015], presenting OG in the *rely-guarantee* (RG) [Jones 1983] style allows compositional reasoning. In this presentation, Hoare triples are interpreted under an *RG context*, $\langle \mathcal{R}; \mathcal{G} \rangle$. The *rely* component, $\mathcal{R}$, comprises a set of assertions assumed to be *stable* under memory updates carried out by the environment (i.e. other threads). The *guarantee* component, $\mathcal{G}$, in turn comprises a set of *guarded updates* that the thread may perform. A guarded update is of the form $\langle x, \mathsf{e}, P \rangle$, stating that when the program state satisfies the guard P, then the thread may update x to e.

An RG-style OG triple is of the form $\langle \mathcal{R}; \mathcal{G} \rangle \vdash \{P\}\ \mathsf{C}\ \{Q\}$, stating that: (1) every terminating run of C from a state in P results in a state in Q; (2) C updates the state in accordance with $\mathcal{G}$ while satisfying the prescribed guards; and (3) the same holds when C is run in parallel with any program C' whose updates preserve the assertions in $\mathcal{R}$. We can then rewrite the parallel composition rule in the RG-style as shown in Fig. 3 (PAR), requiring that the RG contexts of the proofs be non-interfering. That is, every update of C_1 in $\mathcal{G}_1$ preserves every assertion in $\mathcal{R}_2$, and vice versa.

Invariant-based Reasoning. A main advantage of Hoare logic and its descendants such as OG is their support for *compositional* reasoning: once we verify the behaviour of a small program C, we can use its specification to verify larger programs that use C. For instance, having established $\{P\}\ \mathsf{C}\ \{Q\}$, we can then use it to verify the larger program $\mathsf{C}' \triangleq \mathsf{C}; \mathsf{C}''$. That is, it suffices to find Q' such that $\{Q\}\ \mathsf{C}''\ \{Q'\}$, as we can then compose the two triples to derive $\{P\}\ \mathsf{C}'\ \{Q'\}$. In other words, this allows us to treat C as a black box, jumping over its body and assuming Q at the end.

However, this is no longer the case in the persistent setting: as the execution of C may crash, we cannot assume that its execution completed successfully and that Q describes the state on its completion. Rather, we must account for the possibility of C crashing at any point during its execution. As such, the state upon returning from C (either due to a crash or after successful completion) is that described by the *disjunction* (union) of states at each program point. To keep our presentation simple, we typically define an *invariant*, I, that holds at all program points, and could simply be defined as the disjunction of all states. Intuitively, I corresponds to the persistency behaviour we seek to establish, e.g. $y{=}1 \Rightarrow x{=}1$ in Fig. 1b. At each program point we can still strengthen I by adding additional conjuncts. However, when C is used in the larger context of C', we can simply treat its specification as $\{I\}\ \mathsf{C}\ \{I\}$. Note that this is a generalisation of the approach in [Chen et al. 2015], where Hoare triples are of the form $\{P\}\ \mathsf{C}\ \{Q\}\{I\}$, with Q denoting the non-crashing postcondition that holds once C executes successfully, and I denoting the crashing postcondition that holds in case of a crash, and is itself typically defined as a disjunction of postconditions at each point. As such, the overall postcondition of C is described by $Q \vee I$.

Stability. Recall that under Ix86_{sim}, x_{v} may be non-determi-nistically copied to x_{s} for each location, while x_{s} may in turn be non-deterministically copied to x_{p} for all locations at once. As such, we stipulate that the assertions used in our proofs be *stable* with respect to these propagations. In particular, we require that for all x: $P \Rightarrow P[x_{\mathsf{v}}/x_{\mathsf{s}}]$; i.e. if P holds beforehand, it should still hold after x_{v} is propagated to x_{s}. Analogously, we require that for all x: $P \Rightarrow P[\overline{x_{\mathsf{s}}/x_{\mathsf{p}}}]$.

POG Reasoning. We develop the POG ('persistent Owicki-Gries') logic for reasoning about persistency behaviours of programs under Px86_{sim}. We formulate POG as an extension of OG, and build it over our Ix86_{sim} model. We present the formal details of POG in §3. Here we introduce POG by verifying the example in Fig. 1b. Later in §4 we verify several other examples in POG.

Recall that our goal is to devise an invariant I that holds at all program points in Fig. 1b. In particular, we are interested in establishing $I \triangleq y_{\mathsf{p}}{=}1 \Rightarrow x_{\mathsf{p}}{=}1$, as shown in Fig. 1b. As discussed, memory operations (e.g. writes) on x in Ix86_{sim} manipulate the x_{v} location. This is reflected in the (WRITE) rule in Fig. 3, stating that when executing $x := \mathsf{e}$, the postcondition Q is obtained from the precondition P by substituting the new value e for x_{v}. Analogously, the (FLUSH) proof rule in Fig. 3 states that when executing **flush** x, the postcondition Q is obtained from the precondition P by copying x'_{v} to x'_{s} for every x' in the cache line of x. As such, assuming that initially all locations hold 0 and that x and x' are in the same cache line, we can verify the program in Fig. 1b as follows:

$$\begin{gathered}
\{I \wedge x_{\mathsf{v}}{=}x_{\mathsf{s}}{=}x_{\mathsf{p}}{=}0 \wedge y_{\mathsf{v}}{=}y_{\mathsf{s}}{=}y_{\mathsf{p}}{=}0\} \\
x := 1; \\
\{I \wedge x_{\mathsf{v}}{=}1 \wedge x_{\mathsf{s}}, x_{\mathsf{p}} \in \{0,1\} \wedge y_{\mathsf{v}}{=}y_{\mathsf{s}}{=}y_{\mathsf{p}}{=}0\} \\
\textbf{flush}\ x'; \\
\{I \wedge x_{\mathsf{v}}{=}x_{\mathsf{s}}{=}1 \wedge x_{\mathsf{p}} \in \{0,1\} \wedge y_{\mathsf{v}}{=}y_{\mathsf{s}}{=}y_{\mathsf{p}}{=}0\} \\
y := 1; \\
\{I \wedge x_{\mathsf{v}}{=}x_{\mathsf{s}}{=}1 \wedge y_{\mathsf{v}}{=}1 \wedge x_{\mathsf{p}}, y_{\mathsf{s}}, y_{\mathsf{p}} \in \{0,1\}\}
\end{gathered}$$

Note that as the program in Fig. 1b is sequential, we define the RG context as $\langle \top; \top \rangle$; i.e. the environment must preserve all assertions ($\mathcal{R}{=}\top$), and the program may perform any assignment ($\mathcal{G}{=}\top$). For simplicity, we have elided the RG context above. Observe that although $x := 1$ solely manipulates x_{v} and in its precondition we have $x_{\mathsf{s}}{=}x_{\mathsf{p}}{=}0$, after its execution we weakened the postcondition to $x_{\mathsf{s}}, x_{\mathsf{p}} \in \{0,1\}$. This is to ensure that the postcondition is *stable* since the value of x_{v} (i.e. 1) may non-deterministically propagate to x_{s} (and subsequently from x_{s} to x_{p}), as discussed above. Moreover, when analogously stabilising the post-condition of $y := 1$, we must propagate both x_{s} to x_{p} and y_{s} to y_{p} *simultaneously*, thus ruling out $x_{\mathsf{p}}{=}0$, $y_{\mathsf{p}}{=}1$ (as $x_{\mathsf{s}}{=}1$) and establishing I.

3 THE POG PROGRAM LOGIC

POG Language. The POG language given below is that of Px86_{sim} in [Raad et al. 2020] excluding $\textbf{flush}_{\text{opt}}$. We assume a finite set LOC of memory locations; a finite set REG of (local) registers; a finite set VAL of values; a finite set TID of thread identifiers; any standard interpreted language for expressions containing registers and values; and a finite set CL of cache lines partitioning locations ($\text{LOC}{=} \uplus \text{CL}$). We use x, y, z as metavariables for locations; $\mathsf{a}, \mathsf{b}, \mathsf{c}$ for registers; v for values; τ for thread identifiers; e for expressions; and X for cache lines.

$$\begin{aligned}
\text{SCOM} \ni \mathsf{c} ::= {} & \textbf{skip} \mid \textbf{while}(\mathsf{e})\ \mathsf{c} \mid \textbf{if}\ (\mathsf{e})\ \textbf{then}\ \mathsf{c}_1\ \textbf{else}\ \mathsf{c}_2 \mid \mathsf{c}_1; \mathsf{c}_2 \mid \mathsf{a} := \mathsf{e} \mid \mathsf{a} := x \mid x := \mathsf{e} \\
& \mid \textbf{flush}\ x \mid \textbf{sfence} \mid \textbf{mfence} \mid \mathsf{a} := \textbf{CAS}(x, \mathsf{e}_1, \mathsf{e}_2) \mid \mathsf{a} := \textbf{FAA}(x, \mathsf{e}) \\
\text{COM} \ni \mathsf{C} \triangleq {} & \text{TID} \xrightarrow{\text{fin}} \text{SCOM}
\end{aligned}$$

$$\frac{}{\langle\{P\};\emptyset\rangle \vdash \{P\}\ \mathbf{skip}\ \{P\}}\ (\textsc{skip}) \qquad \frac{P \Rightarrow Q[\mathsf{e}/\mathsf{a}]}{\langle\{P,Q\},\emptyset\rangle \vdash \{P\}\ \mathsf{a}:=\mathsf{e}\ \{Q\}}\ (\textsc{assign}) \qquad \frac{P \Rightarrow Q[x_{\mathsf{v}}/\mathsf{a}]}{\langle\{P,Q\},\emptyset\rangle \vdash \{P\}\ \mathsf{a}:=x\ \{Q\}}\ (\textsc{read})$$

$$\frac{P \Rightarrow Q[\mathsf{e}/x_{\mathsf{v}}]}{\langle\{P,Q\},\{(x_{\mathsf{v}},\mathsf{e},P)\}\rangle \vdash \{P\}\ x:=\mathsf{e}\ \{Q\}}\ (\textsc{write}) \qquad \frac{P \Rightarrow Q[x_{\mathsf{v}}/\mathsf{a}][x_{\mathsf{v}}+\mathsf{e}/x_{\mathsf{v}}]}{\langle\{P,Q\},\{(x_{\mathsf{v}},x_{\mathsf{v}}+\mathsf{e},P)\}\rangle \vdash \{P\}\ \mathsf{a}:=\mathbf{FAA}(x,\mathsf{e})\ \{Q\}}\ (\textsc{faa})$$

$$\frac{\begin{array}{c}P \wedge x_{\mathsf{v}}{\neq}\mathsf{e}_1 \Rightarrow Q[x_{\mathsf{v}}/\mathsf{a}] \\ P \wedge x_{\mathsf{v}}{=}\mathsf{e}_1 \Rightarrow Q[\mathsf{e}_1/\mathsf{a}][\mathsf{e}_2/x_{\mathsf{v}}]\end{array}}{\langle\{P,Q\},\{(x_{\mathsf{v}},\mathsf{e}_2,P\wedge x_{\mathsf{v}}{=}\mathsf{e}_1)\}\rangle \vdash \{P\}\ \mathsf{a}:=\mathbf{CAS}(x,\mathsf{e}_1,\mathsf{e}_2)\ \{Q\}}\ (\textsc{cas}) \qquad \frac{\begin{array}{c}x\in X \qquad X=\{x^1\ldots x^n\} \\ P \Rightarrow Q[x_{\mathsf{v}}^1/x_{\mathsf{s}}^1\ldots x_{\mathsf{v}}^n/x_{\mathsf{s}}^n]\end{array}}{\langle\{P,Q\},\emptyset\rangle \vdash \{P\}\mathbf{flush}\ x\{Q\}}\ (\textsc{flush})$$

$$\frac{\begin{array}{c}\langle\mathcal{R}_1;\mathcal{G}_1\rangle \vdash \{P\}\ \mathsf{c}_1\ \{R\} \\ \langle\mathcal{R}_2;\mathcal{G}_2\rangle \vdash \{R\}\ \mathsf{c}_2\ \{Q\}\end{array}}{\langle\mathcal{R}_1\cup\mathcal{R}_2;\mathcal{G}_1\cup\mathcal{G}_2\rangle \vdash \{P\}\ \mathsf{c}_1;\mathsf{c}_2\ \{Q\}}\ (\textsc{seq}) \qquad \frac{\begin{array}{c}\langle\mathcal{R};\mathcal{G}\rangle \vdash \{P\wedge \mathsf{e}{\neq}0\}\ \mathsf{c}_1\ \{Q\} \\ \langle\mathcal{R};\mathcal{G}\rangle \vdash \{P\wedge \mathsf{e}{=}0\}\ \mathsf{c}_2\ \{Q\}\end{array}}{\langle\mathcal{R}\cup\{P\};\mathcal{G}\rangle \vdash \{P\}\ \mathbf{if}\ (\mathsf{e})\ \mathbf{then}\ \mathsf{c}_1\ \mathbf{else}\ \mathsf{c}_2\ \{Q\}}\ (\textsc{ite})$$

$$\frac{\begin{array}{c}P\wedge \mathsf{e}{=}0 \Rightarrow Q \\ \langle\mathcal{R};\mathcal{G}\rangle \vdash \{P\wedge \mathsf{e}{\neq}0\}\ \mathsf{c}\ \{P\}\end{array}}{\langle\mathcal{R}\cup\{Q\};\mathcal{G}\rangle \vdash \{P\}\ \mathbf{while}(\mathsf{e})\ \mathsf{c}\ \{Q\}}\ (\textsc{while}) \qquad \frac{\begin{array}{c}P\Rightarrow P' \quad \mathcal{R}'\subseteq\mathcal{R} \quad \mathcal{G}'\subseteq\mathcal{G} \quad Q'\Rightarrow Q \\ \langle\mathcal{R}';\mathcal{G}'\rangle \vdash \{P'\}\ \mathsf{C}\ \{Q'\}\end{array}}{\langle\mathcal{R}\cup\{P,Q\};\mathcal{G}\rangle \vdash \{P\}\ \mathsf{C}\ \{Q\}}\ (\textsc{conseq})$$

$$\frac{\begin{array}{ccc}\langle\mathcal{R}_1;\mathcal{G}_1\rangle \vdash \{P_1\}\mathsf{C}_1\{Q_1\} & \langle\mathcal{R}_2;\mathcal{G}_2\rangle \vdash \{P_2\}\mathsf{C}_2\{Q_2\} & \langle\mathcal{R}_1;\mathcal{G}_1\rangle \text{ and } \langle\mathcal{R}_2;\mathcal{G}_2\rangle \text{ are non-interfering} \\ Q_1\wedge Q_2 \Rightarrow Q & \mathtt{fr}(\mathcal{R}_1,\mathsf{C}_1)\cap\mathtt{wr}(\mathsf{C}_2){=}\emptyset & \mathtt{fr}(\mathcal{R}_2,\mathsf{C}_2)\cap\mathtt{wr}(\mathsf{C}_1){=}\emptyset\end{array}}{\langle\mathcal{R}_1\cup\mathcal{R}_2\cup\{Q\};\mathcal{G}_1\cup\mathcal{G}_2\rangle \vdash \{P_1\wedge P_2\}\ \mathsf{C}_1\,||\,\mathsf{C}_2\ \{Q\}}\ (\textsc{par})$$

$$\frac{P\Rightarrow P' \quad \langle\top;\top\rangle \vdash \{P'\}\ \mathsf{C}\ \{Q'\} \quad \langle\top;\top\rangle \vdash \{R\}\ \mathsf{C}_{\mathsf{rec}}\ \{Q'\} \quad Q'\Rightarrow Q \quad Q'\Rightarrow \exists\overline{v}.\,R[\overline{v/\mathsf{a}}][\overline{x_{\mathsf{p}}/x_{\mathsf{s}}}][\overline{x_{\mathsf{p}}/x_{\mathsf{v}}}]}{\mathsf{C}_{\mathsf{rec}} \vdash \{P\}\ \mathsf{C}\ \{Q\}}\ (\textsc{rec})$$

Fig. 3. POG proof rules with the implicit assumption that the pre- and post-conditions are stable (Def. 2).

The sequential fragment of the language is given by the c grammar and includes the standard constructs of **skip**, loops, conditionals and sequential composition, as well as local variable assignment ($\mathsf{a}:=\mathsf{e}$), memory read from location x ($\mathsf{a}:=x$), memory write to x ($x:=\mathsf{e}$), memory persist (**flush** x), store fence (**sfence**), memory fence (**mfence**) and atomic RMW (read-modify-write) instructions. The RMW instruction $\mathsf{a}:=\mathbf{CAS}(x,\mathsf{e}_1,\mathsf{e}_2)$ denotes the atomic 'compare-and-swap', where the value of x is compared against e_1: if the values match then x is set to e_2 and 1 is returned in a; otherwise x is left unchanged and 0 is returned in a. Analogously, $\mathsf{a}:=\mathbf{FAA}(x,\mathsf{e})$ denotes the atomic 'fetch-and-add', where x is incremented by e and its old value is returned in a. Lastly, we model a multi-threaded program C as a function mapping each thread to its (sequential) program. We write $\mathsf{C}{=}\mathsf{c}_1\,||\cdots||\,\mathsf{c}_n$ when $dom(\mathsf{C}){=}\{\tau_1\cdots\tau_n\}$ and $\mathsf{C}(\tau_i){=}\mathsf{c}_i$, and write $\mathsf{C}_1\,||\,\mathsf{C}_2$ for $\mathsf{C}_1\uplus\mathsf{C}_2$. We lift a sequential program c to a program in Com and simply write c for $\mathsf{C}\triangleq[\tau\mapsto\mathsf{c}]$.

Instrumented Locations. Recall from §2 that in order to reason about the persistency behaviours of programs, we instrument the memory to track three separate versions for each memory location. To this end, we define the set of *instrumented locations* as $\textsc{iLoc}\triangleq\{x_{\mathsf{v}},x_{\mathsf{s}},x_{\mathsf{p}}\mid x\in\textsc{Loc}\}$, and define an instrumented memory, $\mathsf{IM}:\textsc{IMem}$, as a finite map from instrumented locations to values: $\textsc{IMem}:\textsc{ILoc}\xrightarrow{\text{fin}}\textsc{Val}$. As discussed in §2, for each memory location x:

(1) $\mathsf{IM}(x_{\mathsf{v}})$ denotes the *volatile* value of x, i.e. the value observed for x during the execution;
(2) $\mathsf{IM}(x_{\mathsf{s}})$ denotes the *synchronously persisted* value of x, i.e. the value observed for x after a crash, had the **flush** instructions executed synchronously; and

(3) $\mathsf{IM}(x_{\mathsf{p}})$ denotes the *persistent* value of x, i.e. the value observed for x after a crash.

Intuitively, $\mathsf{IM}(x_{\mathsf{p}})$ reflects the current value of x in memory, while $\mathsf{IM}(x_{\mathsf{v}})$ and $\mathsf{IM}(x_{\mathsf{s}})$ record additional information that enables persistent reasoning, as discussed in §2.

Assertions. The POG assertions represent sets of states in our Ix86_{sim} semantics. Our assertion language is that of first order logic with *equality* and i) three constant symbols x_{v}, x_{s}, x_{p} for each location x; ii) a constant symbol a for each register; and iii) a constant symbol v for each value.

3.1 The POG Proof System

The *POG triples* are of the form: $\langle \mathcal{R}; \mathcal{G} \rangle \vdash \{P\}\ \mathsf{C}\ \{Q\}$, stating that: (1) the persistent parts of Q (describing persistent versions of variables, e.g. x_{p}) are *invariant* throughout the execution of C; (2) every *terminating* run of C from a state in P results in a state in Q; (3) C updates the state in accordance with $\mathcal{G}$ while satisfying the prescribed guards; and (4) the above holds when C is run in parallel with any program C', provided that the C' updates preserve the assertions in $\mathcal{R}$. That is, while the persistent parts of Q hold at *all points* during the execution of C, those parts describing register values and volatile/synchronous versions hold only when C terminates successfully.

POG Proof Rules. We present the *POG proof rules* in Fig. 3. Ignoring the (REC) rule at the bottom, most rules remain largely unchanged from their OG counterparts and are merely adapted to RG-style as in [Lahav and Vafeiadis 2015]. Intuitively, the pre- and post-conditions of triples are accumulated in the rely component $\mathcal{R}$ to ensure they remain stable (invariant) under the updates performed by other threads. Conversely, each time a thread updates a location x via (WRITE), (CAS) and (FAA), this is recorded in its guarantee component $\mathcal{G}$ with the corresponding precondition. Moreover, the (READ), (WRITE), (CAS) and (FAA) rules accessing memory location x have been accordingly adjusted to access the latest value of x, namely that in x_{v}.

As discussed in §2.3, when $x \in X$, **flush** x propagates the latest persist-pending value of each $x' \in X$ to memory. This is reflected in the $P \Rightarrow Q[x_{\mathsf{v}}^1/x_{\mathsf{s}}^1 \ldots x_{\mathsf{v}}^n/x_{\mathsf{s}}^n]$ premise of the (FLUSH) rule.

The (PAR) rule describes the concurrent execution $\mathsf{C}_1 \,||\, \mathsf{C}_2$, where the non-interference premise ensures that C_1 and C_2 do not interfere with one another. Intuitively, C_1 and C_2 are non-interfering iff each update performed by C_1 preserves the state assumptions of C_2, and vice versa. Put formally, when C_1 and C_2 are respectively run under contexts $\langle \mathcal{R}_1; \mathcal{G}_1 \rangle$ and $\langle \mathcal{R}_2; \mathcal{G}_2 \rangle$, then every update $\langle x, \mathsf{e}, P \rangle$ of C in $\mathcal{G}_1$ must preserve every assertion R in $\mathcal{R}_2$, i.e. $P \wedge R \Rightarrow R[\mathsf{e}/x_{\mathsf{v}}]$; and vice versa.

Definition 1. The tuples $\langle \mathcal{R}_1; \mathcal{G}_1 \rangle$ and $\langle \mathcal{R}_2; \mathcal{G}_2 \rangle$ are *non-interfering* iff:

- for all $\langle x_{\mathsf{v}}, \mathsf{e}, P \rangle \in \mathcal{G}_1$ and all $R \in \mathcal{R}_2$: $R \wedge P \Rightarrow R[\mathsf{e}/x_{\mathsf{v}}]$
- for all $\langle x_{\mathsf{v}}, \mathsf{e}, P \rangle \in \mathcal{G}_2$ and all $R \in \mathcal{R}_1$: $R \wedge P \Rightarrow R[\mathsf{e}/x_{\mathsf{v}}]$

Stability. Recall from §2.4 that we require assertions to be stable against volatile-to-synchronous and synchronous-to-persistent version propagations. This is formalised in Def. 2.

Definition 2. An assertion P is *stable*, written $\mathsf{stable}(P)$, iff $P \Rightarrow P[\overline{x_{\mathsf{s}}/x_{\mathsf{p}}}]$ and $\forall x_{\mathsf{s}}.\ P \Rightarrow P[x_{\mathsf{v}}/x_{\mathsf{s}}]$.

Reasoning about Crash Recovery. The POG triples discussed thus far are of the form $\langle \mathcal{R}; \mathcal{G} \rangle \vdash \{P\}\mathsf{C}\{Q\}$, where Q describes the state once C has *terminated* (i.e. without crashing). However, as discussed in §2, the execution of C may crash (e.g. due to power loss), at which point the volatile and synchronous versions (e.g. x_{v} and x_{s}) are lost while the persistent versions (e.g. x_{p}) are preserved, and the execution is resumed by running a *recovery program*. As such, in anticipation of a possible crash at any point, we require the persistent parts of Q to be *invariant* throughout the execution.

In order to reason about programs in the presence of crashes, we present POG *recovery* triples of the form $\mathsf{C}_{\mathsf{rec}} \vdash \{P\}\mathsf{C}\{Q\}$, stating that every run of C from a state in P either: (1) terminates

successfully (without crashing) in a state in Q; or (2) crashes and its execution is resumed by repeatedly running the recovery program $\mathsf{C}_{\mathsf{rec}}$ until $\mathsf{C}_{\mathsf{rec}}$ terminates successfully in a state in Q. That is, the execution of $\mathsf{C}_{\mathsf{rec}}$ may itself crash and is rerun repeatedly until it terminates successfully. In other words, the volatile (and synchronous) values in the postcondition of POG triples describe the post-states *only if* C does not crash, whereas the volatile (and synchronous) values in the postcondition of recovery triples always describe the post-states after termination (either after C terminates successfully, or after C crashes and $\mathsf{C}_{\mathsf{rec}}$ is run repeatedly until it terminates successfully).

The recovery rule (REC) is given at the bottom of Fig. 3. The $\langle \top; \top \rangle \vdash \{P'\}\mathsf{C}\{Q'\}$ premise ensures that executing C from a state in P' either terminates successfully in a state in Q', or it crashes and the recovery $\mathsf{C}_{\mathsf{rec}}$ is run thereafter from a state in R, obtained from Q' by resetting register values and replacing volatile/synchronous versions with persistent ones ($Q' \Rightarrow \exists \overline{v}.\, R[\overline{v/\mathsf{a}}][\overline{x_{\mathsf{p}}/x_{\mathsf{s}}}][\overline{x_{\mathsf{p}}/x_{\mathsf{v}}}]$), as they are lost upon a crash. The existential quantification of $\overline{v}$ assigns (havocs) arbitrary values to local registers $\overline{a}$ upon recovery, ensuring that R makes no assumptions about the post-crash values of registers. The $\langle \top; \top \rangle \vdash \{R\}\mathsf{C}_{\mathsf{rec}}\{Q'\}$ in turn ensures that executing $\mathsf{C}_{\mathsf{rec}}$ from R either terminates successfully in Q', or it crashes and is rerun from R. Put together, as $P \Rightarrow P'$ and $Q' \Rightarrow Q$, this ensures that executing C (under $\mathsf{C}_{\mathsf{rec}}$) from P eventually terminates successfully in Q. Lastly, the RG contexts $\langle \top; \top \rangle$ ensure that C and $\mathsf{C}_{\mathsf{rec}}$ are run as closed programs (i.e. not in parallel with another program). In §4 we present an example of using (REC) to reason about recovery.

Fence Proof Rules. Note that our proof system in Fig. 3 does not include rules for **sfence** and **mfence**. For **sfence**, we can simply extend our rules with: $\langle \{P\}; \emptyset \rangle \vdash \{P\}$ **sfence** $\{P\}$. That is, **sfence** acts as a no-op in our Ix86_{sim} model and has the same specification as (SKIP). This is caused by two factors. First recall that as discussed in §2.3, under Ix86_{sim} a **flush** x instruction with $x \in X$ copies x'_{v} to x'_{s} for each $x' \in X$. That is, Ix86_{sim} *eliminates* each **flush** instruction on $x \in X = \{x^1 \cdots x^n\}$, and simply treats it as a series of writes on $x^1_{\mathsf{s}} \cdots x^n_{\mathsf{s}}$. Second, as noted by Raad et al. [2020], in the absence of **flush**/$\textbf{flush}_{\text{opt}}$ instructions, **sfence** instructions behave as no-ops (**skip**) and impose no additional ordering constraints. As such, since Ix86_{sim} eliminates **flush** instructions (and treats them as writes) and excludes $\textbf{flush}_{\text{opt}}$ instructions by design, **sfence** is a no-op under Ix86_{sim}. Nevertheless, as discussed in §2.1 and detailed later in in §7, **sfence** instructions can be used to enforce additional ordering constraints on $\textbf{flush}_{\text{opt}}$ instructions, allowing us in most cases to transform programs using $\textbf{flush}_{\text{opt}}$ to those using **flush** instead. We therefore opt to include **sfence** in the POG programming language to facilitate such transformations.

For **mfence**, we can derive reasoning principles by treating them as RMW instructions (as in Fig. 7 of [Lahav and Vafeiadis 2015]). More concretely, we can treat **mfence** instructions as RMWs (e.g. **FAA**) on a designated location f, which enforces a global order on all **mfence** instructions.

4 EXAMPLES

We use the POG proof rules to verify several representative examples.

Example 1. We begin with the concurrent example in Fig. 1e, with its proof sketch given in Fig. 4a. The RG contexts of the left and right threads are given respectively by $\langle \mathcal{R}_1; \mathcal{G}_1 \rangle$ and $\langle \mathcal{R}_2; \mathcal{G}_2 \rangle$, defined in Fig. 4a. As discussed in §3, the rely of each thread contains the assertions used in its proof outline (e.g. P_1 in $\mathcal{R}_1$), while the guarantee contains a guarded assignment for each write performed by the thread (e.g. $\langle P_2, y_{\mathsf{v}}, 1 \rangle$ in $\mathcal{G}_1$). The invariant we are interested in establishing is given by I. The first three conjuncts capture the chronological order between the different versions of x and z. The penultimate conjunct states that when z_{v}=1 (i.e. once the second thread executes $z := 1$ after having read 1 for y_{v} in a), then x_{s}=1 (i.e. **flush** x' must have already executed). The last conjunct is that of main interest, corresponding to the desired property in Fig. 1e. Note that the assertions at each program point are stable, and that $\langle \mathcal{R}_1; \mathcal{G}_1 \rangle$ and $\langle \mathcal{R}_2; \mathcal{G}_2 \rangle$ are non-interfering.

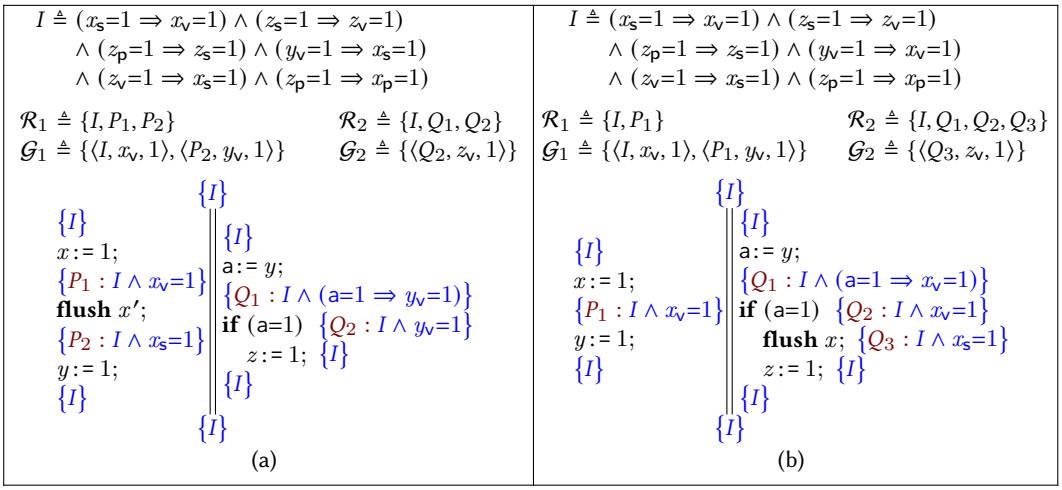

Fig. 4. Proof sketches of Example 1 (a) and Example 2 (b)

Example 2. We proceed with the example in Fig. 4b which is an adaptation of Fig. 4a with the **flush** moved to the right thread after reading from y. As such, the invariant I is similar to that of Fig. 4a, with the main difference lying in the fourth conjunct. In particular, when the left thread executes $y := 1$ yielding $y_v{=}1$, the earlier $x := 1$ has already executed, i.e. $x_v{=}1$. However, due to the absence of an intervening **flush** between the two writes, unlike in Fig. 4a we cannot assert $x_s{=}1$.

Example 3 (Atomic persists)**.** Our next example in Fig. 5a is inspired by the *persistent transactions* of Raad et al. [2019], where the authors showed how to ensure multiple stores on different locations (appear to) persist atomically before subsequent stores. Let us write τ_1 and τ_2 for the left and right threads, respectively. As shown in Fig. 5a , τ_1 writes 1 to x and y, and τ_2 writes 1 to z only if x contains 1, i.e. only if τ_1 has already executed $x := 1$. Our goal in this example is to ensure that the writes on x and y (by τ_1) both persist before the write on z (by τ_2). That is, we must ensure that the **flush** instructions of τ_1 are executed before τ_2 executes $z := 1$.

To this end, since τ_2 writes to z only after reading 1 from x (written by τ_1), we use a *lock* to control the accesses on x. More concretely, τ_1 acquires the lx lock on x at the beginning and releases it only after executing its **flush** instructions. Similarly, τ_2 acquires the lx lock prior to accessing x. As such, if τ_2 reads 1 for x (i.e. observes $x := 1$ by τ_1) and executes $z := 1$, then it must have acquired lx after it was released by τ_1, i.e. after τ_1 executed its **flush** instructions, as required.

The lx lock is acquired by calling **lock**, implemented as a spin lock in Fig. 5a: the implementation of **lock**(lx, a, v) loops until lx is free (i.e. $lx{=}0$), at which point it acquires it by atomically setting it to the non-zero value v. In order to distinguish which thread currently holds the lock, τ_1 and τ_2 respectively write the distinct values 1 and 2 to lx upon acquiring it. Lastly, the a argument denotes a thread-local register used as the loop flag by **lock**, and is passed on to **lock** by the calling thread.

We present a proof sketch of this program in Fig. 5a, where for brevity we have omitted the RG contexts. As before, our goal is to establish the I invariant, with its first, second and fourth conjuncts describing the chronological order on different versions of x, y and z. The third conjunct states that once τ_1 has finished executing, i.e. it has released the lock ($lx \in \{0, 2\}$) having written 1 to x ($x_v{=}1$), its flush instructions have also executed ($x_s{=}y_s{=}1$). Similarly, the penultimate conjunct states that once τ_2 has written 1 to z ($z_v{=}1$) having read 1 for x, i.e. after acquiring lx once it is released by τ_1 (see above), then the τ_1 **flush** instructions must have already executed ($x_s{=}y_s{=}1$).

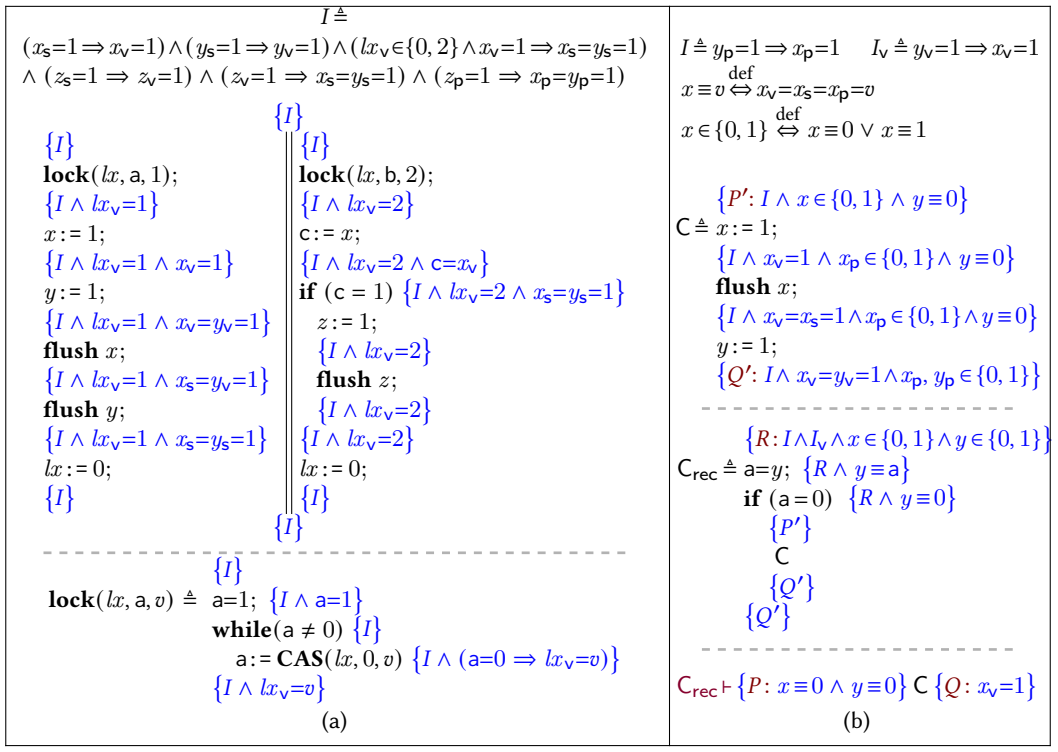

Fig. 5. Proof sketches of Example 3 (a) and Example 4 (b)

The last conjunct describes the desired persist ordering: if the write on z has persisted ($z_p{=}1$), then so must have both writes on x and y ($x_p{=}y_p{=}1$), as required.

Example 4 (Recovery)**.** In Fig. 5b we show how to use the (REC) rule to reason about recovery. The original program C is similar to that in Fig. 1b: first 1 is written to x and persisted by calling **flush**, and then 1 is written to y. As in Fig. 1b, the intervening **flush** ensures that $x := 1$ persists before $y := 1$ and thus $y_p{=}1 \Rightarrow x_p{=}1$ as described by I, assuming initially $x \equiv 0$ (i.e. $x_v{=}x_s{=}x_p{=}0$) and $y \equiv 0$.

As $I \triangleq y_p{=}1 \Rightarrow x_p{=}1$ is invariant throughout the execution (I only concerns persistent versions), the recovery C_{rec} treats the write on y as a flag to ascertain if $x := 1$ has persisted. That is, if y holds 1 upon recovery, then x must also hold 1 and thus C_{rec} simply returns; otherwise, C_{rec} reruns C.

We present a proof sketch of C and C_{rec} in Fig. 5b. We first establish $\langle \top; \top \rangle \vdash \{P'\}C\{Q'\}$ and $\langle \top; \top \rangle \vdash \{R\}C_{rec}\{Q'\}$, and then use (REC) to prove $C_{rec} \vdash \{P\}C\{Q\}$. Note that as mandated by the (REC) premise, we must show $P \Rightarrow P'$, $Q' \Rightarrow Q$, and $Q' \Rightarrow R[x_p/x_s][\overline{x_p}/\overline{x_v}]$ which follow immediately.

5 THE Ix86$_{sim}$ OPERATIONAL SEMANTICS

We present the Ix86$_{sim}$ operational semantics discussed in §2.3. Recall that in Ix86$_{sim}$ a memory operation on x manipulates x_v; when $x \in X$, **flush** x copies x'_v to x'_s for $x' \in X$; x_v may be copied to x_s non-deterministically; and $\overline{x_s}$ may be non-deterministically copied to $\overline{x_p}$. To model this, we define a *translation function* that transforms Px86$_{sim}$ programs to access the instrumented memory.

Translation. Our translation function, $[\![.]\!]$, is defined in Fig. 6 and uses the auxiliary function, $(\!|.|\!)$, to translate sequential programs. As discussed, memory accesses on x are translated to access x_v; conditionals, loops and sequential composition are translated inductively; and **flush** x is translated

$(\!|\mathsf{a}:=\mathsf{e}|\!) \triangleq \mathsf{a}:=\mathsf{e}$ $\quad (\!|\mathsf{a}:=x|\!) \triangleq \mathsf{a}:=x_{\mathsf{v}}$ $\quad (\!|x:=\mathsf{e}|\!) \triangleq x_{\mathsf{v}}:=\mathsf{e}$ $\quad (\!|\mathsf{a}:=\mathbf{CAS}(x,\mathsf{e}_1,\mathsf{e}_2)|\!) \triangleq \mathsf{a}:=\mathbf{CAS}(x_{\mathsf{v}},\mathsf{e}_1,\mathsf{e}_2)$

$(\!|\mathsf{a}:=\mathbf{FAA}(x,\mathsf{e})|\!) \triangleq \mathsf{a}:=\mathbf{FAA}(x_{\mathsf{v}},\mathsf{e})$ $\quad (\!|\mathbf{sfence}|\!) \triangleq \mathbf{skip}$ $\quad (\!|\mathbf{mfence}|\!) \triangleq \mathbf{mfence}$ $\quad (\!|\mathbf{flush}\ x|\!) \triangleq \langle \mathbf{persist}\ X \rangle$

$(\!|\mathbf{if}\ (\mathsf{e})\ \mathbf{then}\ \mathsf{c}_1\ \mathbf{else}\ \mathsf{c}_2|\!) \triangleq \mathbf{if}\ (\mathsf{e})\ \mathbf{then}\ (\!|\mathsf{c}_1|\!)\ \mathbf{else}\ (\!|\mathsf{c}_2|\!)$ $\quad (\!|\mathbf{while}(\mathsf{e})\ \mathsf{c}|\!) \triangleq \mathbf{while}(\mathsf{e})\ (\!|\mathsf{c}|\!)$

$(\!|\mathsf{c}_1;\mathsf{c}_2|\!) \triangleq (\!|\mathsf{c}_1|\!);(\!|\mathsf{c}_2|\!)$ $\quad [\![\mathsf{c}_1 \,||\cdots||\, \mathsf{c}_n]\!] \triangleq (\!|\mathsf{c}_1|\!) \,||\cdots||\, (\!|\mathsf{c}_n|\!) \,||\, \mathsf{c}_{\mathsf{s}} \,||\, \mathsf{c}_{\mathsf{p}}$

with $\quad \mathsf{c}_{\mathsf{s}} \triangleq \mathbf{while}(*)\ \langle \mathbf{pick}\ x;\ x_{\mathsf{s}} := x_{\mathsf{v}} \rangle$ $\quad \mathsf{c}_{\mathsf{p}} \triangleq \mathbf{while}(*)\ \langle \overline{x_{\mathsf{p}} := x_{\mathsf{s}}}; \rangle$

Fig. 6. Ix86_{sim} program translation where we assume $x \in X$

to persist X (when $x \in X$) *atomically*, as indicated by $\langle . \rangle$. That is, in one computation step **persist** X reads the value of x_{v} and subsequently copies it to x_{s} for each $x \in X$, as we describe shortly.

The **mfence** instructions are left unchanged by the translation, while **sfence** instructions are translated as **skip**. This is because as discussed in §3, **sfence** behaves as a no-op in the absence of **flush**/$\mathbf{flush}_{\text{opt}}$. As such, since our translation eliminates **flush** and our language excludes $\mathbf{flush}_{\text{opt}}$, **sfence** is simply translated to **skip**. Lastly, the translation of a concurrent program is obtained from the point-wise translation of each thread, run in parallel with c_{s} and c_{p}. Intuitively, c_{s} models the non-deterministic propagation of x_{v} to x_{s} for an arbitrary x, carried out atomically. Analogously, c_{p} models the non-deterministic propagation of $\overline{x_{\mathsf{s}}}$ to $\overline{x_{\mathsf{p}}}$, for all locations. We write τ_{s} and τ_{p} for the threads executing c_{s} and c_{p}, respectively. Given a program C, we write C^{sp} for $\mathsf{C} \,||\, \mathsf{c}_{\mathsf{s}} \,||\, \mathsf{c}_{\mathsf{p}}$.

We next describe the Ix86_{sim} operational semantics 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 and thread-local buffers in Fig. 2a) determine the execution steps. The Ix86_{sim} operational semantics is then defined by combining the transitions of its program and storage subsystems.

Program Transitions. The Ix86_{sim} program transitions are given at the top of Fig. 7 and are defined via the transitions of their constituent threads. Thread transitions are of the form: $\mathsf{c}, \mathsf{S} \xrightarrow{\tau:l} \mathsf{c}', \mathsf{S}'$, where $\mathsf{c}, \mathsf{c}' \in \textsc{SCom}$ denote translated sequential programs, and $\mathsf{S}, \mathsf{S}' \in \textsc{Stack}$ denote *stacks* mapping registers to values. The $\tau{:}l$ marks the transition by recording the executing thread τ, and the transition *label* l. A label may be ϵ for silent transitions of no-ops; $(\mathtt{R}, x, v)$ for reading v from x; $(\mathtt{W}, x, v)$ for writing v to x; $(\mathtt{U}, x, v, v')$ for a successful RMW (update) modifying the value of x to v' when its value matches v; $\mathtt{MF}$ for executing an **mfence**; $(\mathtt{FL}, X)$ for persisting the X cache line; $(\mathtt{S}, x_{\mathsf{s}}, v)$ for the atomic propagation of v to x_{s}; and $(\mathtt{P}, \overline{x_{\mathsf{p}}^i}, \overline{v^i})$ for the atomic propagation of $\overline{v^i}$ to $\overline{x_{\mathsf{p}}^i}$.

Given an expression e, we write $\mathsf{S}(\mathsf{e})$ for the value to which e evaluates under S; this definition is standard and omitted. Most program transitions are standard. The (P-MF) transition describes executing an **mfence**. The (P-CAS0) transition describes the unsuccessful execution of $\mathbf{CAS}(x, \mathsf{e}_1, \mathsf{e}_2)$; i.e. when the value read (v) is different from $\mathsf{S}(\mathsf{e}_1)$. The (P-CAS1) transition dually describes the successful execution of **CAS**. Note that in the failure case no update takes place and the transition is labelled with a read, and not an update as in the success case. The (P-FAA) transition behaves analogously. When executing **persist** X (i.e. a translated **flush** x with $x \in X$), the volatile-to-synchronous propagation of X is modelled by the $(\mathtt{FL}, X)$ transition in (P-AtomFL). The volatile-to-synchronous propagation of c_{s} is modelled by $(\mathtt{S}, x_{\mathsf{s}}, v)$ in (P-AtomS); *mutatis mutandis* for (P-AtomP).

Storage Transitions. The Ix86_{sim} storage transitions are given at the bottom of Fig. 7 and are of the form: $\mathsf{IM}, \mathsf{B} \xrightarrow{\tau:l} \mathsf{IM}', \mathsf{B}'$, where $\mathsf{IM}, \mathsf{IM}'$ denote the instrumented memory, and B, B' denote the *buffer map*, associating each thread with its *buffer*. Each buffer entry may be of the form: (1) $\langle x_{\mathsf{v}}, v \rangle$, denoting a pending write of value v on x_{v}; or (2) $X \subseteq \textsc{Loc}$, denoting a pending **flush** on cache line

Program transitions: $\textsc{Com} \times \textsc{Stack} \xrightarrow{\textsc{Tid}:\textsc{Lab}\cup\{\epsilon\}} \textsc{Com} \times \textsc{Stack}$ $\quad \mathsf{S} \in \textsc{Stack} \triangleq \textsc{Reg} \xrightarrow{\text{fin}} \textsc{Val}$

$\textsc{Lab} \triangleq \left\{ (\mathtt{R}, x_\mathsf{v}, v), (\mathtt{W}, x_\mathsf{v}, v), (\mathtt{U}, x_\mathsf{v}, v, v'), \mathtt{MF}, (\mathtt{FL}, X), (\mathtt{S}, x_\mathsf{s}, v), (\mathtt{P}, \overline{x_\mathsf{p}^i, v^i}) \;\middle|\; x, x^i \in \textsc{Loc} \wedge X \in \textsc{CL} \wedge v, v', v^i \in \textsc{Val} \right\}$

$$\frac{\mathsf{S}'{=}\mathsf{S}[\mathsf{a} \mapsto \mathsf{S}(\mathsf{e})]}{\mathsf{a} := \mathsf{e}, \mathsf{S} \xrightarrow{\tau:\epsilon} \mathbf{skip}, \mathsf{S}'} \text{(P-Assign)} \qquad \frac{\mathsf{S}'{=}\mathsf{S}[\mathsf{a} \mapsto v]}{\mathsf{a} := x_\mathsf{v}, \mathsf{S} \xrightarrow{\tau:(\mathtt{R}, x_\mathsf{v}, v)} \mathbf{skip}, \mathsf{S}'} \text{(P-Read)} \qquad \frac{}{\mathbf{mfence}, \mathsf{S} \xrightarrow{\tau:\mathtt{MF}} \mathbf{skip}, \mathsf{S}} \text{(P-MF)}$$

$$\frac{\mathsf{S}(\mathsf{e}){=}v}{x_\mathsf{v} := \mathsf{e}, \mathsf{S} \xrightarrow{\tau:(\mathtt{W}, x_\mathsf{v}, v)} \mathbf{skip}, \mathsf{S}} \text{(P-Write)} \qquad \frac{\mathsf{S}(\mathsf{e}_1) \neq v \quad \mathsf{S}'{=}\mathsf{S}[\mathsf{a} \mapsto v]}{\mathsf{a} := \mathbf{CAS}(x_\mathsf{v}, \mathsf{e}_1, \mathsf{e}_2), \mathsf{S} \xrightarrow{\tau:(\mathtt{R}, x_\mathsf{v}, v)} \mathbf{skip}, \mathsf{S}'} \text{(P-CAS0)}$$

$$\frac{\mathsf{S}(\mathsf{e}){=}v \quad \mathsf{S}'{=}\mathsf{S}[\mathsf{a} \mapsto v_0]}{\mathsf{a} := \mathbf{FAA}(x_\mathsf{v}, \mathsf{e}), \mathsf{S} \xrightarrow{\tau:(\mathtt{U}, x_\mathsf{v}, v_0, v_0+v)} \mathbf{skip}, \mathsf{S}'} \text{(P-FAA)} \qquad \frac{\mathsf{S}(\mathsf{e}_1){=}v_1 \quad \mathsf{S}(\mathsf{e}_2){=}v_2 \quad \mathsf{S}'{=}\mathsf{S}[\mathsf{a} \mapsto v_1]}{\mathsf{a} := \mathbf{CAS}(x_\mathsf{v}, \mathsf{e}_1, \mathsf{e}_2), \mathsf{S} \xrightarrow{\tau:(\mathtt{U}, x_\mathsf{v}, v_1, v_2)} \mathbf{skip}, \mathsf{S}'} \text{(P-CAS1)}$$

$$\frac{}{\langle \mathbf{persist}\ X \rangle, \mathsf{S} \xrightarrow{\tau:(\mathtt{FL}, X)} \mathbf{skip}, \mathsf{S}} \text{(P-AtomFL)} \qquad \frac{\mathsf{S}(\mathsf{e}){\neq}0 \Rightarrow \mathsf{c}{=}\mathsf{c}_1 \quad \mathsf{S}(\mathsf{e}){=}0 \Rightarrow \mathsf{c}{=}\mathsf{c}_2}{\mathbf{if}\ (\mathsf{e})\ \mathbf{then}\ \mathsf{c}_1\ \mathbf{else}\ \mathsf{c}_2, \mathsf{S} \xrightarrow{\tau:\epsilon} \mathsf{c}, \mathsf{S}} \text{(P-If)}$$

$$\frac{}{\mathbf{while}(\mathsf{e})\ \mathsf{c}, \mathsf{S} \xrightarrow{\tau:\epsilon} \mathbf{if}\ (\mathsf{e})\ \mathbf{then}\ \mathsf{c}; (\mathbf{while}(\mathsf{e})\ \mathsf{c})\ \mathbf{else}\ \mathbf{skip}, \mathsf{S}} \text{(P-While)} \qquad \frac{}{\mathbf{skip}; \mathsf{c}, \mathsf{S} \xrightarrow{\tau:\epsilon} \mathsf{c}, \mathsf{S}} \text{(P-Seq1)}$$

$$\frac{}{\langle \mathbf{pick}\ x; x_\mathsf{s} := x_\mathsf{v} \rangle, \mathsf{S} \xrightarrow{\tau_\mathsf{s}:(\mathtt{S}, x_\mathsf{s}, v)} \mathbf{skip}, \mathsf{S}} \text{(P-AtomS)} \qquad \frac{\mathsf{c}_1, \mathsf{S} \xrightarrow{\tau:l} \mathsf{c}_1', \mathsf{S}'}{\mathsf{c}_1; \mathsf{c}_2, \mathsf{S} \xrightarrow{\tau:l} \mathsf{c}_1'; \mathsf{c}_2, \mathsf{S}'} \text{(P-Seq2)}$$

$$\frac{}{\langle \overline{x_\mathsf{p}^i := x_\mathsf{s}^i} \rangle, \mathsf{S} \xrightarrow{\tau_\mathsf{p}:(\mathtt{P}, \overline{x_\mathsf{p}^i, v^i})} \mathbf{skip}, \mathsf{S}} \text{(P-AtomP)} \qquad \frac{\mathsf{C}(\tau), \mathsf{S} \xrightarrow{\tau:l} \mathsf{c}, \mathsf{S}'}{\mathsf{C}, \mathsf{S} \xrightarrow{\tau:l} \mathsf{C}[\tau \mapsto \mathsf{c}], \mathsf{S}'} \text{(P-Par)}$$

Storage transitions: $\textsc{IMem} \times \textsc{IBMap} \xrightarrow{\textsc{Tid}:\textsc{Lab}\cup\{\epsilon\}} \textsc{IMem} \times \textsc{IBMap}$ $\quad \mathsf{B} \in \textsc{IBMap} \triangleq \textsc{Tid} \xrightarrow{\text{fin}} \textsc{IBuff}$

$\mathsf{b} \in \textsc{IBuff} \triangleq \textsc{Seq}\,\langle (\textsc{ILoc} \times \textsc{Val}) \cup \mathcal{P}\,(\textsc{Loc}) \rangle$

$$\frac{\mathsf{B}(\tau){=}\mathsf{b}}{\mathsf{IM}, \mathsf{B} \xrightarrow{\tau:(\mathtt{W}, x_\mathsf{v}, v)} \mathsf{IM}, \mathsf{B}[\tau \mapsto \mathsf{b}.\langle x_\mathsf{v}, v \rangle]} \text{(M-Write)} \qquad \frac{\mathsf{B}(\tau){=}\epsilon \quad \mathsf{IM}(x_\mathsf{v}){=}v_1}{\mathsf{IM}, \mathsf{B} \xrightarrow{\tau:(\mathtt{U}, x_\mathsf{v}, v_1, v_2)} \mathsf{IM}[x_\mathsf{v} \mapsto v_2], \mathsf{B}} \text{(M-RMW)}$$

$$\frac{\mathsf{rd}(\mathsf{IM}, \mathsf{B}(\tau), x_\mathsf{v}){=}v}{\mathsf{IM}, \mathsf{B} \xrightarrow{\tau:(\mathtt{R}, x_\mathsf{v}, v)} \mathsf{IM}, \mathsf{B}} \text{(M-Read)} \qquad \frac{\mathsf{B}(\tau){=}\epsilon}{\mathsf{IM}, \mathsf{B} \xrightarrow{\tau:\mathtt{MF}} \mathsf{IM}, \mathsf{B}} \text{(M-MF)} \qquad \frac{\mathsf{IM}(x_\mathsf{v}){=}v}{\mathsf{IM}, \mathsf{B} \xrightarrow{\tau_\mathsf{s}:(\mathtt{S}, x_\mathsf{s}, v)} \mathsf{IM}[x_\mathsf{s} \mapsto v], \mathsf{B}} \text{(M-AtomS)}$$

$$\frac{\forall i.\, \mathsf{IM}(x_\mathsf{s}^i){=}v^i}{\mathsf{IM}, \mathsf{B} \xrightarrow{\tau_\mathsf{p}:(\mathtt{P}, \overline{x_\mathsf{p}^i, v^i})} \mathsf{IM}[\overline{x_\mathsf{p}^i \mapsto v^i}], \mathsf{B}} \text{(M-AtomP)} \qquad \frac{\mathsf{B}(\tau){=}\mathsf{b}}{\mathsf{IM}, \mathsf{B} \xrightarrow{\tau:(\mathtt{FL}, X)} \mathsf{IM}, \mathsf{B}[\tau \mapsto \mathsf{b}.X]} \text{(M-AtomFL)}$$

$$\frac{\mathsf{B}(\tau){=}\langle x_\mathsf{v}, v \rangle.\mathsf{b}}{\mathsf{IM}, \mathsf{B} \xrightarrow{\tau:\epsilon} \mathsf{IM}[x_\mathsf{v} \mapsto v], \mathsf{B}[\tau \mapsto \mathsf{b}]} \text{(M-PropW)} \qquad \frac{\mathsf{B}(\tau){=}X.\mathsf{b} \qquad X{=}\overline{x^i} \qquad \forall i.\, \mathsf{IM}(x_\mathsf{v}^i){=}v^i}{\mathsf{IM}, \mathsf{B} \xrightarrow{\tau:\epsilon} \mathsf{IM}[\overline{x_\mathsf{s}^i \mapsto v^i}], \mathsf{B}[\tau \mapsto \mathsf{b}]} \text{(M-PropFL)}$$

with $\quad \mathsf{rd}(\mathsf{IM}, \mathsf{b}, x_\mathsf{v}) \triangleq \begin{cases} v & \text{if } \exists \mathsf{b}_1, \mathsf{b}_2.\ \mathsf{b}{=}\mathsf{b}_1.\langle x_\mathsf{v}, v \rangle.\mathsf{b}_2 \wedge \forall v'.\ \langle x_\mathsf{v}, v' \rangle \notin \mathsf{b}_2 \\ \mathsf{IM}(x_\mathsf{v}) & \text{otherwise} \end{cases}$

Fig. 7. The Ix86_{sim} program transitions (above); the Ix86_{sim} storage transitions (below)

X. When a thread writes v to x_v, this is recorded in its buffer as the $\langle x_\mathsf{v}, v \rangle$ entry, as described by (M-Write). Similarly, when a thread makes a $(\mathtt{FL}, X)$ transition (i.e. executes **persist** X translated from **flush** x with $x \in X$), this is recorded in its buffer as X, as shown in (M-AtomFL). Recall that

$$\textsc{Com} \vdash \textsc{Conf} \xrightarrow{\textsc{Tid}\times\textsc{Lab}\cup\{\epsilon,\lightning\}} \textsc{Conf} \qquad \textsc{Conf} \triangleq \textsc{Com} \times \textsc{Stack} \times \textsc{IMem} \times \textsc{IBMap}$$

$$\frac{\mathsf{C},\mathsf{S} \xrightarrow{\tau:\epsilon} \mathsf{C}',\mathsf{S}'}{\Delta \vdash \mathsf{C},\mathsf{S},\mathsf{IM},\mathsf{B} \xRightarrow{\tau:\epsilon} \mathsf{C}',\mathsf{S}',\mathsf{IM},\mathsf{B}}\ (\textsc{SilentP}) \qquad \frac{\mathsf{C},\mathsf{S} \xrightarrow{\tau:l} \mathsf{C}',\mathsf{S}' \quad \mathsf{IM},\mathsf{B} \xrightarrow{\tau:l} \mathsf{IM}',\mathsf{B}'}{\Delta \vdash \mathsf{C},\mathsf{S},\mathsf{IM},\mathsf{B} \xRightarrow{\tau:l} \mathsf{C}',\mathsf{S}',\mathsf{IM}',\mathsf{B}'}\ (\textsc{Step})$$

$$\frac{\mathsf{IM},\mathsf{B} \xrightarrow{\tau:\epsilon} \mathsf{IM}',\mathsf{B}'}{\Delta \vdash \mathsf{C},\mathsf{S},\mathsf{IM},\mathsf{B} \xRightarrow{\tau:\epsilon} \mathsf{C},\mathsf{S},\mathsf{IM}',\mathsf{B}'}\ (\textsc{SilentS}) \qquad \frac{\mathsf{B}_0 \triangleq \lambda\tau.\epsilon \quad \mathsf{IM}'{=}\mathsf{IM}[\overline{x_\mathsf{s} \mapsto \mathsf{IM}(x_\mathsf{p})}][\overline{x_\mathsf{v} \mapsto \mathsf{IM}(x_\mathsf{p})}]}{\mathsf{C}_\mathsf{rec} \vdash \mathsf{C},\mathsf{S},\mathsf{IM},\mathsf{B} \xRightarrow{\lightning} \mathsf{C}_\mathsf{rec},\mathsf{S}_0,\mathsf{IM}',\mathsf{B}_0}\ (\textsc{Crash})$$

Fig. 8. The Ix86_{sim} operational semantics

when a thread reads from x_v, it first consults its own buffer, followed by the memory (if no write to x_v is found in the buffer). This lookup chain is captured by $\mathsf{rd}(\mathsf{IM},\mathsf{b},x_\mathsf{v})$ in the premise of (M-Read).

The (M-MF) rule ensures that an **mfence** proceeds only when the buffer of the executing thread is empty, as stipulated by the $\mathsf{B}(\tau){=}\epsilon$ premise. In the (M-RMW) rule, when executing an RMW instruction on x_v, a similar lookup chain is followed to determine the value of x_v, as with a read. To ensure their atomicity, RMW instructions may only proceed when the buffer of the executing thread is drained. Moreover, the resulting update is committed directly to the memory, bypassing the thread buffer. This is to ensure that the resulting update is immediately visible to other threads.

As with (P-AtomS), (M-AtomS) describes the non-deterministic copying of x_v to x_s, with the result written directly to memory, provided that the buffer of the executing thread (i.e. τ_s) is empty; (P-AtomP) behaves analogously. Lastly, (M-PropW) describes the debuffering of a pending write in a thread-local buffer and propagating it to memory. Similarly, (M-PropFL) describes the debuffering of a pending flush on X, where the value of x_v^i (in $\mathsf{IM}(x_\mathsf{v}^i)$) is propagated to $\mathsf{IM}(x_\mathsf{s}^i)$, for each $x^i \in X$.

Combined Transitions. The Ix86_{sim} operational semantics in Fig. 8 is defined by combining the program and storage transitions, under a *recovery program*, C_rec, run after a crash. The (SilentP) rule describes the case when the program subsystem takes a silent step and thus the storage subsystem is left unchanged; similarly 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 registers (in S) and the thread-local buffers are lost (as they are volatile), and are thus reset; the memory is left largely unchanged (as it is non-volatile); and the execution is restarted with the recovery program. Note that upon a crash the persistent versions in the resulting memory (IM') remain unchanged, while the synchronous and volatile versions are lost and are simply overwritten by the persistent versions. This is because if $x_\mathsf{v} \neq x_\mathsf{p}$ (resp. $x_\mathsf{s} \neq x_\mathsf{p}$) upon a crash, then intuitively the write responsible for the current value of x_v (resp. x_s) has not yet reached the persistent memory and is thus lost after recovery.

Ix86_{sim} Subsumes Px86_{sim}. In Thm. 1 below we show that our Ix86_{sim} model is weaker than Px86_{sim} and subsumes all its behaviours. This then allows us to establish the soundness of POG over the simpler Ix86_{sim} model. To prove that Ix86_{sim} subsumes Px86_{sim}, we show that for all non-instrumented memories M produced by a Px86_{sim} trace, there exists an instrumented memory IM' produced by an Ix86_{sim} trace such that M and IM agree on the (persisted) values of all locations.

Theorem 1. *For all memories* M *produced by a* Px86_{sim} *trace, there exists an instrumented memory* IM' *produced by an* Ix86_{sim} *trace such that:* $\forall x.\ \mathsf{M}(x){=}\mathsf{IM}(x_\mathsf{p})$.

6 POG SOUNDNESS

We prove that the POG proof rules in Fig. 3 are sound. We proceed with several auxiliary definitions.

Assertion Semantics. We define the set of *states* as $\textsc{State} \triangleq \textsc{IMem} \times \textsc{Stack}$, and use σ as a metavariable for states. Assertions are interpreted as sets of states as expected: the instrumented memory provides the interpretation of the $x_{\mathsf{v}}, x_{\mathsf{s}}, x_{\mathsf{p}}$ constants, and the stack provides the interpretation of the a constants. In what follows we write $[\![P]\!]$ for $\{\sigma \mid \sigma \models P\}$.

$\mathcal{R}/\mathcal{G}$ Semantics. $\mathcal{R}/\mathcal{G}$ components are interpreted as relations on states. The guarantee $\mathcal{G}$ is interpreted point-wise as the smallest preorder that admits all constituent guarded assignments. That is, if $\langle x_{\mathsf{v}}, \mathsf{e}, P\rangle \in \mathcal{G}$ and $(\mathsf{IM}, \mathsf{S}) \in [\![P]\!]$, then $((\mathsf{IM}, \mathsf{S}), (\mathsf{IM}', \mathsf{S})) \in [\![\mathcal{G}]\!]^{\mathsf{g}}$, when $\mathsf{IM}'{=}\mathsf{IM}[x_{\mathsf{v}} \mapsto \mathsf{S}(\mathsf{e})]$:

$$[\![\emptyset]\!]^{\mathsf{g}} \triangleq \mathsf{id} \qquad [\![\mathcal{G} \cup \{\langle x_{\mathsf{v}}, \mathsf{e}, P\rangle\}]\!]^{\mathsf{g}} \triangleq \left([\![\mathcal{G}]\!]^{\mathsf{g}} \cup \left\{\left((\mathsf{IM}, \mathsf{S}), (\mathsf{IM}[x_{\mathsf{v}} \mapsto \mathsf{S}(\mathsf{e})], \mathsf{S})\right) \mid (\mathsf{IM}, \mathsf{S}) \models P\right\}\right)^*$$

Dually, the rely $\mathcal{R}$ is interpreted point-wise as the largest relation on states that preserves the stability of all constituent assertions. That is, if $P \in \mathcal{R}$, $\sigma \in [\![P]\!]$ and $(\sigma, \sigma') \in [\![\mathcal{R}]\!]^{\mathsf{r}}$, then $\sigma' \in [\![P]\!]$:

$$[\![\emptyset]\!]^{\mathsf{r}} \triangleq \textsc{State} \times \textsc{State} \qquad [\![\mathcal{R} \cup \{P\}]\!]^{\mathsf{r}} \triangleq [\![\mathcal{R}]\!]^{\mathsf{r}} \cap \left\{(\sigma, \sigma') \mid \sigma \models P \Rightarrow \sigma' \models P\right\}$$

Configuration Safety. We define the semantics of POG triples via an auxiliary predicate, $\mathsf{safe}_n(\mathsf{C}^{\mathsf{sp}}, \sigma, R, G, Q, I)$, stating that executing the translated program C^{sp} over the σ state is safe with respect to the interpreted R, G relations, post-states Q and invariant I for up to n steps.

Definition 3 (Configuration safety)**.** For all C, IM, S and $R, G \subseteq \textsc{State} \times \textsc{State}$, $Q \subseteq \textsc{State}$: $\mathsf{safe}_0(\mathsf{C}^{\mathsf{sp}}, (\mathsf{IM}, \mathsf{S}), R, G, Q, I)$ always holds; and $\mathsf{safe}_{n+1}(\mathsf{C}^{\mathsf{sp}}, (\mathsf{IM}, \mathsf{S}), R, G, Q, I)$ holds iff:

(1) $(\mathsf{IM}, \mathsf{S}) \in I$ and (2) if $\mathsf{C}{=}\mathsf{C}_{\mathbf{skip}}$, then $(\mathsf{IM}, \mathsf{S}) \in Q$, where $\mathsf{C}_{\mathbf{skip}} \triangleq \lambda\tau.\mathbf{skip}$

(3) for all σ: if $((\mathsf{IM}, \mathsf{S}), \sigma) \in R \cup \mathsf{A}_{\mathsf{sp}}$, then $\mathsf{safe}_n(\mathsf{C}^{\mathsf{sp}}, \sigma, R, G, Q, I)$

(4) for all $\tau, l, \mathsf{IM}_1, \mathsf{IM}_2, \mathsf{B}_1, \mathsf{B}_2, \mathsf{C}', \mathsf{IM}', \mathsf{S}', l{\neq}\not{l}$:

$$\text{if } \mathsf{C}, \mathsf{IM}_1, \mathsf{B}_1, \mathsf{S} \xRightarrow{\tau:l} \mathsf{C}', \mathsf{IM}_2, \mathsf{B}_2, \mathsf{S}' \wedge \mathsf{IM}{=}\Downarrow(\mathsf{IM}_1, \mathsf{B}_1, \tau) \wedge \mathsf{IM}'{=}\Downarrow(\mathsf{IM}_2, \mathsf{B}_2, \tau)$$
$$\text{then } ((\mathsf{IM}, \mathsf{S}), (\mathsf{IM}', \mathsf{S})) \in G \cup \mathsf{A}_{\mathsf{sp}} \wedge \mathsf{safe}_n((\mathsf{C}')^{\mathsf{sp}}, (\mathsf{IM}', \mathsf{S}'), R, G, Q, I)$$

$$\text{where } \Downarrow(\mathsf{IM}, \mathsf{B}, \tau){=}\mathsf{IM}' \overset{\text{def}}{\Leftrightarrow} (\mathsf{IM}, \mathsf{B}) \xrightarrow{\tau:\epsilon}{}^{*} \mathsf{IM}', \mathsf{B}' \wedge \mathsf{B}'(\tau){=}\epsilon$$
$$\text{and } \mathsf{A}_{\mathsf{sp}} \triangleq \left\{\left((\mathsf{IM}, \mathsf{S}), (\mathsf{IM}[\overline{x_{\mathsf{p}} \mapsto \mathsf{IM}(x_{\mathsf{s}})}], \mathsf{S})\right), \left((\mathsf{IM}, \mathsf{S}), (\mathsf{IM}[y_{\mathsf{s}} \mapsto y_{\mathsf{p}}], \mathsf{S})\right) \mid y \in \mathrm{Loc}\right\}^*$$

Recall that a translated program (via $[\![.]\!]$) is of the form $\mathsf{C}^{\mathsf{sp}} \triangleq \mathsf{C} \,||\, \mathsf{c}_{\mathsf{s}} \,||\, \mathsf{c}_{\mathsf{p}}$. A configuration is always safe for zero steps. For $n{+}1$ steps, a configuration is safe if: (1) $(\mathsf{IM}, \mathsf{S})$ is a state in the invariant I; (2) whenever the program has finished execution (i.e. $\mathsf{C}{=}\mathsf{C}_{\mathbf{skip}}$), then $(\mathsf{IM}, \mathsf{S})$ must be a post-state in Q; (3) whenever the environment changes the state according to the rely (in R) or performs a (volatile-to-synchronous or synchronous-to-persistent) propagation (in A_{sp}), then the resulting configuration remains safe for a further n steps; and (4) whenever the thread performs an $\mathrm{Ix86}_{\mathrm{sim}}$ transition, its changes to the state are either those permitted by the guarantee (in G) or those of propagation (in A_{sp}), and the new configuration remains safe for n more steps.

Note that the $\mathrm{Ix86}_{\mathrm{sim}}$ transitions (Fig. 8) are over an instrumented memory and a buffer map. As such, the memory IM in the pre-state of the thread τ describes several storage pairs of the form $(\mathsf{IM}_1, \mathsf{B}_1)$ such that $\Downarrow(\mathsf{IM}_1, \mathsf{B}_1, \tau){=}\mathsf{IM}$. That is, once the pending entries of τ in $\mathsf{B}_1(\tau)$ are propagated to IM_1 (via $\xrightarrow{\tau:\epsilon}$ storage transitions in rule (M-PropW) of Fig. 8), the resulting memory corresponds to IM. Intuitively, IM denotes the *view* of τ of the storage subsystem, in that τ observes its pending writes and flushes in $\mathsf{B}_1(\tau)$, even though they have not yet reached the memory.

We next define *valid POG judgements*, and show that POG triples (Fig. 3) yield valid judgements. Note that the invariant of a triple, Q_{p}, is obtained from the postcondition Q by erasing the values of registers as well as the volatile/synchronous versions, and replacing them with arbitrary values.

Definition 4. A *judgement* $\langle \mathcal{R}; \mathcal{G} \rangle \Vdash \{P\}\ \mathsf{C}\ \{Q\}$ is valid iff for all $n, \sigma \in [\![P]\!]$, $\mathsf{safe}_n([\![\mathsf{C}]\!], \sigma, [\![\mathcal{R}]\!]^{\mathsf{r}}, [\![\mathcal{G}]\!]^{\mathsf{g}}, [\![Q]\!], [\![Q_{\mathsf{p}}]\!])$ holds with $Q_{\mathsf{p}} \triangleq \exists \overline{v_a, v_{\mathsf{v}}, v_{\mathsf{s}}}.\, Q[\overline{v_a/\mathsf{a}}]\,[\overline{v_{\mathsf{v}}/x_{\mathsf{v}}}]\,[\overline{v_{\mathsf{s}}/x_{\mathsf{s}}}]$. A judgement $\mathsf{C}_{\mathsf{rec}} \Vdash \{P\}\mathsf{C}\{Q\}$ is valid iff for all $(\mathsf{IM}, \mathsf{S}) \in [\![P]\!]$, $\mathsf{IM}', \mathsf{S}'$, if $\mathsf{C}_{\mathsf{rec}} \vdash \mathsf{C}, \mathsf{S}, \mathsf{IM}, \mathsf{B}_0 \Rightarrow^* \mathsf{C}_{\mathbf{skip}}, \mathsf{S}', \mathsf{IM}', \mathsf{B}_0$, then $(\mathsf{IM}', \mathsf{S}') \in [\![Q]\!]$.

Theorem 2 (POG soundness). *For all POG triples* $\langle \mathcal{R}; \mathcal{G} \rangle \vdash \{P\}\mathsf{C}\{Q\}$ *and recovery triples* $\mathsf{C}_{\mathsf{rec}} \vdash \{P\}\ \mathsf{C}\{Q\}$ *in Fig. 3, the POG judgements* $\langle \mathcal{R}; \mathcal{G} \rangle \Vdash \{P\}\ \mathsf{C}\ \{Q\}$ *and* $\mathsf{C}_{\mathsf{rec}} \Vdash \{P\}\ \mathsf{C}\ \{Q\}$ *are valid.*

Finally, we establish the following *adequacy* theorem showing that POG is sound with respect to $\mathrm{Px86}_{\mathrm{sim}}$. Note that it is sufficient to consider *closed* triples with RG contexts $\langle \top; \top \rangle$, i.e. whole programs (not run in parallel with another program) under an empty environment.

Theorem 3 (Adequacy). *For all* $\langle \top; \top \rangle \vdash \{P\}\mathsf{C}\{Q\}$, $(\mathsf{IM}, \mathsf{S}) \models P$, *and* $\mathsf{M}, \mathsf{M}', \mathsf{S}'$, *if* IM *and* M *agree (i.e.* $\forall x.\ \mathsf{M}(x){=}\mathsf{IM}(x_{\mathsf{v}}) = \mathsf{IM}(x_{\mathsf{s}}){=}\mathsf{IM}(x_{\mathsf{p}})$*) and starting from* (M, S) *with empty (thread-local and persistent) buffers, program* C *runs to completion under Px86$_{sim}$ and yields* $(\mathsf{M}', \mathsf{S}')$ *and empty buffers, then there exists* IM' *such that* IM' *and* M' *agree and* $(\mathsf{IM}', \mathsf{S}') \models Q$.

7 A TWO-STEP TRANSFORMATION FOR ELIMINATING $\mathbf{flush}_{\mathbf{opt}}$ INSTRUCTIONS

We describe the most common use-case of $\mathbf{flush}_{\mathrm{opt}}$ instructions, *epoch persistency*, where the use of $\mathbf{flush}_{\mathrm{opt}}$ (rather than **flush**) may prove advantageous for performance. We observe that programs using $\mathbf{flush}_{\mathrm{opt}}$ for epoch persistency follow a certain pattern, and describe how we transform such programs to ones that use **flush** instead, without altering their persistency behaviour.

Epoch Persistency using $\mathbf{flush}_{\mathbf{opt}}$. Recall from §2 that $\mathbf{flush}_{\mathrm{opt}}$ instructions provide weaker ordering constraints and can be reordered e.g. with respect to writes on different cache lines. As such, in order to mitigate the weak ordering constraints on $\mathbf{flush}_{\mathrm{opt}}$ and to ensure their execution by a particular program point, they are typically followed by an **sfence**/**mfence**/RMW in program text; see e.g. Fig. 1d. Indeed, Fig. 1d is an example of *epoch persistency*. More concretely, by combining $\mathbf{flush}_{\mathrm{opt}}$ and **sfence**/**mfence**/RMW instructions, one can divide an execution into distinct *epochs*, where the writes in each epoch may persist in an arbitrary order, while the writes of earlier (in program order) epochs persist before those in later epochs. This is illustrated in Fig. 9a, where $x := 1$ and $y := 1$ both persist before $z := 1$, whereas $x := 1$ and $y := 1$ themselves may persist in either order; i.e. the write on x, y persist in the first epoch before the write on z in the second epoch.

Let L and L' denote locations to be persisted in two consecutive epochs; one can then enforce epoch persistency by following the pattern below in three steps:

(1) executing the writes on L and their corresponding $\mathbf{flush}_{\mathrm{opt}}$ for each cache line, provided that each $\mathbf{flush}_{\mathrm{opt}}$ on a cache line X follows the writes in L on X;
(2) executing an *epoch barrier*, namely an **sfence**, **mfence** or RMW; and (Epoch)
(3) executing the writes on L'.

The combination of $\mathbf{flush}_{\mathrm{opt}}$ instructions in (1) and the epoch barrier in (2) ensures that the writes on L in (1) persist before those on L' in (3) (e.g. $L{=}\{x, y\}$ and $L'{=}\{z\}$ in Fig. 9a, assuming that x and y are in distinct cache lines). Note that in step (1) it is sufficient to execute one $\mathbf{flush}_{\mathrm{opt}}$ per *cache line* as $\mathbf{flush}_{\mathrm{opt}}$ persists all (earlier) pending writes on the same cache line. For instance, if $y := 1$ in Fig. 9a is replaced with $x' := 1$ (where x and x' are on the same cache line), step (1) may simply comprise $x := 1; x' := 1; \mathbf{flush}_{\mathrm{opt}}\ x$ (without a separate $\mathbf{flush}_{\mathrm{opt}}\ x'$). This epoch persistency pattern is used by e.g. Raad et al. [2020] to implement several persistent libraries.

Note that replacing each $\mathbf{flush}_{\mathrm{opt}}$ in Fig. 9a with **flush** also achieves epoch persistency, albeit at a finer granularity (one write per epoch). This is because each **flush** is ordered with respect to all writes and thus introduces a new epoch. Using $\mathbf{flush}_{\mathrm{opt}}$ thus admits more than one write per epoch, and may improve performance as it allows the writes in the same epoch to persist in any order.

$\mathsf{c}_1 \triangleq$ $x := 1;$ **flush**$_{\text{opt}}$ $x;$ $y := 1;$ **flush**$_{\text{opt}}$ $y;$ **sfence**; $z := 1;$	$\mathsf{c}_2 \triangleq$ $x := 1;$ $y := 1;$ **flush**$_{\text{opt}}$ $x;$ **flush**$_{\text{opt}}$ $y;$ **sfence**; $z := 1;$	$\mathsf{c}_3 \triangleq$ $x := 1;$ $y := 1;$ **flush** $x;$ **flush** $y;$ **sfence**; $z := 1;$	$\mathsf{c}_i \;\Big\|\; \mathsf{a} := z;$ **if** (a=1) $w := 1;$
(a)	(b)	(c)	(d)
$\lightning : z{=}1 \Rightarrow x{=}y{=}1$	$\lightning : z{=}1 \Rightarrow x{=}y{=}1$	$\lightning : z{=}1 \Rightarrow x{=}y{=}1$	$\lightning : w{=}1 \Rightarrow x{=}y{=}1$

Fig. 9. Epoch persistency using **flush**$_{\text{opt}}$ (a); the program obtained from c_1 by reordering **flush**$_{\text{opt}}x$ (b); the program obtained from c_2 by converting **flush**$_{\text{opt}}$ to **flush** (c); a common concurrent use-case of **flush**$_{\text{opt}}$ (d), where $\mathsf{c}_i \in \{\mathsf{c}_1, \mathsf{c}_2, \mathsf{c}_3\}$. The c_1, c_2, c_3 programs all have the same persistency behaviour (observing the same values upon recovery), and can be used interchangeably in (d) without changing the persistency behaviour.

Two-Step Transformation. Note that as **flush**$_{\text{opt}}$ instructions are not ordered with respect to writes on different cache lines, each **flush**$_{\text{opt}}$ in step (1) can be reordered after the writes on different cache lines *without changing the persistency behaviour*. For instance, **flush**$_{\text{opt}}$ x in c_1 of Fig. 9a can be reordered past the $y := 1$ write to obtain c_2 in Fig. 9b, where c_1 and c_2 have equivalent persistency behaviours. As such, step (1) of the pattern above can further be split as: (1.a) executing the writes on L; (1.b) executing **flush**$_{\text{opt}}$ for each L cache line.

Indeed, this intuition informs the first step of our transformation towards eliminating **flush**$_{\text{opt}}$: given a program of the form $\mathsf{C} \triangleq \mathsf{c}_a; \textbf{flush}_{\text{opt}}\ x; \mathsf{c}_b; \mathsf{c}$ with $x \in X$, if c is the *first* epoch barrier following **flush**$_{\text{opt}}$ x (i.e. c_b contains no **sfence**/**mfence**/RMW), then one can rewrite C as $\mathsf{C}' \triangleq \mathsf{c}_a; \mathsf{c}_b; \textbf{flush}_{\text{opt}}\ x; \mathsf{c}$ without changing its persistency behaviour (i.e. C and C' yield the same values for all locations upon recovery), *provided that* c_b contains no writes on X and no read instructions. Note that $\mathsf{c}_b \triangleq y := 1$ in Fig. 9a simply meets the stipulated provisos. We shortly elaborate on these provisos on c_b in §7.1, and note that it is always possible to achieve epoch persistency in such a way that fulfils these conditions. Note that one can repeatedly apply this transformation to push down all **flush**$_{\text{opt}}$ in step (1) just before the epoch barrier in (2), thus splitting (1) as (1.a) and (1.b).

Next, we observe that once all **flush**$_{\text{opt}}$ have been pushed down before the epoch barrier, one can almost always replace each **flush**$_{\text{opt}}$ with a corresponding **flush** without altering its persistency behaviour. This is thanks to the strong ordering between each **flush**$_{\text{opt}}$ and the subsequent epoch barrier. For instance, c_2 in Fig. 9b can be transformed to c_3 in Fig. 9c, while leaving its persistency behaviour unchanged. This rewriting constitutes the second and last step of our transformation for eliminating **flush**$_{\text{opt}}$ instructions. In §7.2 we elaborate on the only scenarios under which this rewriting may alter the persistency behaviour, and note that such scenarios do not arise realistically.

Finally, note that this two-step transformation can be used to eliminate **flush**$_{\text{opt}}$ instructions in concurrent programs by applying the transformation to each thread containing **flush**$_{\text{opt}}$. This is illustrated in Fig. 9d, where e.g. c_1 in the left thread can be first transformed to c_2 and subsequently to c_3, while preserving the persistency behaviour of the concurrent program at each step.

7.1 Caveats of Reordering flush$_{\text{opt}}$ after Later Instructions (Transformation Step 1)

Recall that in the first step of our transformation we push down a **flush**$_{\text{opt}}$ x instruction with $x \in X$ just before the epoch barrier, provided that there are *no writes on* X and *no reads* between **flush**$_{\text{opt}}$ x and the epoch barrier, as otherwise this transformation alters the persistency behaviour.

In the case of writes on X, consider the example in Fig. 10a where $x, x' \in X$ and the $x' := 1$ write is between **flush**$_{\text{opt}}$ x and **sfence**. Fig. 10b depicts the program obtained from Fig. 10a by reordering **flush**$_{\text{opt}}$ x after $x' := 1$. Recall that executing **flush**$_{\text{opt}}$ x persists *all earlier* writes on X; as such

$x := 1;$ $\mathbf{flush}_{\mathrm{opt}}\ x;$ $x' := 1;$ $\mathbf{sfence};$ $z := 1;$ (a)	$x := 1;$ $x' := 1;$ $\mathbf{flush}_{\mathrm{opt}}\ x;$ $\mathbf{sfence};$ $z := 1;$ (b)	$x := 1;$ $\mathbf{flush}_{\mathrm{opt}}\ x;$ $\mathbf{sfence};$ $x' := 1;$ $z := 1;$ (c)
$\lightning: z{=}1 \Rightarrow x{=}1 \quad z{=}1 \wedge x'{=}0$ ✓	$\lightning: z{=}1 \Rightarrow x{=}x'{=}1 \quad z{=}1 \wedge x'{=}0$ ✗	$\lightning: z{=}1 \Rightarrow x{=}1 \quad z{=}1 \wedge x'{=}0$ ✓

Fig. 10. An epoch persistency example not following the (Epoch) pattern (a); the program obtained from (a) by reordering $\mathbf{flush}_{\mathrm{opt}}\ x$ after $x' := 1$ (where $x, x' \in X$), thus altering its persistency behaviour (b); a program with equivalent persistency behaviour to (a) that follows the (Epoch) pattern (c).

$\mathbf{flush}_{\mathrm{opt}}\ x$ in Fig. 10b is guaranteed to persist both $x := 1$ and $x' := 1$ (i.e. $z{=}1 \Rightarrow x{=}x'{=}1$ upon recovery), while $\mathbf{flush}_{\mathrm{opt}}\ x$ in Fig. 10a is guaranteed to persist only $x := 1$ (i.e. $z{=}1 \Rightarrow x{=}1$ upon recovery). The two programs may therefore have different persistency behaviours: it is possible to observe $z{=}1 \wedge x'{=}0$ after recovery in Fig. 10a but not in Fig. 10b.

Note that Fig. 10a does not adhere to (Epoch): either $x := 1$ and $x' := 1$ are to persist in the same epoch and the program should have been rewritten as in Fig. 10b, or they are to persist in separate epochs in which case the program could have been rewritten as in Fig. 10c. That is, the intended persistency behaviour of Fig. 10a is ambiguous, and it is better practice to rewrite it as either Fig. 10b or Fig. 10c, both of which adhere to (Epoch). We thus argue that it is possible to achieve epoch persistency *without* an intervening write on X between $\mathbf{flush}_{\mathrm{opt}}\ x$ and its epoch barrier. As such, it is possible to apply our first transformation step while preserving the persistency behaviour. Observe that the first transformation step in both Fig. 10b and Fig. 10c is idempotent ($\mathbf{flush}_{\mathrm{opt}}\ x$ is already before **sfence**) and thus trivially preserves the persistency behaviour.

In the case of reads, consider the example in Fig. 11a where the $\mathtt{a} := x$ read is between $\mathbf{flush}_{\mathrm{opt}}\ x$ and **sfence**. Fig. 11b depicts the program obtained from Fig. 11a by reordering $\mathbf{flush}_{\mathrm{opt}}\ x$ after $\mathtt{a} := x$. Note that the right thread executes $y := 1$ only when $\mathtt{a} := x$ reads 2 from x, which in turn implies that $x := 2$ in the left thread is store-ordered after $x := 1$ in the right (otherwise $\mathtt{a} := x$ would read 1 from x). That is, $y{=}1$ implies the following store order in both Fig. 11a and Fig. 11b: $x := 1 \rightarrow x := 2 \rightarrow \mathtt{a} := x$. In Fig. 11b we further have $\mathtt{a} := x \rightarrow \mathbf{flush}_{\mathrm{opt}}\ x$. Put together, this ensures $x := 1 \rightarrow x := 2 \rightarrow \mathbf{flush}_{\mathrm{opt}}\ x$ when $y{=}1$ in Fig. 11b. Consequently, executing $\mathbf{flush}_{\mathrm{opt}}\ x$ first persists $x := 1$ and then persists $x := 2$, as executing $\mathbf{flush}_{\mathrm{opt}}\ x$ persists all pending writes on x in the store order. As such, $z{=}y{=}1 \Rightarrow x{=}2$ upon recovery in Fig. 11b. By contrast, in Fig. 11a the $\mathtt{a} := x \rightarrow \mathbf{flush}_{\mathrm{opt}}\ x$ order does not hold, and thus $\mathbf{flush}_{\mathrm{opt}}\ x$ is only guaranteed to persist $x := 1$, yielding $z{=}y{=}1 \Rightarrow x \in \{1, 2\}$ upon recovery. The two programs may thus have different persistency behaviours: it is possible to observe $z{=}y{=}x{=}1$ after recovery in Fig. 11a but not in Fig. 11b.

Note that once again Fig. 11a does not adhere to (Epoch) and its intended persistency behaviour is ambiguous. More concretely, either: (1) x, y are on different cache lines, in which case the absence of a corresponding $\mathbf{flush}_{\mathrm{opt}}\ y$ implies that $y := 1$ is to persist in the epoch after $x := 1$, and is thus better practice to rewrite the program as in Fig. 11c; or (2) x, y are on the same cache line but $y := 1$ is to persist in the epoch after $x := 1$ and thus as in the previous case it is clearer to rewrite the program as in Fig. 11c; or (3) x, y are on the same cache line and $y := 1$ is to persist in the same epoch as $x := 1$, in which case $\mathbf{flush}_{\mathrm{opt}}\ x$ must follow $y := 1$. That is, in all three cases it is possible to rewrite Fig. 11c such that adheres to (Epoch) and avoids the intervening read between $\mathbf{flush}_{\mathrm{opt}}$ and the epoch barrier. We thus argue that it is possible to achieve epoch persistency *without* an intervening read between a $\mathbf{flush}_{\mathrm{opt}}$ and the epoch barrier, and thus it is often possible to apply our first transformation step while preserving the persistency behaviour. Finally, we prove that the first step of our transformation is sound in that it does not alter the persistency behaviour.

$x := 2;$ || $x := 1;$ $\mathbf{flush}_{\text{opt}}\ x;$ $\mathsf{a} := x;$ **if** $(\mathsf{a}{=}2)\ \ y := 1;$ **sfence**; $z := 1;$	$x := 2;$ || $x := 1;$ $\mathsf{a} := x;$ $\mathbf{flush}_{\text{opt}}\ x;$ **if** $(\mathsf{a}{=}2)\ \ y := 1;$ **sfence**; $z := 1;$	$x := 2;$ || $x := 1;$ $\mathbf{flush}_{\text{opt}}\ x;$ **sfence**; $\mathsf{a} := x;$ **if** $(\mathsf{a}{=}2)\ \ y := 1;$ $z := 1;$
(a)	(b)	(c)
$\lightning: z{=}y{=}1 \Rightarrow x \in \{1,2\}$ $z{=}y{=}x{=}1$✓	$\lightning: z{=}y{=}1 \Rightarrow x{=}2$ $z{=}y{=}x{=}1$✗	$\lightning: z{=}y{=}1 \Rightarrow x \in \{1,2\}$ $z{=}y{=}x{=}1$✓

Fig. 11. An epoch persistency example not following the (Epoch) pattern (a); the program obtained from (a) by reordering $\mathbf{flush}_{\text{opt}}\ x$ after a read ($\mathsf{a} := x$), thus altering its persistency behaviour (b); a program with equivalent persistency behaviour to (a) that follows the (Epoch) pattern (c).

Theorem 4 (Step 1 soundness). *Given* C *with* $\mathsf{C}(\tau){=}\mathsf{c}_a; \mathbf{flush}_{opt}\ x; \mathsf{c}_b; \mathsf{c}$ *and* $x \in X$*, if* c *is the first epoch barrier after* $\mathbf{flush}_{opt}\ x$ *(i.e.* c_b *contains no* **sfence**/**mfence**/*RMW) and* c_b *contains no writes on* X *and no reads, then* C *and* C' *have equivalent persistency behaviours, where* $\mathsf{C}' \triangleq \mathsf{C}[\tau \mapsto \mathsf{c}_a; \mathsf{c}_b; \mathbf{flush}_{opt}\ x; \mathsf{c}]$.

7.2 Caveats of Converting $\mathbf{flush}_{\text{opt}}$ to flush (Transformation Step 2)

Recall that once all $\mathbf{flush}_{\text{opt}}$ instruction have been pushed down just before the epoch barrier, our second transformation step replaces each $\mathbf{flush}_{\text{opt}}$ with a corresponding **flush**. However, in the case of a *blind persist* this transformation may alter the persistency behaviour of the program.

A blind persist denotes a scenario where a persist operation on x is executed without a previous access (read/write) on x. An example of this is illustrated in Fig. 12a, where $\mathbf{flush}_{\text{opt}}\ x$ is issued without any prior access on x (assuming x, y are on different cache lines). Fig. 12b depicts the program obtained from Fig. 12a by replacing $\mathbf{flush}_{\text{opt}}$ with **flush**. Note that the left thread executes $w := 1$ only when $\mathsf{a}{=}2$, which in turn implies that $y := 1$ in the left thread is store-ordered before $y := 2$ in the right (otherwise $\mathsf{a}{=}2$ would not be possible). That is, $w{=}1$ implies the following store order in both Fig. 12a and Fig. 12b: $x := 1 \rightarrow y := 1 \rightarrow y := 2$. As **flush** instructions are ordered with respect to *all* writes, in Fig. 12b we further have $y := 2 \rightarrow \mathbf{flush}\ x$. Put together, this ensures $x := 1 \rightarrow \mathbf{flush}\ x$ when $w{=}1$ in Fig. 12b, and thus executing $\mathbf{flush}\ x$ persists $x := 1$; i.e. $w{=}z{=}1 \Rightarrow x{=}1$ upon recovery in Fig. 12b. By contrast, as $\mathbf{flush}_{\text{opt}}$ instructions may be reordered with respect to writes on different cache lines, in Fig. 12a the $y := 2 \rightarrow \mathbf{flush}_{\text{opt}}\ x$ order does not hold, and thus $\mathbf{flush}_{\text{opt}}\ x$ may not persist $x := 1$, yielding $w{=}z{=}1 \Rightarrow x \in \{0,1\}$ upon recovery. The two programs may thus have different persistency behaviours: it is possible to observe $w{=}z{=}1 \wedge x{=}0$ after recovery in Fig. 12a but not in Fig. 12b.

Note that blind persists are uncommon: persist instructions are expensive and are not typically issued without ascertaining that there is a corresponding write pending to be persisted. More concretely, prior to issuing a $\mathbf{flush}_{\text{opt}}$/**flush** on x, the existence of pending writes on x is usually ascertained by either reading from x, or by having written to x earlier (in the same thread), as shown in Fig. 12c and Fig. 12d, respectively, where $\mathbf{flush}_{(\text{opt})}$ denotes either $\mathbf{flush}_{\text{opt}}$ or **flush**. For instance, recall that in epoch persistency (Epoch) each persist is non-blind as it is preceded by a write on the same cache line. In more realistic scenarios such as (Epoch), Figs. 12c and 12d, $\mathbf{flush}_{\text{opt}}$ instructions can thus be replaced with corresponding **flush** without altering the persistency behaviour.

Finally, we prove that the second transformation step is sound (see the accompanying technical appendix for the definition of blind persists).

Theorem 5 (Step 2 soundness). *Given* C *with* $\mathsf{C}(\tau){=}\mathsf{c}_a; \mathbf{flush}_{opt}\ x; \mathsf{c}; \mathsf{c}_b$*, if* $\mathbf{flush}_{opt}$ *is not blind and* c *is an epoch barrier, then* C *and* $\mathsf{C}' \triangleq \mathsf{C}[\tau \mapsto \mathsf{c}'; \mathbf{flush}\ x; \mathsf{c}]$ *have equivalent persistency behaviours.*

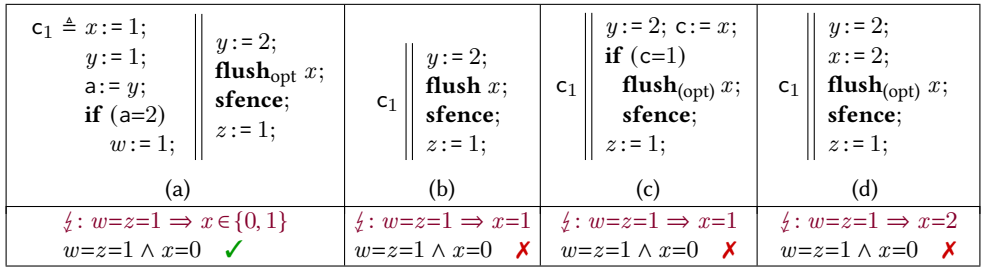

Fig. 12. A blind persist example with $x \in X$, $y \notin X$ (a); the program obtained from (a) by replacing $\mathbf{flush}_{\mathsf{opt}}$ with **flush**, altering its persistency (b); more realistic examples analogous to (a) with non-blind persists, where **flush**/$\mathbf{flush}_{\mathsf{opt}}$ can be used interchangeably (denoted by $\mathbf{flush}_{(\mathsf{opt})}$) without altering the persistency (c, d).

8 CONCLUSIONS, RELATED AND FUTURE WORK

We presented POG, the *first* program logic for reasoning about persistency behaviours under the Px86_{sim} fragment that excludes $\mathbf{flush}_{\mathsf{opt}}$ instructions. We used POG to verify several representative examples. To establish the soundness of POG, we developed an intermediate operational model, Ix86_{sim}, which simplifies Px86_{sim} by forgoing its persistent buffer and modelling its persist orderings by tracking three different versions for each location. We demonstrated that Ix86_{sim} subsumes Px86_{sim} and emulates all its valid behaviours. We then proved that POG is sound with respect to Ix86_{sim} and thus also with respect to Px86_{sim}. As we note below (see future work), Ix86_{sim} is a valuable contribution in its own right, as it facilitates automated verification techniques for persistency behaviours. Finally, in order to extend the reasoning principles of POG to the full Px86_{sim} that also contains $\mathbf{flush}_{\mathsf{opt}}$ instructions, we presented a two-step transformation mechanism that allows us, in most cases, to rewrite a program using $\mathbf{flush}_{\mathsf{opt}}$ instructions, to an equivalent one that uses **flush** instructions instead without altering its persistency behaviour. As such, to reason about a program C that uses $\mathbf{flush}_{\mathsf{opt}}$, one can first use our transformation to rewrite C to a program C′ with equivalent persistency behaviour, and then use POG to reason about C′.

We based POG on the OGRA program logic [Lahav and Vafeiadis 2015]. As such, since OGRA is proved sound for the release-acquire (RA) consistency model and RA is a weaker model than x86-TSO, POG is also sound for RA. Consequently, POG is incomplete for reasoning about x86-TSO in that it cannot be used to prove the absence of behaviours that are admissible under RA but not x86-TSO. For instance, given the program $\mathsf{C} \triangleq (x := 1; y := 1) \,||\, (y := 2; x := 2)$, we cannot use POG to prove that the final states of C exclude those in which $x{=}1 \wedge y{=}2$ holds.

However, we note that almost all existing program logics in the literature, including those for the strong sequential consistency model, are incomplete (e.g. [Dinsdale-Young et al. 2010; Jones 1983; Jung et al. 2015; Kaiser et al. 2017; Lahav and Vafeiadis 2015; Nanevski et al. 2014; Raad et al. 2015; Svendsen et al. 2018; Turon et al. 2014; Vafeiadis and Narayan 2013]) as their main aim to enable simple high-level reasoning principles that apply to common patterns, albeit at the cost of compromising completeness for certain cases. Nevertheless, as shown in the literature, incompleteness can be remedied to some extent by including ghost state [Jacobs and Piessens 2011]. Specifically, it is possible to prove the example above using ghost state, provided that we also show that introducing such state is sound under Ix86_{sim}/x86-TSO, which we leave for future work.

Related work. Although existing literature on NVM has grown rapidly in recent years, formally verifying programs and algorithms that operate on NVM has largely remained unexplored. Friedman et al. [2018] developed several persistent queue implementations using the Intel-x86 persist instructions (e.g. **flush**); Zuriel et al. [2019] also developed two persistent set implementations

using Intel-x86 persist instructions. Both [Friedman et al. 2018; Zuriel et al. 2019] argue that their implementations are correct by providing an informal argument at the level of program traces. Derrick et al. [2019] provided a formal correctness proof of the queue implementation by Friedman et al. [2018]; this proof is also at the level of program traces. Moreover, all three of [Derrick et al. 2019; Friedman et al. 2018; Zuriel et al. 2019] assume that the underlying memory model is sequential consistency (SC) [Lamport 1979], rather than Intel x86-TSO. Raad et al. [2019] recently developed a persistent transactional library on top of the ARM architecture; they later adapted this implementation to the Px86_{sim} architecture. In both cases they provide a formal proof of their implementation correctness on top of the corresponding architecture. Nevertheless, these proofs are low-level in that they operate at the level of execution traces, rather than the program syntax.

To our knowledge, no existing work provides a syntactic proof system for verifying the persistency guarantees of concurrent programs, especially in the presence of relaxed (out-of-order) or asynchronous persists. The most closely related work to ours are those of [Chen et al. 2015; Ntzik et al. 2015]. Chen et al. [2015] present crash Hoare logic (CHL) for reasoning about the crashing behaviour of the FSCQ file system. CHL is more restricted than POG in two ways. First, CHL can only be used for *sequential* programs and does not support concurrent reasoning. Second, in contrast to POG's support for explicit **flush** instructions, CHL does not support the Unix explicit persist instruction **fsync**. Ntzik et al. [2015] extend the Views framework [Dinsdale-Young et al. 2013] to support fault conditions. As an extension of Views, this work supports concurrency; however, their support for persistent reasoning is rather limited: (1) it assumes that the underlying memory model is SC and does not account for weak concurrent behaviours; (2) it does not distinguish between stores and persists, and thus assumes that all stores persist synchronously and in the store order; and consequently (3) does not support any explicit persist instructions such as **flush**.

Future work. We plan to build on our work here in several ways. First, we will use POG to verify existing implementations of persistent libraries and data structures such as [Friedman et al. 2018; Intel 2015; Zuriel et al. 2019]. Second, we will build automated techniques such as model checking (MC) for verifying persistency. We plan to do this by extending existing MC algorithms that already support x86-TSO (e.g. [Abdulla et al. 2015; Kokologiannakis et al. 2019a,b]) with the atomic propagation constructs of Ix86_{sim}. This will allow us to leverage cutting-edge MC tools to verify persistency with minimal implementation overhead. Lastly, building on the ideas underpinning POG, we will devise a similar program logic for reasoning about persistency under the ARMv8 architecture [Raad et al. 2019].

ACKNOWLEDGMENTS

We thank the OOPSLA 2020 reviewers for their valuable feedback. Azalea Raad 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). Ori Lahav was supported by the Israel Science Foundation (grant number 5166651), by Len Blavatnik and the Blavatnik Family foundation, and by the Alon Young Faculty Fellowship.

REFERENCES

Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. 2015. Stateless Model Checking for TSO and PSO. In *Tools and Algorithms for the Construction and Analysis of Systems*, Christel Baier and Cesare Tinelli (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 353–367.

Arm. 2018. ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile (DDI 0487D.a). https://static.docs.arm.com/ddi0487/da/DDI0487D_a_armv8_arm.pdf

Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2015. Using Crash Hoare Logic for Certifying the FSCQ File System. In *Proceedings of the 25th Symposium on Operating Systems Principles*

(Monterey, California) *(SOSP '15)*. ACM, New York, NY, USA, 18–37. https://doi.org/10.1145/2815400.2815402

Edmund Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, Kurt Jensen and Andreas Podelski (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 168–176.

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

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

Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew Parkinson, and Hongseok Yang. 2013. Views: Compositional Reasoning for Concurrent Programs. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Rome, Italy) *(POPL '13)*. ACM, New York, NY, USA, 287–300. https://doi.org/10.1145/2429069.2429104

Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. 2010. Concurrent Abstract Predicates. In *ECOOP 2010 – Object-Oriented Programming*, Theo D'Hondt (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 504–528.

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

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

V. Gogte, W. Wang, S. Diestelhorst, P. M. Chen, S. Narayanasamy, and T. F. Wenisch. 2020. Relaxed Persist Ordering Using Strand Persistency. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 652–665.

Shiyou Huang and Jeff Huang. 2016. Maximal Causality Reduction for TSO and PSO. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Amsterdam, Netherlands) *(OOPSLA 2016)*. ACM, New York, NY, USA, 447–461. https://doi.org/10.1145/2983990.2984025

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

Intel. 2019. 3D XPoint. 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). https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf Order Number: 325462-069US.

Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. 2016. 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.

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

Bart Jacobs and Frank Piessens. 2011. Expressive Modular Fine-Grained Concurrency Specification. *SIGPLAN Not.* 46, 1 (Jan. 2011), 271–282. https://doi.org/10.1145/1925844.1926417

C. B. Jones. 1983. Tentative Steps Toward a Development Method for Interfering Programs. *ACM Trans. Program. Lang. Syst.* 5, 4 (Oct. 1983), 596–619. https://doi.org/10.1145/69575.69577

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

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15)*. Association for Computing Machinery, New York, NY, USA, 637–650. https://doi.org/10.1145/2676726.2676980

J. Kaiser, Hoang-Hai Dang, D. Dreyer, O. Lahav, and Viktor Vafeiadis. 2017. Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris. In *ECOOP*.

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

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

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

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

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

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* (Austin, TX, USA) *(ISCA '09)*. ACM, New York, NY, USA, 2–13. https://doi.org/10.1145/1555754.1555758

Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. 2014. Communicating State Transition Systems for Fine-Grained Concurrent Resources. In *Programming Languages and Systems*, Zhong Shao (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 290–310.

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*.

Gian Ntzik, Pedro da Rocha Pinto, and Philippa Gardner. 2015. Fault-Tolerant Resource Reasoning. In *Programming Languages and Systems*, Xinyu Feng and Sungwoo Park (Eds.). Springer International Publishing, Cham, 169–188.

Susan Owicki and David Gries. 1976. An axiomatic proof technique for parallel programs I. *Acta Informatica* 6, 4 (01 Dec 1976), 319–340. https://doi.org/10.1007/BF00268134

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

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, Jules Villard, and Philippa Gardner. 2015. CoLoSL: Concurrent Local Subjective Logic. In *Proceedings of the 24th European Symposium on Programming (ESOP'15) (Lecture Notes in Computer Science)*, Jan Vitek (Ed.), Vol. 9032. Springer, 710–735.

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 (Jan. 2020), 31 pages. https://doi.org/10.1145/3371079

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

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

Filip Sieczkowski, Kasper Svendsen, Lars Birkedal, and Jean Pichon-Pharabod. 2015. A Separation Logic for Fictional Sequential Consistency. In *Programming Languages and Systems*, Jan Vitek (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 736–761.

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

Kasper Svendsen, Jean Pichon-Pharabod, Marko Doko, Ori Lahav, and Viktor Vafeiadis. 2018. A Separation Logic for a Promising Semantics. In *Programming Languages and Systems*, Amal Ahmed (Ed.). Springer International Publishing, Cham, 357–384.

Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. 2014. GPS: Navigating Weak Memory with Ghosts, Protocols, and Separation. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages*

& Applications (Portland, Oregon, USA) *(OOPSLA '14)*. ACM, New York, NY, USA, 691–707. https://doi.org/10.1145/2660193.2660243

Viktor Vafeiadis and Chinmay Narayan. 2013. Relaxed Separation Logic: A Program Logic for C11 Concurrency. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*. 867–884.

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