View-Based Owicki–Gries Reasoning for Persistent x86-TSO*

Eleni Vafeiadi Bila[1], Brijesh Dongol[1](✉), Ori Lahav[2], Azalea Raad[3], and John Wickerson[3]

[1] University of Surrey, Guildford, UK b.dongol@surrey.ac.uk
[2] Tel Aviv University, Tel Aviv, Israel
[3] Imperial College London, London, UK

Abstract. The rise of persistent memory is disrupting computing to its core. Our work aims to help programmers navigate this brave new world by providing a program logic for reasoning about x86 code that uses low-level operations such as memory accesses and fences, as well as persistency primitives such as flushes. Our logic, Pierogi, benefits from a simple underlying operational semantics based on *views*, is able to handle *optimised* flush operations, and is mechanised in the Isabelle/HOL proof assistant. We detail the proof rules of Pierogi and prove them sound. We also show how Pierogi can be used to reason about a range of challenging single- and multi-threaded persistent programs.

Keywords: Persistent memory, x86-TSO, Owicki-Gries, Isabelle/HOL, verification

1 Introduction

In our era of big data, the long-established boundary between 'memory' and 'storage' is increasingly blurred. Persistent memory is a technology that sits in both camps, promising both the durability of disks and data access times similar to those of DRAM. Embracing this technology requires rethinking our decades-old programming paradigms. As data held in memory is no longer wiped after a system restart, there is an opportunity to write *persistent* programs – programs that can recover their progress and continue computing even after a crash.

However, writing persistent programs is extremely challenging, as it requires the programmer to keep track of which memory writes have become persistent,

* Vafeiadi Bila is supported by VeTSS. Dongol is supported by EPSRC grants EP/V038915/1, EP/R032556/1, EP/R025134/2 and ARC Discovery Grant DP190102142. Lahav is supported by the Israel Science Foundation (grant 1566/18), by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement no. 851811), and by the Alon Young Faculty Fellowship. Raad is supported by a UKRI Future Leaders Fellowship [grant number MR/V024299/1]. Wickerson is supported by an EPSRC Programme Grant (EP/R006865/1).

I. Sergey (Ed.): ESOP 2022, LNCS 13240, pp. 234–261, 2022.
https://doi.org/10.1007/978-3-030-99336-8_9

and which have not. This is further complicated in a multi-threaded setting by the intricate interplay between the rules of memory *persistency* (which determine the order in which writes become persistent) and those of memory *consistency* (which determine what data can be observed by which threads).

To address this difficulty, we provide a foundation for persistent programming. We develop a program logic, Pierogi, for reasoning about x86 code that uses low-level operations such as memory accesses and fences, as well as persistency primitives such as flushes. We demonstrate the utility of Pierogi by using it to reason about a range of challenging single- and multi-threaded persistent programs, including some that demonstrate the subtle interplay between optimised flush ($\mathbf{flush}_{\mathrm{opt}}$) and store fence (**sfence**) instructions. Using the Isabelle/HOL proof assistant, we have mechanised the Pierogi rules and proved them sound with respect to an operational semantics for x86 persistency [9]. One benefit of our Isabelle/HOL formalisation is that Pierogi is already partially automated: once the user has produced a proof outline (i.e. annotated each instruction with a postcondition), they can simply use Isabelle/HOL's *sledgehammer*, which automatically decides which axioms and rules of the proof system need invoking to verify the whole program. Our mechanisation, which includes all the example programs discussed in this paper, is available as auxiliary material [4,5].

State of the art To our knowledge, the only program logic for persistent programs is POG (Persistent Owicki–Gries) [31]. As with Pierogi, POG enables reasoning about persistent x86 programs and is based on the Owicki–Gries method [30]. However, unlike Pierogi, POG is not mechanised in a proof assistant, and does not support optimised flush ($\mathbf{flush}_{\mathrm{opt}}$) instructions. Optimised flush instructions are an important persistency primitive as they are considerably faster than ordinary flush instructions. Indeed, Intel's experiments on their Skylake microarchitecture indicate that they can be *nine times* faster when applied to buffers that hold tens of kilobytes of data [19, p. 289], and hence programmers are impelled, "If $\mathbf{flush}_{\mathrm{opt}}$ is available, use $\mathbf{flush}_{\mathrm{opt}}$ over **flush**." However, $\mathbf{flush}_{\mathrm{opt}}$ is a tricky instruction for programmers and program logic designers alike: compared to **flush**, $\mathbf{flush}_{\mathrm{opt}}$ can be reordered with more instructions under x86.

Pierogi can reason efficiently about x86 persistency (including $\mathbf{flush}_{\mathrm{opt}}$ instructions) thanks to two key recent advances: 1) $\mathrm{Px86}_{\mathrm{view}}$ [9], the view-based operational semantics of x86 persistency; and 2) the C11 Owicki-Gries logic [11–13] to reason about view-based operational semantics, which we adapt to $\mathrm{Px86}_{\mathrm{view}}$.

Our contributions 1) We present a program logic, called Pierogi, for reasoning about persistent x86 programs. 2) We mechanise (and partially automate) Pierogi in Isabelle/HOL, and prove it sound relative to an established operational semantics for x86 persistency. 3) We demonstrate the utility of Pierogi by using it to verify several idiomatic persistent x86 programs.

Outline We begin with an overview of memory consistency and persistency in x86 and provide an example-driven account of Pierogi reasoning (§2). We describe the assertion language and proof rules of Pierogi in §3, and verify a selection of programs using Pierogi in §4. We present the view-based operational semantics of x86 persistency and prove the soundness of Pierogi in §5.

Auxiliary material Additional examples as well as the proofs of theorems stated in the paper are given in the accompanying technical appendix [5]. Our Isabelle/HOL mechanisation is available as auxiliary material [4].

2 Overview and Motivation

Recent operational models for weak memory use *views* to capture relaxed behaviours of concurrent programs [9, 11, 21, 22], where the memory records the entire history of writes that have taken place thus far. This way, different threads can have different subsets of these writes (i.e. different *views*) visible to them. Below, we review Px86$_{\text{view}}$, a view-based operational semantics for x86 persistency (§2.1); we then describe Pierogi (§2.2) using a series of running examples.

2.1 Px86$_{\text{view}}$ at a Glance

In the literature of concurrency semantics, *consistency* models 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, while *persistency* models describe the permitted behaviours of programs upon recovering from a crash (e.g. a power failure) by defining the persistent memory order, i.e. the order in which writes are committed to persistent memory. To distinguish between the two, 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 to persistent memory (durably).

Px86$_{\text{view}}$ Consistency The consistency semantics of Px86$_{\text{view}}$ is that of the well-known TSO (total store ordering) [36] model, where later (in program order) reads can be reordered before earlier writes on different locations. This is illustrated in the *store buffering* (sb) example below (left):

$$\begin{array}{l||l} \textbf{store}\ x\ 1; & \textbf{store}\ y\ 1; \\ a := \textbf{load}\, y & b := \textbf{load}\, x \end{array} \quad (\textsc{sb}) \qquad\qquad \begin{array}{l||l} \textbf{store}\ x\ 42; & a := \textbf{load}\, y; \\ \textbf{store}\ y\ 7 & b := \textbf{load}\, x \end{array} \quad (\textsc{mp})$$

$$a = 0 \land b = 0 : \checkmark \qquad\qquad\qquad a = 7 \land b = 0 : ✗$$

Specifically, assuming $x = y = 0$ initially, since $a := \textbf{load}\, y$ (resp. $b := \textbf{load}\, x$) can be reordered before **store** x 1 (resp. **store** y 1), it is possible to observe the weak behaviour $a = 0 \land b = 0$. A well-known way of modelling such reorderings in TSO is through *store buffers*: when a thread τ executes a write **store** $x\ v$, its effects are not immediately made visible to other threads; rather they are delayed in a thread-local (store) buffer only visible to τ, and propagated to the memory at a later time, whereby they become visible to other threads. For instance, when **store** x 1 and **store** y 1 are delayed in the respective thread buffers (and thus not visible to one another), then $a := \textbf{load}\, y$ and $b := \textbf{load}\, x$ may both read 0.

Cho et al. [9] capture this by associating each thread τ with a *coherence view* (also called a thread-observable view), describing the writes observable by τ. Distinct threads may have different coherence views. For instance, after executing **store** x 1 and **store** y 1, the coherence view of the left thread may include

store x 1 and *not* **store** y 1, while that of the right may include **store** y 1 and *not* **store** x 1. This way, $a := \textbf{load}\, y$ (resp. $b := \textbf{load}\, x$) may read the initial value 0, as its coherence view does not include **store** y 1 (resp. **store** x 1).

After SC (sequential consistency) [27], TSO is one of the strongest consistency models and supports synchronisation patterns such as *message passing*, as shown in MP above, where $a = 7 \wedge b = 0$ cannot be observed. Specifically, (assuming $x = y = 0$ initially) if the right thread reads 7 from y (written by the left thread), then the left thread passes a message to the right. Under TSO, message passing ensures that the instruction writing the message and all those ordered before it (e.g. **store** x 42; **store** y 7) are executed (ordered) before the instruction reading it (e.g. $a := \textbf{load}\, y$). As such, since $b := \textbf{load}\, x$ is executed after $a := \textbf{load}\, y$, if $a = 7$ (i.e. **store** x 42 is executed before $a := \textbf{load}\, y$), then $b = 42$.

Px86 $_{\textit{\textbf{view}}}$ ***Persistency*** Cho et al. [9] recently developed the Px86$_{\text{view}}$ model, a view-based description of the Intel-x86 persistency semantics, which follows a *buffered*, *relaxed* persistency model. Under a buffered model, memory persists occur *asynchronously* [10]: 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 persistency semantics, while allowing the execution to proceed ahead of persists. As such, after recovering from a crash, only a *prefix* of the persistent memory order may have persisted. (The alternative is *unbuffered* persistency in which stores and persists happen simultaneously.)

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 alternative is *strict* persistency in which the volatile and persistent memory orders coincide.)

The relaxed and buffered persistency of Px86$_{\text{view}}$ is shown in Fig. 1a. If a crash occurs during (or after) the execution of Fig. 1a, at crash time either write may have persisted and thus $x, y \in \{0, 1\}$ upon recovery. Note that the two writes cannot be reordered under Intel-x86 (TSO) consistency and thus at no point during the normal (non-crashing) execution of Fig. 1a is $x{=}0, y{=}1$ observable. Nevertheless, in case of a crash it is possible to observe $x{=}0, y{=}1$ after recovery. That is, due to the relaxed persistency of Px86$_{\text{view}}$, the store order (x before y) is separate from the persist order (y before x). More concretely, under Px86$_{\text{view}}$ 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.[4]

To afford more control over when pending writes are persisted, Intel-x86 provides explicit *persist* instructions such as **flush** x and **flush**$_{\text{opt}}$ x that can be used to persist the pending writes on x.[5] This is illustrated in Fig. 1b: executing **flush** x persists the earlier write on x (i.e. **store** x 1) to memory. As such, if

[4] Given a *cache line* (a set of locations), writes on distinct cache lines may persist in any order, while writes on the same cache line persist in the volatile memory order. For brevity, we assume that each cache line contains a single location, thus forgoing the need for cache lines. However, it is straightforward to lift this assumption.

[5] Executing **flush** x or **flush**$_{\text{opt}}$ x persists the pending writes on *all locations in the cache line of* x. However, as discussed, we assume cache lines contain single locations.

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

Fig. 1: Example Px86$_{\text{view}}$ programs and possible values after recovery from a crash (↯). In all examples x, y, z are distinct locations in persistent memory such that $x{=}y{=}z{=}0$ initially, and a is a (thread-local) register.

the execution of Fig. 1b crashes and upon recovery $y{=}1$, then $x{=}1$. That is, if **store** y 1 has executed and persisted before the crash, then so must the earlier **store** x 1; **flush** x. Note that $y{=}1 \Rightarrow x{=}1$ describes a *crash invariant*, in that it holds upon crash recovery *regardless* of when (i.e. at which program point) the crash may have occurred. Observe that this crash invariant is guaranteed thanks to the ordering constraints on **flush** instructions. Specifically, **flush** instructions are ordered with respect to all writes; as such, **flush** x in Fig. 1b cannot be reordered with respect to either write, and thus upon recovery $y{=}1 \Rightarrow x{=}1$.

However, instruction reordering means that persist instructions may not execute at the intended program point and thus not guarantee the intended persist ordering. Specifically, **flush**$_{\text{opt}}$ x is only ordered with respect to earlier writes on x, and may be reordered with respect to later writes, as well as earlier writes on different locations. This is illustrated in Fig. 1c: **flush**$_{\text{opt}}$ x is not ordered with respect to **store** y 1 and may be reordered after it. Therefore, if a crash occurs after **store** y 1 has executed and persisted but before **flush**$_{\text{opt}}$ x has executed, then it is possible to observe $y{=}1, x{=}0$ on recovery. That is, there is no guarantee that **store** x 1 persists before **store** y 1, *despite* the intervening **flush**$_{\text{opt}}$ x.

In order to prevent such reorderings and to strengthen the ordering constraints between **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) **flush**$_{\text{opt}}$, **flush** and write instructions, and can be used to prevent reorderings such as that in Fig. 1c. This is illustrated in Fig. 1d. Unlike in Fig. 1c, the intervening **sfence** ensures that **flush**$_{\text{opt}}$ in Fig. 1d is ordered with respect to **store** y 1 and cannot be reordered after it, ensuring that **store** x 1 persists before **store** y 1 (i.e. $y{=}1 \Rightarrow x{=}1$ upon recovery), as in Fig. 1b. Note that replacing **sfence** in Fig. 1d with **mfence** or an RMW yields the same result. Alternatively, one can think of **flush**$_{\text{opt}}$ x executing *asynchronously*, in that its effect (persisting x) does not take place immediately upon execution, but rather at a later time. However, upon executing a barrier instruction (i.e. **mfence**, **sfence** or an RMW), execution is blocked until the effect of earlier **flush**$_{\text{opt}}$ instructions take place; that is, executing such barrier instructions ensures that earlier **flush**$_{\text{opt}}$ behave *synchronously* (like **flush**).

$P : \{a = b = 0 \land \forall \tau \in \{1, 2\}.\, [x]_\tau = [y]_\tau = \{0\}\}$

$P_1 : \{7 \notin [y]_2 \ \land a = 0\}$	$Q_1 : \{[y]_2 \subseteq \{0, 7\} \land (7 \in [y]_2 \Rightarrow \langle y, 7\rangle[x]_2 = \{42\})\}$
$\textbf{store}\ x\ 42;$ // $\mathsf{SP}_1, \mathsf{Cons}$	$a := \textbf{load}\, y;$ // LP_2
$P_2 : \{[x]_1 = \{42\} \land 7 \notin [y]_2\}$	$Q_2 : \{a \in \{0, 7\} \land (a = 7 \Rightarrow [x]_2 = \{42\})\}$
$\textbf{store}\ y\ 7;$ // $\mathsf{SP}_1, \mathsf{Cons}$	$b := \textbf{load}\, x;$ // $\mathsf{LP}_1, \mathsf{Cons}$
$P_3 : \{\mathsf{true}\}$	$Q_3 : \{a = 7 \Rightarrow b = 42\}$

$Q : \{a = 7 \Rightarrow b = 42\}$

Fig. 2: A Pierogi proof sketch of message passing (mp), where the // annotation at each step identifies the Pierogi proof rule (in §3.4) applied, and the highlighted assertions capture the effects of the preceding instruction.

The example in Fig. 1e illustrates how message passing can impose persist orderings on the writes of *different* threads. (Note that the program in the left thread of Fig. 1e is that of Fig. 1b.) As in mp, if $a = 1$, then **store** x 1; **flush** x is executed before $a := \textbf{load}\, y$ (thanks to message passing). Consequently, since **store** z 1 is executed after $a := \textbf{load}\, y$ when $a = 1$, we know **store** x 1; **flush** x is executed before **store** z 1. Therefore, if upon recovery z=1 (i.e. **store** z 1 has persisted before the crash), then x=1 (**store** x 1; **flush** x must have also persisted before the crash). As before, replacing **flush** x in Fig. 1e with $\textbf{flush}_{\text{opt}}\ x; C$ yields the same result upon recovery when C is an **sfence**/**mfence** or an RMW.

2.2 Pierogi: View-Based Owicki–Gries Reasoning for $\mathbf{Px86_{view}}$

Sequential Reasoning about Consistency using Views In Fig. 2 we present a Pierogi proof sketch of mp. Recall that in order to account for possible write-read reorderings on Intel-x86 architectures, $\text{Px86}_{\text{view}}$ associates each thread τ with a coherence view, describing the writes visible to τ. To reason about such thread-observable views, Pierogi supports assertions of the form $[x]_\tau = S$, stating that τ may read any value in the set S for location x. That is, the coherence view of τ for x consists of the writes whose values are those in S.

In the remainder of this article we enumerate the threads in our examples from left to right; e.g. the left and right threads in Fig. 2 are identified as 1 and 2, respectively. Moreover, we assume the registers of distinct threads have distinct names. The precondition P in Fig. 2 thus states that both threads may initially only read 0 for both x and y: $\forall \tau \in \{1, 2\}.\, [x]_\tau = [y]_\tau = \{0\}$.

In the case of thread 1, we can weaken P (using the standard rule of consequence of Hoare logic – see Cons in §3) to obtain P_1. Upon executing **store** x 42 (1) we weaken the resulting assertion by dropping the $a = 0$ conjunct; and (2) we update the observable view of thread 1 on x to reflect the new value of x: $[x]_1 = \{42\}$; that is, after executing **store** x 42, the only value observable by thread 1 for x is 42. Similarly, after executing **store** y 7, we could assert $[y]_1 = \{7\}$; however, this is not necessary for establishing the final postcondition Q, and we thus simply weaken the postcondition to true (P_3).

```
$\{[y]^{\mathsf{P}} = \{0\}\}$
    store $x$ 1; // $\mathsf{SP}_1$
$\{[x]_1 = \{1\} \land [y]^{\mathsf{P}} = \{0\}\}$
    flush $x$; // $\mathsf{FP}_1$
$\{[x]_1 = \{1\} \land [x]^{\mathsf{P}} = \{1\} \land [y]^{\mathsf{P}} = \{0\}\}$
    store $y$ 1; // $\mathsf{SP}_1$
$\{[x]_1 = \{1\} \land [x]^{\mathsf{P}} = \{1\} \land [y]_1 = \{1\}\}$
$\{\!\{ \lightning : [y]^{\mathsf{P}} = \{1\} \Rightarrow [x]^{\mathsf{P}} = \{1\}\}\!\}$
```

```
$\{[y]^{\mathsf{P}} = \{0\}\}$
    store $x$ 1; // $\mathsf{SP}_1$
$\{[x]_1 = \{1\} \land [y]^{\mathsf{P}} = \{0\}\}$
    flush$_{\text{opt}}$ $x$; // $\mathsf{OP}_1$
$\{[x]_1=\{1\} \land [x]_1^{\mathsf{A}}=\{1\} \land [y]^{\mathsf{P}}=\{0\}\}$
    sfence; // $\mathsf{SFP}_1$
$\{[x]_1=\{1\} \land [x]^{\mathsf{P}}=\{1\} \land [y]^{\mathsf{P}}=\{0\}\}$
    store $y$ 1; // $\mathsf{SP}_1$
$\{[x]_1=\{1\} \land [x]^{\mathsf{P}}=\{1\} \land [y]_1=\{1\}\}$
$\{\!\{ \lightning : [y]^{\mathsf{P}} = \{1\} \Rightarrow [x]^{\mathsf{P}} = \{1\}\}\!\}$
```

Fig. 3: Proof sketches of Fig. 1b (left) and Fig. 1d (right)

Analogously, in the case of thread 2 we weaken P to obtain Q_1: $[y]_2 = \{0\}$ implies $[y]_2 \subseteq \{0, 7\}$ and $7 \in [y]_2 \Rightarrow \langle y, 7\rangle[x]_2 = \{42\}$. Note that $7 \in [y]_2 \Rightarrow \langle y, 7\rangle[x]_2 = \{42\}$ yields a vacuously true implication as $[y]_2 = \{0\}$ and thus $7 \notin [y]_2$. The $\langle y, 7\rangle[x]_2$ denotes a *conditional view assertion* [11] that describes how reading a value on one location (y) affects the thread-observable view on a different location (x). More concretely, $\langle y, 7\rangle[x]_2 = \{42\}$ states that if thread 2 executes a load on y and reads value 7, it subsequently may only observe value 42 for x. This is indeed the essence of message passing in MP: once thread 2 reads 7 from y, it may only read 42 for x thereafter. As such, after executing the read instruction $a := \textbf{load}\, y$ (1) we apply the LP_1 rule (in Fig. 7) which simply replaces $[y]_2$ with the local register a in which the value of y is read; and (2) we replace the conditional assertion $\langle y, 7\rangle[x]_2 = \{42\}$ with the implication $a = 7 \Rightarrow [x]_2 = \{42\}$, stating that if the value read by thread 2 for y (in a) is 7, then its observable view for x is $\{42\}$. Similarly, upon executing $b := \textbf{load}\, x$ we simply apply LP_1 to replace $[x]_2$ with the local register b in which the value of x is read. Lastly, the final postcondition Q is given by the conjunction of the thread-local postconditions ($P_3 \land Q_3$).

Concurrent Reasoning and Stability In our description of the Pierogi proof sketch in Fig. 2 thus far we focused on *sequential* (per-thread) reasoning, ignoring how concurrent threads may affect the validity of assertions at each program point. Specifically, as in existing concurrent logics [11, 26, 30, 31], we must ensure that the assertions at each program point are *stable* under concurrent operations. For instance, to ensure that P_1 remains stable under the concurrent operation $a := \textbf{load}\, y$, we require that executing $a := \textbf{load}\, y$ on states satisfying the conjunction of P_1 and the precondition of $a := \textbf{load}\, y$ (i.e. Q_1) not invalidate P_1, in that the resulting states continue to satisfy P_1; that is, $\{P_1 \land Q_1\} a := \textbf{load}\, y \{P_1\}$ holds. Similarly, we must ensure that P_1 is stable under $b := \textbf{load}\, x$, i.e. $\{P_1 \land Q_2\} b := \textbf{load}\, x \{P_1\}$ holds. Analogously, we must establish the stability of P_2, P_3, Q_1, Q_2 and Q_3 under concurrent operations. In §3 we present syntactic rules that simplify the task of checking stability obligations. It is then straightforward to show that the assertions in Fig. 2 are stable.

Reasoning about* flush *Persistency To reason about the relaxed, buffered persistency of Px86$_{\text{view}}$, Cho et al. [9] introduce *persistency views*, determining the possible *persisted* values for each location; i.e. the values of those writes that may have persisted to memory. Note that the persistency view determines the possible values observable upon recovery from a crash. By contrast, the (per-thread) coherence views determine the observable values during normal (non-crashing) executions, and have no bearing on the post-crash values.

Analogously, we extend PIEROGI with assertions of the form $[x]^{\mathsf{P}} = S$, stating that the persistent view for x includes writes whose values are given by S. To see this, consider the PIEROGI proof sketch of Fig. 1b in Fig. 3 (left). Initially, y holds 0 in persistent memory: $[y]^{\mathsf{P}} = \{0\}$. (Note that the precondition could additionally include $[x]_1 = [y]_1 = \{0\} \wedge [x]^{\mathsf{P}} = \{0\}$ to denote that initially the thread may only observe 0 for x and y and that x holds 0 in persistent memory; however, this is not needed for the proof and we thus forgo it.)

As before, after executing **store** x 1, the observable value for x is updated, as denoted by $[x]_1 = \{1\}$. Moreover, after executing **flush** x, the persisted value for x is 1, as denoted by $[x]^{\mathsf{P}} = \{1\}$, by committing (persisting) the observable value for x (i.e., $[x]_1 = \{1\}$) to memory (see FP_1 in Fig. 7). Finally, after executing **store** y 1, the observable value for y is updated, as denoted by $[y]_1 = \{1\}$.

Crash Invariants Recall that $\lightning$: $y{=}1 \Rightarrow x{=}1$ in Fig. 1b denotes a *crash invariant* in that it describes the persistent memory upon recover from a crash at *any* program point. This is because we have no control over when a crash may occur. To capture such invariants, in PIEROGI we write *quadruples* of the form $\{P\}\ C\ \{Q\}\{\!\{\lightning : I\}\!\}$, where $\{P\}\ C\ \{Q\}$ denotes a Hoare triple and I denotes the crash invariant. If C is a sequential program, I must follow from *every* assertion (including P and Q) in the proof. For instance, in the proof outline of Fig. 3 (left) all four assertions imply the invariant $[y]^{\mathsf{P}} = \{1\} \Rightarrow [x]^{\mathsf{P}} = \{1\}$. We discuss the meaning of crash invariants for concurrent programs below.

Reasoning about* flush$_{\text{opt}}$ *Persistency Recall that unlike **flush**, **flush**$_{\text{opt}}$ instructions (due to instruction reordering) may behave asynchronously and their effects may not take place immediately after execution. As such, unlike for **flush** x, after executing **flush**$_{\text{opt}}$ x we cannot simply copy the observable view on x to the persistent view on x.

To capture the asynchronous nature of **flush**$_{\text{opt}}$, Cho et al. [9] introduce yet another set of views, namely the *thread-local asynchronous view*: the asynchronous view of thread τ on x describes the values (writes) that will be persisted at a later time (asynchronously) by τ upon executing a barrier instruction. That is, 1) when thread τ executes **flush**$_{\text{opt}}$ x, its asynchronous view of x is advanced to at least its observable view of x; and 2) when τ executes a barrier (**sfence**, **mfence** or RMW), then its persistent view for each location is advanced to at least its corresponding asynchronous view. We model this in PIEROGI by 1) setting $[x]^{\mathsf{A}}_{\tau}$ to be a subset of $[x]_{\tau}$ when **flush**$_{\text{opt}}$ x is executed; and 2) setting $[x]^{\mathsf{P}}$ to be a subset of $[x]^{\mathsf{A}}_{\tau}$ (for each location x) when a barrier is executed.

This is illustrated in the proof sketch of Fig. 1d in Fig. 3 (right). In particular, unlike the proof sketch of Fig. 1b in Fig. 3 (left), after executing **flush**$_{\text{opt}}$ x we

```
$P : \{a = 0 \land \forall o \in \{x, y, z\}, \tau \in \{1, 2\}.\, [o]_\tau = [o]^{\mathsf{P}} = \{0\}\}$
$P_1 : \{[y]_2 = \{0\} \land [z]^{\mathsf{P}} = \{0\} \land a = 0\}$                              ||  $\{\mathsf{true}\}$
    store $x$ 1; // $\mathsf{SP}_1$                                                               ||      $a :=$ load $y$;
$P_2 : \{[y]_2 = \{0\} \land [z]^{\mathsf{P}} = \{0\} \land a = 0 \land [x]_1 = \{1\}\}$            ||  $\{\mathsf{true}\}$
    flush $x$; // $\mathsf{FP}_1, \mathsf{Cons}$                                                  ||      if $(a = 1)$
$P_3 : \{[x]^{\mathsf{P}} = \{1\}\}$                                                              ||          $\{a = 1\}$
    store $y$ 1; // $\mathsf{SP}_1, \mathsf{Cons}$                                                ||              store $z$ 1;
$P_4 : \{[x]^{\mathsf{P}} = \{1\}\}$                                                              ||  $\{\mathsf{true}\}$
$Q : \{[x]^{\mathsf{P}} = \{1\}\}$
$I : \{\!\{ ↯ : [z]^{\mathsf{P}} = \{1\} \Rightarrow [x]^{\mathsf{P}} = \{1\}\}\!\}$
```

Fig. 4: A PIEROGI proof sketch of Fig. 1e

cannot simply copy the thread-observable view to the persistent view. Rather, we copy the thread-observable view $[x]_1$ to its asynchronous view and assert $[x]_1^{\mathsf{A}} = \{1\}$; and upon executing the subsequent **sfence**, we copy the thread-asynchronous view to the persistent view and assert $[x]^{\mathsf{P}} = \{1\}$.

Putting It All Together We next present a PIEROGI proof sketch of Fig. 1e in Fig. 4. The proof of the left thread is analogous to that in Fig. 3 (left); the proof of the right thread is straightforward and applies standard reasoning principles. The final postcondition Q is obtained by weakening the conjunction of per-thread postconditions.

Note that the crash invariant I follows from the assertions at each program point of thread 1 (i.e. $P_1 \lor P_2 \lor P_3 \lor P_4 \Rightarrow I$). That is, the crash invariant must follow from the assertions at *all* program points of *some* thread (e.g. thread 1 in Fig. 4). In the case of sequential programs (e.g. in Fig. 3), this amounts to all program points (of the only executing thread). Intuitively, we must ensure that the crash invariant holds at every program point regardless of how the underlying state changes. As the assertions are stable under concurrent operations, it is thus sufficient to ensure that there exists some thread whose assertions at each program point imply the crash invariant.

3 The PIEROGI Proof rules and Reasoning Principles

We proceed with a description of our verification framework. As with prior work [11], the view-based semantics for persistent TSO [9] allows us to use the standard Owicki–Gries rules [2, 30] for compound statements. The main adjustment is the introduction of a new specialised assertion language capable of expressing properties about the different "views" described intuitively in §2. As such, since view updates are highly non-deterministic, the standard "assignment axiom" of Hoare Logic (and by extension Owicki–Gries) is no longer applicable. Moreover, unlike SC, reads in a weak memory setting have a side-effect: their interaction with the memory location being read causes the view of the executing

$v, u \in \textsc{Val} \triangleq \mathbb{N} \qquad x, y, \ldots \in \textsc{Loc} \qquad a, b, \ldots \in \textsc{Reg} \qquad \tau \in \textsc{Tid} \triangleq \mathbb{N} \qquad i, j, k, \ldots \in \textsc{Lab}$

$\hat{a}, \hat{b}, \ldots \in \textsc{AuxVar} \qquad \hat{e} \in \textsc{AuxExp} ::= v \mid \hat{a} \mid \hat{e}{+}\hat{e} \mid \cdots$

$e \in \textsc{Exp} ::= v \mid a \mid e{+}e \mid \cdots \qquad B \in \textsc{BExp} ::= \mathsf{true} \mid B \wedge B \mid \cdots$

$\alpha \in \textsc{ASt} ::= \mathbf{skip} \mid a := e \mid a := \mathbf{load}\, x \mid \mathbf{store}\ x\ e$

$\qquad \mid a := \mathbf{CAS}\ x\ e\ e \mid \mathbf{sfence} \mid \mathbf{mfence} \mid \mathbf{flush}\ x \mid \mathbf{flush}_{\mathrm{opt}}\ x$

$ls \in \textsc{LSt} ::= \alpha\ \mathbf{goto}\ j \mid \mathbf{if}\ B\ \mathbf{goto}\ j\ \mathbf{else\ to}\ k \mid \langle \alpha\ \mathbf{goto}\ j, \hat{a} := \hat{e} \rangle$

$\mathit{\Pi} \in \textsc{Prog} \triangleq \textsc{Tid} \times \textsc{Lab} \rightarrow \textsc{LSt} \qquad \vec{pc} \in \mathrm{PC} \triangleq \textsc{Tid} \rightarrow \textsc{Lab}$

Fig. 5: The Pierogi domains and programming language

thread to advance. Therefore, we resort to a set of proof rules that describe how views are modified and manipulated, as formalised by our view-based assertions.

3.1 The Pierogi Programming Language

We present the programming language in Fig. 5. *Atomic statements* (in ASt) comprise **skip**, assignment, memory reads and writes, barrier instructions and explicit persists. Specifically, $a := e$ evaluates expression e and returns it in (thread-local) register a; $a := \mathbf{load}\, x$ reads from memory location x and returns it in register a; and **store** x e writes the evaluated value of e into location x. The $a := \mathbf{CAS}\ x\ e_1\ e_2$ denotes 'compare-and-set' on location x, from the evaluated value of e_1 to the evaluated value of e_2, and sets a to 1 if the CAS succeeds and to 0, otherwise. Finally, **mfence** denotes a memory fence, **sfence** denotes a store fence, and **flush** x and $\mathbf{flush}_{\mathrm{opt}}$ x denote explicit persist instructions (see §2).

Formally, we model a program $\mathit{\Pi}$ as a function mapping each pair (τ, i) of thread identifier and label to the *labelled statement* (in LSt) to be executed. A labelled statement may be 1) a plain statement of the form α **goto** j, comprising an atomic statement α to be executed and the label j of the next statement; 2) a conditional statement of the form **if** B **goto** j **else to** k to accommodate branching, which proceeds to label j if B holds and to k, otherwise; or 3) a statement with an auxiliary update $\langle \alpha\ \mathbf{goto}\ j, \hat{a} := \hat{e} \rangle$, which behaves as α **goto** j, but in addition (in the same atomic step) updates the value of the auxiliary variable $\hat{a}$ with the auxiliary expression $\hat{e}$. It is well known that Owicki-Gries proofs require auxiliary variables to record the history of executions to differentiate states that would otherwise not be distinguishable [30]. We show how auxiliary variables are used in Pierogi in the flush buffering example (§4).

We track the control flow within each thread via the *program counter function*, $\vec{pc}$, recording the program counter of each thread. We assume a designated label, $\iota \in \textsc{Lab}$, representing the *initial label*; i.e. each thread begins execution with $\vec{pc}(\tau) = \iota$. Similarly, $\zeta \in \textsc{Lab}$ represents the *final label*. Moreover, if $\vec{pc}(\tau) = i$ at the current execution step, then: 1) when $\mathit{\Pi}(\tau, i) = \alpha\ \mathbf{goto}\ j$ or $\mathit{\Pi}(\tau, i) = \langle \alpha\ \mathbf{goto}\ j, a := \hat{e} \rangle$, then $\vec{pc}(\tau) = j$ at the next step; 2) when $\mathit{\Pi}(\tau, i) = \mathbf{if}\ B\ \mathbf{goto}\ j\ \mathbf{else\ to}\ k$ at the current step, then if B holds in the current state, then $\vec{pc}(\tau) = j$ at the next step; otherwise $\vec{pc}(\tau) = k$ at the next step.

Example 1. The program in Fig. 4, assuming that the left thread has id 1, is given as follows. The formalisation of the right thread is omitted, but is similar.

$$\Pi \triangleq \left\{ \begin{array}{l} (1,\iota) \mapsto \textbf{store}\ x\ 1\ \textbf{goto}\ 2, (1,2) \mapsto \textbf{flush}\ x\ \textbf{goto}\ 3, \\ (1,3) \mapsto \textbf{store}\ y\ 1\ \textbf{goto}\ \zeta, ... \end{array} \right\}$$

3.2 View-Based Expressions

As with prior work on the RC11 model [21], we interpret Pierogi expressions directly over a view-based state. We use expressions tailored for the view-based $\text{Px86}_{\text{view}}$ model [9], which allow us to express relationships between different system components, including the persistent memory.

Our expressions fall into one of four categories: 1) *current view* expressions, which describe the current views of different system components (e.g. the persistent view); 2) *conditional view* expressions [11], which describe a view on a location after reading a particular value on a *different* location; 3) *last view* expressions, which hold if a component is viewing the last write to a location; and 4) *write-count* expressions, which describe the number of writes to a location.

Our current view expressions comprise $[x]_\tau$, $[x]^{\mathsf{P}}$ and $[x]^{\mathsf{A}}_\tau$, as described below; as shown in §2, each of these expressions describes a *set* of possible values.

$[x]_\tau$ denotes the *coherence view* of thread τ: the set of values τ may read for x.

$[x]^{\mathsf{P}}$ denotes the *persistent memory view*: the set of values that x may hold in (persistent) memory.

$[x]^{\mathsf{A}}_\tau$ denotes the *asynchronous memory view* of thread τ: the set of values that can be persisted after a barrier instruction (**sfence**/**mfence**/RMW) is executed by τ (see rule OP in Fig. 7). Asynchronous views are updated after executing a $\textbf{flush}_{\text{opt}}$; however, unlike persistent memory views, the values in asynchronous views are not guaranteed to be persisted until a subsequent barrier is executed by the same thread.

Conditional view expressions are of the form $\langle x, v\rangle[y]_\tau$, as described below. As discussed in §2, conditional expressions capture the crux of message passing.

$\langle x, v\rangle[y]_\tau$ returns a set of values that τ may read for y after it reads value v for x. In particular, if $\langle x, v\rangle[y]_\tau = S$ holds for some set S and τ executes $a := \textbf{load}\, x$, then in the state immediately after the load, if $a = v$, then $[y]_\tau \subseteq S$ (see LP_2 in Fig. 7).

Last-view expressions (*cf.* [16]) are boolean-valued and hold if a particular component is synchronised (i.e. observes the latest value) on the given location. Such expressions provide determinism guarantees on **load** and **flush**. For instance if the view of τ is the last write on x, then a read from x by τ will load this last value. Last-view expressions comprise $[\![x]\!]_\tau$ and $[\![x]\!]^{\mathsf{F}}_\tau$:

$[\![x]\!]_\tau$ holds iff τ is currently viewing the *last* write to x. Thus, for example, if $[\![x]\!]_\tau$ holds, then a **load** from x by τ reads the last write to x. Note that unlike architectural operational models [36], in the view model [9], writes are visible to all threads as soon as they occur.

$[\![x]\!]^{\mathsf{F}}_{\tau}$ holds iff a **flush** of x by τ is guaranteed to flush the *last* write to x to persistent memory.

Lastly, write-count expressions are of the form $|x, v|$, as described below. Such assertions are useful for inferring view expressions from known facts about the number of writes in the system with a particular value (see Fig. 11).

$|x, v|$ returns the number of writes to x with value v. If $|x, v|$ holds and τ writes to $y \neq x$, or writes a value $u \neq v$, then $|x, v|$ continues to hold afterwards.

3.3 Owicki–Gries Reasoning

We present the Pierogi proof system, as an extension of Hoare Logic with Owicki–Gries reasoning to account for concurrency. The main differences are that 1) our program annotations contain view-based assertions that allow reasoning about weak and persistent memory behaviours; and 2) we define a crash invariant to describe the recoverable state of the program after a crash. We proceed by first defining proof outlines, then providing syntactic rules for proving their validity. Our proof rules are *syntactic*, and thus can be understood and used without having to understand the details of the underlying $\text{Px86}_{\text{view}}$ model.

We let $\textsc{Assertion}_{\textsc{pv}}$ be the set of *assertions* (i.e. predicates over $\text{Px86}_{\text{view}}$ states) that use view-based expressions (§3.2). A *crash invariant*, $I \in \textsc{Inv} \subset \textsc{Assertion}_{\textsc{pv}}$, is defined over persistent views only, i.e. it only comprises the persistent view expressions of the form $[x]^{\mathsf{P}}$. We model program annotations via an *annotation function*, $ann \in \textsc{Ann} = \textsc{Tid} \times \textsc{Lab} \rightarrow \textsc{Assertion}_{\textsc{pv}}$, associating each program point (τ, i) with its associated assertion. A *proof outline* is a tuple (in, ann, I, fin), where $in, fin \in \textsc{Assertion}_{\textsc{pv}}$ are the initial and final assertions.

Example 2. The annotation of the proof in Fig. 4 is given by ann, with the mappings of thread 1 as shown below; the mappings of thread 2 are similar.

$$ann \triangleq \{(1, \iota) \mapsto P_1, (1, 2) \mapsto P_2, (1, 3) \mapsto P_3, (1, \zeta) \mapsto P_4, \ldots \quad \}$$

Additionally, we have $in \triangleq a = 0 \wedge \forall o \in \{x, y, z\}, \tau \in \{1, 2\}.\, [o]_\tau = [o]^{\mathsf{P}} = \{0\}$, $fin \triangleq [x]^{\mathsf{P}} = \{1\}$ and $I \triangleq [z]^{\mathsf{P}} = \{1\} \Rightarrow [x]^{\mathsf{P}} = \{1\}$.

Definition 1 (Valid proof outline). A proof outline (in, ann, I, fin) is *valid* for a program Π iff the following hold:

Initialisation. For all $\tau \in \textsc{Tid}$, $in \Rightarrow ann(\tau, \iota)$.
Finalisation. $(\bigwedge_{\tau \in \textsc{Tid}} ann(\tau, \zeta)) \Rightarrow fin$.
Local correctness. For all $\tau \in \textsc{Tid}$ and $i \in \textsc{Lab}$, either:

- $\Pi(\tau, i) = \alpha$ **goto** j and $\{ann(\tau, i)\}\ \alpha\ \{ann(\tau, j)\}$; or
- $\Pi(\tau, i) =$ **if** B **goto** j **else to** k and both $ann(\tau, i) \wedge B \Rightarrow ann(\tau, j)$ and $ann(\tau, i) \wedge \neg B \Rightarrow ann(\tau, k)$ hold; or
- $\Pi(\tau, i) = \langle \alpha$ **goto** $j, \hat{a} := \hat{e} \rangle$ and $\{ann(\tau, i)\}\ \alpha\ \{ann(\tau, j)[\hat{e}/\hat{a}]\}$.

Stability. For all $\tau_1, \tau_2 \in \textsc{Tid}$ such that $\tau_1 \neq \tau_2$ and $i_1, i_2 \in \textsc{Lab}$:

- if $\Pi(\tau_1, i_1) = \alpha$ **goto** j, then $\{ann(\tau_2, i_2) \wedge ann(\tau_1, i_1)\}\ \alpha\ \{ann(\tau_2, i_2)\}$;

- if $\Pi(\tau_1, i_1) = \langle \alpha \ \textbf{goto}\ j, \hat{a} := \hat{e} \rangle$, then
  $\{ann(\tau_2, i_2) \wedge ann(\tau_1, i_1)\}\ \alpha\ \{ann(\tau_2, i_2)[\hat{e}/\hat{a}]\}$.

Persistence. There exists $\tau \in$ TID such that for all $i \in$ LAB, $ann(\tau, i) \Rightarrow I$.

Intuitively, Initialisation (resp. Finalisation) ensures that the initial (resp. final) assertion of each thread holds at the beginning (resp. end); Local correctness establishes annotation validity for each thread; Stability ensures that each (local) thread annotation is *interference-free* under the execution of other threads [30]; and Persistence ensures that the crash invariant holds at every program point for some thread.

Example 3. Given the program in Example 1 and its annotation in Example 2, both Initialisation and Finalisation clearly hold. Moreover, Persistence holds for thread 1. For Local correctness of thread 1, we must prove (1)–(3) below; Local correctness of thread 2 is similar.

$$\{P_1\}\ \textbf{store}\ x\ 1\ \{P_2\} \tag{1}$$
$$\{P_2\}\ \ \textbf{flush}\ x\ \ \{P_3\} \tag{2}$$
$$\{P_3\}\ \textbf{store}\ y\ 1\ \{P_4\} \tag{3}$$

For Stability of P (the precondition of **store** x 1 in thread 1) against thread 2 we must prove:

$$\{P_1\}\ a := \textbf{load}\ y\ \{P_1\} \tag{4}$$
$$\{P_1 \wedge a = 1\}\ \ \textbf{store}\ z\ 1\ \ \{P_1\} \tag{5}$$

Stability of other assertions (i.e., P_2–P_4) is similar. We prove (1)–(5) in §3.4.

3.4 PIEROGI Proof rules

One of the main benefits of PIEROGI is the ability to perform proofs at a high level of abstraction. In this section, we provide the set of proof rules that we use. The annotation within a proof outline is, in essence, an invariant mapping each program location to an assertion that holds at the program location. Thus, we prove local correctness by checking that each atomic step of a thread establishes the assertions in that thread. Similarly, we check stability by checking each assertion in one thread against each atomic step of the other threads. To enable proof abstraction, we introduce a set of proof rules that describe the interaction between the assertions from §3.2 and the atomic program steps. We will use the standard decomposition rules from Hoare Logic to reduce proof outlines and enable our rules over atomic steps to be applied.

Standard Decomposition Rules The standard decomposition rules we use are given in Fig. 6, which allow one to weaken preconditions and strengthen postconditions, and decompose conjunctions and disjunctions.

Rules for Atomic Statements and View-Based Assertions Weak and persistent memory models (e.g. Px86) are inherently non-deterministic. Moreover in contrast to sequential consistent, in view-based operational semantics

$$\textsf{Cons}\,\frac{\begin{array}{c}P' \Rightarrow P \quad Q \Rightarrow Q' \\ \{P\}\ \mathit{\Pi}\ \{Q\}\end{array}}{\{P'\}\ \mathit{\Pi}\ \{Q'\}} \qquad \textsf{Conj}\,\frac{\begin{array}{c}\{P_1\}\ \mathit{\Pi}\ \{Q_1\} \\ \{P_2\}\ \mathit{\Pi}\ \{Q_2\}\end{array}}{\{P_1 \wedge P_2\}\ \mathit{\Pi}\ \{Q_1 \wedge Q_2\}} \qquad \textsf{Disj}\,\frac{\begin{array}{c}\{P_1\}\ \mathit{\Pi}\ \{Q_1\} \\ \{P_2\}\ \mathit{\Pi}\ \{Q_2\}\end{array}}{\{P_1 \vee P_2\}\ \mathit{\Pi}\ \{Q_1 \vee Q_2\}}$$

Fig. 6: Standard decomposition rules of Pierogi

Precondition	Statement	Postcondition	Const.	Ref.
$\{[x]_\tau = S\}$	$a := \mathbf{load}\, x$	$\{a \in S \wedge [x]_\tau \subseteq S\}$		LP_1
$\{u \in [x]_\tau \Rightarrow \langle x, u\rangle[y]_\tau = S\}$		$\{a = u \Rightarrow [y]_\tau \subseteq S\}$		LP_2
$\{\lvert x, u\rvert = 1 \wedge \llbracket x \rrbracket_{\tau'} \wedge [x]_{\tau'} = \{u\}\}$		$\{a = u \Rightarrow [x]_\tau = \{u\}\}$		LP_3
$\{true\}$	$\mathbf{store}\ x\ v$	$\{[x]_\tau = \{v\}\}$		SP_1
$\{[x]_{\tau'} = S\}$		$\{[x]_{\tau'} = S \cup \{v\}\}$	$\tau \neq \tau'$	SP_2
$\{[x]^{\mathsf{A}}_{\tau'} = S\}$		$\{[x]^{\mathsf{A}}_{\tau'} = S \cup \{v\}\}$		SP_3
$\{[x]^{\mathsf{P}} = S\}$		$\{[x]^{\mathsf{P}} = S \cup \{v\}\}$		SP_4
$\{[y]_\tau = S \wedge v \notin [x]_{\tau'}\}$		$\{\langle x, v\rangle[y]_{\tau'} \subseteq S\}$	$\tau \neq \tau'$	SP_5
$\{true\}$		$\{\llbracket x \rrbracket_\tau \wedge \llbracket x \rrbracket^{\mathsf{F}}_\tau\}$		SP_6
$\{\lvert x, v\rvert = n\}$		$\{\lvert x, v\rvert = n + 1\}$		SP_7
$\{[x]_\tau = S\}$	$\mathbf{flush}\ x$	$\{[x]^{\mathsf{P}} \subseteq S \wedge [x]^{\mathsf{A}}_\tau \subseteq S\}$		FP_1
$\{[x]^{\mathsf{P}} = S\}$		$\{[x]^{\mathsf{P}} \subseteq S\}$		FP_2
$\{\llbracket x \rrbracket_{\tau'} \wedge [x]_{\tau'} = \{u\} \wedge \llbracket x \rrbracket^{\mathsf{F}}_\tau\}$		$\{[x]^{\mathsf{P}} = \{u\}\}$		FP_3
$\{[x]_\tau = S \vee [x]^{\mathsf{A}}_\tau = S\}$	$\mathbf{flush}_{\mathrm{opt}}\ x$	$\{[x]^{\mathsf{A}}_\tau \subseteq S\}$		OP
$\{[x]^{\mathsf{A}}_\tau = S \vee [x]^{\mathsf{P}} = S\}$	$\mathbf{sfence}$	$\{[x]^{\mathsf{P}} \subseteq S\}$		SFP

Fig. 7: Selected proof rules for atomic statements executed by thread τ

(such as Px86$_{\text{view}}$) instructions such as $a := \mathbf{load}\, x$ have may a side-effect since they may update the view of the thread performing the **load** (*cf.* [11]). Therefore, unlike Hoare Logic, which contains a single rule for assignment, we have a set of rules for atomic statements, describing their interaction with view-based assertions. Each of the rules in this section has been proved sound with respect to the view-based semantics encoded in Isabelle/HOL.

A selection of these rules for the atomic statements is given in Fig. 7, where the statement is assumed to be executed by thread τ. The first column contains the pre/post condition triple, the second any additional constraints and the third, labels that we use to refer to the rules in our descriptions below. Unless explicitly mentioned as a constraint, we do not assume that threads, locations and values are distinct; e.g. rule LP_3 (referring to τ and τ') holds regardless of whether $\tau = \tau'$ or not.

The rules in Fig. 7 provide high-level insights into the low-level semantics of Px86$_{\text{view}}$ without having to understand the operational details. The LP_i rules are for statement $a := \mathbf{load}\, x$. Rule LP_1 states that if τ's view of x is the set of values S, then in the post state a is an element of S and moreover τ's view of x is a subset of S (since τ's view may have shifted). By LP_2, provided the conditional view of τ on y (with condition $x = u$) is S, if the load returns value u, then the view of τ is shifted so that $[y]_\tau \subseteq S$. We only have $[y]_\tau \subseteq S$ in the postcondition because there may be multiple writes to x with value u; reading x

read may shift the view to the latter write, thus *reducing* the set of values that τ can read for y. LP_3 describes conditions for a deterministic load by thread τ. The precondition assumes that there is only one write to x with value u, that *some* thread τ' sees the last write to x with value u. Then, if τ reads u, its view of x is also constrained to just the set containing u.

The store rules, SP_i, reflect that fact that a new write modifies the views of the other threads as well as the persistent memory and asynchronous views. The first four rules describe the interaction of a **store** by thread τ with current view assertions. By SP_1, the **store** ensures that the current view of τ is solely the value v written by τ. This is because in $\mathrm{Px86}_{\mathrm{view}}$, new writes are introduced by the executing thread, τ, with a maximal timestamp (see STORE rule in Fig. 12), and τ's view is updated to this new write. SP_2, SP_3 and SP_4 are similar, and assuming that the view (of another thread, persistent memory and asynchronous view, respectively) in the pre-state is S, shows that the view in the post state is $S \cup \{v\}$. Rule SP_5 allows one to *introduce* a conditional observation assertion $\langle x, v\rangle[y]_{\tau'}$ where $\tau' \neq \tau$. The pre-state of SP_5 assumes that τ's view of y is the set S, and that τ' cannot view value v for y. Rule SP_6 introduces last-view assertions for τ after τ performs a write to x, and finally SP_7 states that the number of writes to x with value v increases by 1 after executing **store** x v.

Rules FP_i describe the effect of **flush** x on the state. FP_1 states that, provided that the current view of τ for x is the set of values S, after executing **flush** x, we are guaranteed that both the persistent view and asynchronous view of τ for x are subsets of S. We obtain a subset in the post state since the $\mathrm{Px86}_{\mathrm{view}}$ semantics potentially moves the persistent and asynchronous views forward. Similarly, by FP_2 if the current persistent view of x is S, then after executing **flush** x the persistent view will be a subset of S. Finally, FP_3 provides a mechanism for establishing a deterministic persistent view u for x. The precondition assumes that *some* thread's view of x is the last write with value u and that τ's view is such that the flush is guaranted to flush to this last write to x.

Rule OP describes how the asynchronous view of τ in the postcondition of $\mathbf{flush}_{\mathrm{opt}}$ x is related to the current view of τ and the asynchronous view in the precondition. Finally, rule SFP describes the relationship between the persistent view in the postcondition and the asynchronous view and persistent view in the precondition for an **sfence** instruction.

Our Isabelle/HOL development contains further rules for the other instructions, including **mfence** and **cas**, which we omit here for space reasons. In addition, we prove the stability of several assertions (see Fig. 8 for a selection). An assertion P is *stable* over a statement α executed by τ iff $\{P\}$ α $\{P\}$ holds.

Well-formedness The final major aspect of our framework is a well-formedness condition that describes the set of reachable states in the $\mathrm{Px86}_{\mathrm{view}}$ semantics. The condition is expressed as an invariant of the semantics: it holds initially, and is stable under every possible transition of $\mathrm{Px86}_{\mathrm{view}}$. In fact, the rules in Figs. 7 and 8 are proved with respect to this well-formedness condition.

The majority of the well-formedness constraints are straightforward, e.g. describing the relationship between the views of different components. The most

Statement	Stable Assert.	Const.	Ref.
$a := \textbf{load}\, x$	$\{[y]_{\tau'} = S\}$	$\tau \neq \tau'$	LS_1
	$\{[y]^{\mathsf{P}} = S\}$		LS_2
	$\{[y]^{\mathsf{A}}_{\tau'} = S\}$		LS_3
	$\{a = k\}$		LS_4
	$\{\llbracket y \rrbracket_{\tau'}\}$		LS_5
flush x	$\{[y]_{\tau'} = S\}$	$x \neq y$	FS_1
	$\{[y]^{\mathsf{P}} = S\}$		FS_2
	$\{\llbracket y \rrbracket_{\tau'}\}$		FS_3
	$\{\llbracket y \rrbracket^{\mathsf{F}}_{\tau'}\}$		FS_4
	$\{\lvert y, v \rvert = n\}$		FS_5
sfence	$\{[x]_{\tau'} = S\}$		SFS_1
	$\{\lvert x, v \rvert = n\}$		SFS_2

Statement	Stable Assert.	Const.	Ref.
store $x\ v$	$\{[y]_{\tau'} = S\}$	$x \neq y$	WS_1
	$\{[y]^{\mathsf{P}} = S\}$	$x \neq y$	WS_2
	$\{[y]^{\mathsf{A}}_{\tau'} = S\}$	$x \neq y$	WS_3
	$\{a = k\}$		WS_4
	$\{\llbracket y \rrbracket_{\tau'}\}$	$x \neq y$	WS_5
	$\{\llbracket y \rrbracket^{\mathsf{F}}_{\tau'}\}$	$x \neq y$	WS_6
	$\{\lvert y, v' \rvert = n\}$	$x \neq y \vee$ $v \neq v'$	WS_7
$\textbf{flush}_{\mathrm{opt}}\ x$	$\{[y]_{\tau'} = S\}$		OS_1
	$\{[y]^{\mathsf{P}} = S\}$		OS_2
	$\{\lvert y, v \rvert = n\}$		OS_3

Fig. 8: Selection of stable assertions for atomic statements executed by thread τ

important component of the well-formedness condition is a non-emptiness condition on views, which states that $[x]_\tau \neq \emptyset \wedge [x]^{\mathsf{P}} \neq \emptyset \wedge [x]^{\mathsf{A}}_\tau \neq \emptyset$. For instance, a consequence of this condition is that, in combination with LP_1, we have:

$$\{[y]_\tau = \{v\}\}\ a := \textbf{load}\, x\ \{[y]_\tau = \{v\}\} \tag{6}$$

Worked Example We now return to the proof obligations from Example 3 and demonstrate how they can be discharged using the proof rules described above. For $\mathsf{Local\ correctness}$, condition (1) holds by Conj (from Fig. 6) together with stability rules WS_1, WS_2 and WS_4 (from Fig. 8) which establish the first three conjunctions in the postcondition, and SP_1 from Fig. 7, which establishes the final conjunction. Condition (2) holds by FP_1 in Fig. 7 together with Cons (from Fig. 6). Finally, condition (3) holds by WS_2 (from Fig. 8).

Both the $\mathsf{Stability}$ conditions (4) and (5) from Example 3 hold by the stability rules in Fig. 8 together with Cons and Conj (from Fig. 6). In particular, for (4), we use rules LS_1, LS_2 and LS_4, and for (5), we use WS_1, WS_2 and WS_4.

4 Examples

In this section we present a selection of programs that we have verified in Isabelle/HOL. These examples highlight specific aspects of Px86, in particular, the interaction between $\textbf{flush}_{\mathrm{opt}}$ and **sfence**, as well as aspects of our view-based assertion language that simplifies verification.

Optimised Message Passing We start by considering a variant of Fig. 1e, which contains two optimisations. First, we notice that flushing of the write to x in thread 1 can be moved to thread 2 since the write to z is guarded by whether or not thread 2 reads the flag y. Second, it is possible to replace the **flush** by a more optimised $\textbf{flush}_{\mathrm{opt}}$ followed by an **sfence**. We confirm correctness of these optimisations via the proof outline in Fig. 9. The optimised message passing in Fig. 9 ensures the same persistent invariant as Fig. 1e. However, the way in

```
$\{\forall o \in \{x, y, z\}, \tau \in \{1, 2\}.\ [o]_\tau = [o]^{\mathsf{P}} = [o]^{\mathsf{A}}_\tau = \{0\}\}$

$\{[y]_2 = \{0\}\}$
store $x$ 1;
$\left\{\begin{array}{l} [y]_2 = \{0\} \wedge \\ [x]_1 = \{1\} \end{array}\right\}$
store $y$ 1;
$\{\mathsf{true}\}$

∥

$\{\ (1 \in [y]_2 \Rightarrow \langle y, 1\rangle[x]_2 = \{1\}) \wedge [y]_2 \subseteq \{0, 1\} \wedge [z]^{\mathsf{P}} = \{0\}\ \}$
$a :=$ load $y$;
$\{(a = 1 \Rightarrow [x]_2 = \{1\}) \wedge [z]^{\mathsf{P}} = \{0\}\}$
if $(a \neq 0)$
    $\{[x]_2 = \{1\} \wedge [z]^{\mathsf{P}} = \{0\}\}$
    flush$_{\mathrm{opt}}$ $x$;
    $\{[x]^{\mathsf{A}}_2 = \{1\} \wedge [z]^{\mathsf{P}} = \{0\}\}$
    sfence;
    $\{[x]^{\mathsf{P}} = \{1\}\}$
    store $z$ 1;
$\{[z]^{\mathsf{P}} = \{0\} \vee [x]^{\mathsf{P}} = \{1\}\}$

$\{[z]^{\mathsf{P}} = \{0\} \vee [x]^{\mathsf{P}} = \{1\}\}$
$\{\!\{ \lightning : [z]^{\mathsf{P}} = \{1\} \Rightarrow [x]^{\mathsf{P}} = \{1\} \}\!\}$
```

Fig. 9: Proof outline for optimised message passing

which this is established differs. In particular, in Fig. 1e, the persistent invariant holds due to thread 1, whereas in Fig. 9 it holds due to thread 2.

With respect to the persistent invariant, the most important sequence of steps takes place in thread 2 if it reads 1 for y. Note that by the conditional view assertion in the precondition of $a :=$ **load** y, thread 2 is guaranteed to read 1 for x after reading 1 for y. Thus, if the test of **if** statement succeeds, then thread 2 must see 1 for x. This view is translated into an asynchronous view after the **flush**$_{\mathrm{opt}}$ is executed, and then to the persistent view after executing **sfence**. Note that until this occurs, we can guarantee that $[z]^{\mathsf{P}} = \{0\}$, which trivially guarantees the persistent invariant.

Flush Buffering Our next example is a variation of store buffering (SB) and is used to highlight how writes by different threads on different locations interact with flushes. Here, thread 1 writes to x and flushes y, while thread 2 writes to y then flushes x.[6] The writes to w and z are used to witness whether the flushes in both threads have occurred. The persistent invariant states that, if both w and z hold 1 in persistent memory, then either x or y has the new value (i.e. 1) in persistent memory. If both threads perform their **flush** operations, then at least one must flush value 1 since a **flush** cannot be reordered with a **store**.

Although simple to state, the proof is non-trivial since it requires careful analysis of the order in which the stores to x and y occur. In the semantics of Cho et al. [9], the **flush** corresponding to the *second* **store** instruction executed synchronises with writes to *all* locations. Thus, for example, if thread 1's store to x is executed after thread 2's store to y, then the subsequent **flush** in thread 1 is guaranteed to flush the new write to y.

The above intuition requires reasoning about the order in which operations occur. To facilitate this, we use auxiliary variables $\hat{a}$ and $\hat{b}$ to record the order in which the writes to x and y occur; $\hat{a} = 1$ iff the write to x occurs before the

[6] Note that the **flush** operations here are analogous to the **load** instructions in SB.

$$\{\forall o \in \{w,x,y,z\}, \tau \in \{1,2\}.\, [o]_\tau = [o]^{\mathsf{P}} = \{0\}\}$$

$$\begin{array}{l||l}
\left\{\begin{array}{l} (\hat{a},\hat{b} = 0,0 \wedge [z]^{\mathsf{P}} = \{0\}) \vee \\ \left(\begin{array}{l} \hat{a},\hat{b} = 0,1 \wedge \llbracket y \rrbracket_2 \wedge \\ {[y]_2} = \{1\} \wedge [w]^{\mathsf{P}} = \{0\} \end{array}\right) \end{array}\right\}
&
\left\{\begin{array}{l} (\hat{a},\hat{b} = 0,0 \wedge [w]^{\mathsf{P}} = \{0\}) \vee \\ \left(\begin{array}{l} \hat{a},\hat{b} = 1,0 \wedge \llbracket x \rrbracket_1 \wedge \\ {[x]_1} = \{1\} \wedge [z]^{\mathsf{P}} = \{0\} \end{array}\right) \end{array}\right\} \\
\langle \mathbf{store}\ x\ 1, \hat{a} := \hat{b}+1 \rangle; & \langle \mathbf{store}\ y\ 1, \hat{b} := \hat{a}+1 \rangle; \\
\left\{\begin{array}{l} \left(\begin{array}{l} \hat{a} = 1 \wedge \hat{b} \in \{0,2\} \wedge \\ ([z]^{\mathsf{P}} = \{0\} \vee [x]^{\mathsf{P}} = \{1\} \end{array}\right) \vee \\ \left(\begin{array}{l} \hat{a},\hat{b} = 2,1 \wedge \llbracket y \rrbracket_2 \wedge \\ {[y]_2} = \{1\} \wedge \llbracket y \rrbracket_1^{\mathsf{F}} \wedge [w]^{\mathsf{P}} = \{0\} \end{array}\right) \end{array}\right\}
&
\left\{\begin{array}{l} \left(\begin{array}{l} \hat{b} = 1 \wedge \hat{a} \in \{0,2\} \wedge \\ ([w]^{\mathsf{P}} = \{0\} \vee [y]^{\mathsf{P}} = \{1\}) \end{array}\right) \vee \\ \left(\begin{array}{l} \hat{a},\hat{b} = 1,2 \wedge \llbracket x \rrbracket_1 \wedge \\ {[x]_1} = \{1\} \wedge \llbracket x \rrbracket_2^{\mathsf{F}} \wedge [z]^{\mathsf{P}} = \{0\} \end{array}\right) \end{array}\right\} \\
\mathbf{flush}\ y; & \mathbf{flush}\ x; \\
\left\{\begin{array}{l} \left(\begin{array}{l} \hat{a} = 1 \wedge \hat{b} \in \{0,2\} \wedge \\ ([z]^{\mathsf{P}} = \{0\} \vee [x]^{\mathsf{P}} = \{1\})) \end{array}\right) \vee \\ (\hat{a},\hat{b} = 2,1 \wedge [y]^{\mathsf{P}} = \{1\}) \end{array}\right\}
&
\left\{\begin{array}{l} \left(\begin{array}{l} \hat{b} = 1 \wedge \hat{a} \in \{0,2\} \wedge \\ ([w]^{\mathsf{P}} = \{0\} \vee [y]^{\mathsf{P}} = \{1\}) \end{array}\right) \vee \\ (\hat{a},\hat{b} = 1,2 \wedge [x]^{\mathsf{P}} = \{1\}) \end{array}\right\} \\
\mathbf{store}\ w\ 1; & \mathbf{store}\ z\ 1; \\
\left\{\begin{array}{l} \left(\begin{array}{l} \hat{a} = 1 \wedge \hat{b} \in \{0,2\} \wedge \\ ([z]^{\mathsf{P}} = \{0\} \vee [x]^{\mathsf{P}} = \{1\}) \end{array}\right) \vee \\ (\hat{a},\hat{b} = 2,1 \wedge [y]^{\mathsf{P}} = \{1\}) \end{array}\right\}
&
\left\{\begin{array}{l} \left(\begin{array}{l} \hat{b} = 1 \wedge \hat{a} \in \{0,2\} \wedge \\ ([w]^{\mathsf{P}} = \{0\} \vee [y]^{\mathsf{P}} = \{1\}) \end{array}\right) \vee \\ (\hat{a},\hat{b} = 1,2 \wedge [x]^{\mathsf{P}} = \{1\}) \end{array}\right\}
\end{array}$$

$$\{\, (\hat{a},\hat{b} = 1,2 \wedge [x]^{\mathsf{P}} = \{1\}) \vee (\hat{a},\hat{b} = 2,1 \wedge [y]^{\mathsf{P}} = \{1\}) \,\}$$

$$\{\!\{ \lightning : [w]^{\mathsf{P}} = \{1\} \wedge [z]^{\mathsf{P}} = \{1\} \Rightarrow [x]^{\mathsf{P}} = \{1\} \vee [y]^{\mathsf{P}} = \{1\} \}\!\}$$

Fig. 10: Proof outline for flush buffering

write to y, and $\hat{a} = 2$ iff the write to x occurs after the write to y. Let us now consider the precondition of **flush** y (the reasoning for **flush** x is symmetric). There are two disjuncts to consider.

- The first disjunct describes the case in which thread 1 executes its store before thread 2. From here, there is a danger that the thread 1 can terminate having flushed 0 for y. However, from this state, thread 2 is guaranteed to flush 1 for x before setting z to 1, satisfying the persistent invariant, as described by the second disjunct of each assertion in thread 2.
- The second disjunct describes the case in which thread 1 executes its store after thread 2. In this case, thread 1 is guaranteed to flush 1 for y, and this fact is captured by the conjunct $\llbracket y \rrbracket_2 \wedge [y]_2 = \{1\} \wedge \llbracket y \rrbracket_1^{\mathsf{F}}$, which ensures that 1) thread 2 sees the last write to y; 2) the only value visible for y to thread 2 is 1; and 3) a flush performed by thread 1 is guaranteed to flush the last write to y. Note that by 1) and 2), we are guaranteed that the last write to y has value 1. We use these three facts to deduce that $[y]^{\mathsf{P}} = \{1\}$ in the second disjunct of the postcondition of **flush** y using rule FP_3.

Epoch Persistency In our next example, we demonstrate how writes of different threads on the same location interact with an optimised flush in the same location, as well as how the ordering of optimised flushes/loads alters the persistency behaviour. The crash invariant of Fig. 11 states that if z and y hold the value 1 in persistent memory then x has the value 2 in persistent memory.

In order for thread 2 to read value 2 for x, the **store** of 2 at x must be performed before the **store** of 1 and $[x]_2 = \{1, 2\}$. Establishing the persistent

$$\{(\forall\tau \in \{1,2\}, o \in \{x,y,z\}.[o]_\tau = [o]^{\mathsf{P}} = \{0\}) \wedge a = 0\}$$

$$\left\{\begin{array}{l} |x,2| = 0 \wedge \\ \left(\begin{array}{l} ([x]_2 = 0 \wedge [x]_1 = 0) \vee \\ ([x]_2 = 1 \wedge [x]_1 = \{0,1\}) \end{array}\right) \end{array}\right\}$$

$\textbf{store}\ x\ 2;$

$$\left\{\begin{array}{l} |x,2| = 1 \wedge \\ \left(\begin{array}{l} \left(\begin{array}{l} \llbracket x \rrbracket_1 \wedge [x]_1 = \{2\} \wedge \\ [x]_2 \subseteq \{1,2\} \end{array}\right) \vee \\ [x]_2 \subseteq \{0,1,2\} \end{array}\right) \end{array}\right\}$$

$\|$

$$\{[y]^{\mathsf{P}} = \{0\} \wedge [z]^{\mathsf{P}} = \{0\} \wedge (|x,2| \in \{0,1\})\}$$

$\textbf{store}\ x\ 1;$

$$\left\{\begin{array}{l} \left([x]_2 = 1 \vee \left(\begin{array}{l} [x]_2 = \{1,2\} \wedge |x,2| = 1 \wedge \\ \llbracket x \rrbracket_1 \wedge [x]_1 = 2 \end{array}\right)\right) \wedge \\ [y]^{\mathsf{P}} = \{0\} \wedge [z]^{\mathsf{P}} = \{0\} \end{array}\right\}$$

$a := \textbf{load}\, x;$

$$\{(a = 2 \Rightarrow [x]_2 = \{2\}) \wedge [y]^{\mathsf{P}} = \{0\} \wedge [z]^{\mathsf{P}} = \{0\}\}$$

$\textbf{flush}_{\text{opt}}\ x;$

$$\{(a = 2 \Rightarrow [x]_2^{\mathsf{A}} = \{2\}) \wedge [y]^{\mathsf{P}} = \{0\} \wedge [z]^{\mathsf{P}} = \{0\}\}$$

$\textbf{if}\ (a = 2)$

$$\{[x]_2^{\mathsf{A}} = \{2\} \wedge [y]^{\mathsf{P}} = \{0\} \wedge [z]^{\mathsf{P}} = \{0\}\}$$

$\quad\textbf{store}\ y\ 1;$

$$\{([x]_2^{\mathsf{A}} = \{2\} \vee [y]^{\mathsf{P}} = \{0\}) \wedge [z]^{\mathsf{P}} = \{0\}\}$$

$\textbf{sfence};$

$$\{[x]^{\mathsf{P}} = \{2\} \vee [y]^{\mathsf{P}} = \{0\}\}$$

$\textbf{store}\ z\ 1;$

$$\{[x]^{\mathsf{P}} = \{2\} \vee [y]^{\mathsf{P}} = \{0\} \vee [z]^{\mathsf{P}} = \{0\}\}$$

$$\{\ [x]^{\mathsf{P}} = \{2\} \vee [y]^{\mathsf{P}} = \{0\} \vee [z]^{\mathsf{P}} = \{0\}\ \}$$

$$\{\!\{ \lightning : [y]^{\mathsf{P}} = \{1\} \wedge [z]^{\mathsf{P}} = \{1\} \Rightarrow [x]^{\mathsf{P}} = \{2\} \}\!\}$$

Fig. 11: Proof outline for epoch persistency

invariant for thread 2 requires reasoning about the view of thread 2 for address x (i.e. $[x]_2$) after the execution of the instruction $a := \textbf{load}\, x$. Notice here that $a := \textbf{load}\, x$ is ordered with respect to the later $\textbf{flush}_{\text{opt}}\ x$ instruction. Consequently, any impact of the execution of the **load** on $[x]_2$, will also affect $[x]_2^{\mathsf{A}}$. Taking into account the ordering of the writes at the address x, we can conclude that if thread 2 reads the value 2, it reads the value of the last write at x. This is expressed with the assertion $\llbracket x \rrbracket_1$ in the precondition of $a := \textbf{load}\, x$, which states that the threads 1's view of x is the last write to x. By rule LP_3, if a thread τ's view of an address x contains only the last write at this address, and the last value written at this address appears only once at the memory, then if a thread τ read this value at x, its view of x (i.e. $[x]_\tau$) is guaranteed to contain only the last written value at x. Consequently, after reading value 2, thread 2's view of x contains only the value 2 (i.e. $[x]_2 = \{2\}$). Execution of $\textbf{flush}_{\text{opt}}\ x$ ensures $[x]_2^{\mathsf{A}}$ (by rule OP). As a result, in the case that the **if** statement succeeds, after the execution of the **sfence** it is guaranteed that the value 2 is persisted at x (i.e. $[x]^{\mathsf{P}} = \{2\}$). In the case that the **if** statement fails, $[y]^{\mathsf{P}} = \{0\}$ must hold, thus the persistent invariant holds trivially.

5 Pierogi Soundness

In this section we present the Px86$_{\text{view}}$ model from [9] (§5.1), formally interpret our assertions as predicates on states of that model (§ 5.2), and establish the soundness of the proposed reasoning technique (§5.3).

(ASSIGN)
$$\frac{\begin{array}{c}\alpha = a := e\\ v = T.\mathsf{regs}(e)\\ T' = T[\mathsf{regs}(a) \mapsto v]\end{array}}{\langle T, M\rangle \xrightarrow{\alpha} \langle T', M\rangle}$$

(STORE)
$$\frac{\begin{array}{c}\alpha = \mathbf{store}\ x\ e\\ v = T.\mathsf{regs}(e)\\ M' = M +\!\!+ [\langle x := v\rangle]\\ T' = T[\mathsf{coh}(x) \mapsto |M|]\end{array}}{\langle T, M\rangle \xrightarrow{\alpha} \langle T', M'\rangle}$$

(LOAD-INTERNAL)
$$\frac{\begin{array}{c}\alpha = a := \mathbf{load}\, x\\ M[t] = \langle x := v\rangle\\ T.\mathsf{coh}(x) = t\\ T' = T[\mathsf{regs}(a) \mapsto v]\end{array}}{\langle T, M\rangle \xrightarrow{\alpha} \langle T', M\rangle}$$

(LOAD-EXTERNAL)
$$\frac{\begin{array}{c}\alpha = a := \mathbf{load}\, x\\ M[t] = \langle x := v\rangle\\ T.\mathsf{coh}(x) < t\\ x \notin M(t..T.\mathsf{v}_{\mathsf{rNew}}]\end{array} \qquad T' = T\begin{bmatrix}\mathsf{regs}(a) \mapsto v,\\ \mathsf{coh}(x) \mapsto t,\\ \mathsf{v}_{\mathsf{rNew}} \mapsto_{\sqcup} t,\\ \mathsf{v}_{\mathsf{pReady}} \mapsto_{\sqcup} t\end{bmatrix}}{\langle T, M\rangle \xrightarrow{\alpha} \langle T', M\rangle}$$

(SFENCE)
$$\frac{\begin{array}{c}\alpha = \mathbf{sfence}\\ T' = T\begin{bmatrix}\mathsf{v}_{\mathsf{pReady}} \mapsto_{\sqcup} T.\mathsf{maxcoh},\\ \mathsf{v}_{\mathsf{pCommit}} \mapsto_{\sqcup} T.\mathsf{v}_{\mathsf{pAsync}}\end{bmatrix}\end{array}}{\langle T, M\rangle \xrightarrow{\alpha} \langle T', M\rangle}$$

(FLUSH)
$$\frac{\begin{array}{c}\alpha = \mathbf{flush}\ x\\ T' = T\begin{bmatrix}\mathsf{v}_{\mathsf{pAsync}}(x) \mapsto_{\sqcup} T.\mathsf{maxcoh},\\ \mathsf{v}_{\mathsf{pCommit}}(x) \mapsto_{\sqcup} T.\mathsf{maxcoh}\end{bmatrix}\end{array}}{\langle T, M\rangle \xrightarrow{\alpha} \langle T', M\rangle}$$

(FLUSHOPT)
$$\frac{\begin{array}{c}\alpha = \mathbf{flush}_{\mathrm{opt}}\ x\\ T' = T[\mathsf{v}_{\mathsf{pAsync}}(x) \mapsto_{\sqcup} T.\mathsf{coh}(x) \sqcup T.\mathsf{v}_{\mathsf{pReady}}]\end{array}}{\langle T, M\rangle \xrightarrow{\alpha} \langle T', M\rangle}$$

(PROGRAM-NORMAL)
$$\frac{\begin{array}{c}\vec{pc}(\tau) = i \qquad \Pi(\tau, i) = \alpha\ \mathbf{goto}\ j\\ \langle \vec{T}(\tau), M\rangle \xrightarrow{\alpha} \langle T', M'\rangle\\ \vec{pc}' = \vec{pc}[\tau \mapsto j] \qquad \vec{T}' = \vec{T}[\tau \mapsto T']\end{array}}{\langle \vec{pc}, \vec{T}, M, G\rangle \Rightarrow_{\Pi} \langle \vec{pc}', \vec{T}', M', G\rangle}$$

(PROGRAM-IF)
$$\frac{\begin{array}{c}\vec{pc}(\tau) = i \qquad \Pi(\tau, i) = \mathbf{if}\ B\ \mathbf{goto}\ j\ \mathbf{else\ to}\ k\\ \vec{pc}' = \vec{pc}\left[\tau \mapsto \begin{cases} j & \vec{T}(\tau).\mathsf{regs}(B) = \mathsf{true}\\ k & \vec{T}(\tau).\mathsf{regs}(B) = \mathsf{false}\end{cases}\right]\end{array}}{\langle \vec{pc}, \vec{T}, M, G\rangle \Rightarrow_{\Pi} \langle \vec{pc}', \vec{T}, M, G\rangle}$$

(PROGRAM-GHOST)
$$\frac{\begin{array}{c}\vec{pc}(\tau) = i \qquad \Pi(\tau, i) = \langle \alpha\ \mathbf{goto}\ j, \hat{a} := \hat{e}\rangle\\ \langle \vec{T}(\tau), M\rangle \xrightarrow{\alpha} \langle T', M'\rangle\\ \vec{pc}' = \vec{pc}[\tau \mapsto j] \qquad \vec{T}' = \vec{T}[\tau \mapsto T'] \qquad G' = G[\hat{a} \mapsto G(\hat{e})]\end{array}}{\langle \vec{pc}, \vec{T}, M, G\rangle \Rightarrow_{\Pi} \langle \vec{pc}', \vec{T}', M', G'\rangle}$$

Fig. 12: Transitions of Px86$_{\text{view}}$ for a program Π

5.1 The Px86$_{\text{view}}$ Model

Like previous view-based models, Px86$_{\text{view}}$ employs a non-standard memory capturing all previously executed writes, alongside so-called "thread views" that track several position(s) of each thread in that history and enforce limitations on the ability of the thread to read from and write to the memory. In addition, the thread views contain the necessary information for determining the possible contents of the non-volatile memory upon a system crash. Formally, Px86$_{\text{view}}$'s memory and thread states are defined as follows.

Definition 2 (Px86$_{\text{view}}$'s memory). A *memory* $M \in \textsc{Memory}$ is a list of *messages*, where each message has the form $\langle x := v\rangle$ for some $x \in \textsc{Loc}$ and $v \in \textsc{Val}$. We use $w.\mathsf{loc}$ and $w.\mathsf{val}$ to refer to the two components of a message

w. We use standard list notations for memories (e.g. $M_1 +\!\!+ M_2$ for appending memories, $[w]$ for a singleton memory, and $|M|$ for the length of M). We refer to indices (starting from 0) in a memory M as *timestamps*, and denote the t'th element of M as $M[t]$. We use $\sqcup$ for obtaining the maximum among timestamps (i.e. $t_1 \sqcup t_2 = \max(t_1, t_2)$), and extend this notation pointwise to functions. We write $x \notin M(t_2..t_1]$ for the condition $\forall t_2 < t \leq t_1.\, M[t].\mathsf{loc} \neq x$.

Definition 3 (Px86$_{\mathsf{view}}$'s thread states). A *thread state* $T \in \textsc{Thread}$ is a record consisting of the following fields: $\mathsf{coh} : \textsc{Loc} \to \mathbb{N}$, $\mathsf{v_{rNew}} : \mathbb{N}$, $\mathsf{v_{pReady}} : \mathbb{N}$, $\mathsf{v_{pAsync}} : \textsc{Loc} \to \mathbb{N}$, and $\mathsf{v_{pCommit}} : \textsc{Loc} \to \mathbb{N}$. We use standard function/record update notation (e.g. $T' = T[\mathsf{coh}(x) \mapsto t]$ denotes the thread state obtained from T be modifying the x entry in the coh component of T to t). In addition, $\mapsto_\sqcup$ is used to incorporate certain timestamps in fields (e.g. $T[\mathsf{v_{rNew}} \mapsto_\sqcup t]$ denotes the thread state obtained from T be modifying the $\mathsf{v_{rNew}}$ component of T to $T.\mathsf{v_{rNew}} \sqcup t$). We denote by $T.\mathsf{maxcoh}$ the maximum among the coherence view timestamps ($T.\mathsf{maxcoh} = \bigsqcup_x T.\mathsf{coh}(x)$).

The two components, together with program counters and the "ghost memory", are combined in Px86$_{\mathrm{view}}$'s machine states as defined next.

Definition 4 (Px86$_{\mathsf{view}}$'s machine states). A *machine state* is a tuple $\sigma = \langle \vec{pc}, \vec{T}, M, G \rangle$ where $\vec{pc} : \textsc{Tid} \to \textsc{Lab}$ is a mapping assigning the next program label to be executed by each thread, $\vec{T} : \textsc{Tid} \to \textsc{Thread}$ is a mapping assigning the current thread state to each thread, $M \in \textsc{Memory}$ is the current memory, and $G : \textsc{AuxVar} \to \textsc{Val}$ is storing the current values of the auxiliary variables. Below we assume that G is extended to expressions $\hat{e} \in \textsc{AuxExp}$ in a standard way. We denote the components of a machine state σ by $\sigma.\vec{pc}$, $\sigma.\vec{T}$, $\sigma.M$, and $\sigma.G$. In addition, we denote by $\sigma.\mathsf{maxpCommit}(x)$ the maximum among the persistency view timestamps for location x ($\sigma.\mathsf{maxpCommit} = \bigsqcup_\tau \sigma.\vec{T}(\tau).\mathsf{v_{pCommit}}(x)$).

The transitions of Px86$_{\mathrm{view}}$ are presented in Fig. 12. These closely follow the model in [9] with minor presentational simplifications. Note, however, that, for simplicity and following [23], we conservatively assume that writes persist atomically at the location granularity (representing, e.g. machine words) rather than at the granularity of the width of a cache line. We refer the interested reader to [9] for a detailed discussion of the transitions rules in Fig. 12.

The above operational definitions naturally induce a notion of a execution (or a "run") of Px86$_{\mathrm{view}}$ on a certain program Π starting from some initial state of the form $\langle \lambda\tau.\, \iota, \vec{T}, M, G \rangle$. A system crash might occur at any point during the execution. Again, following the model of [9], the non-volatile memory (NVM) is not modeled as a concrete part of the state. Instead, the possible contents of the NVM can be inferred from the machine state (specifically from the memory and the $\mathsf{v_{pCommit}}$ views of the different threads), as defined next. This definition is presented as "crash transition" in [9].

Definition 5. A non-volatile memory $NVM : \textsc{Loc} \to \textsc{Val}$ is *possible in a state* σ if for every $x \in \textsc{Loc}$, there exists some t such that $\sigma.M[t] = \langle x := NVM(x) \rangle$ and $x \notin \sigma.M(t..\sigma.\mathsf{maxpCommit}(x)]$.

5.2 The Semantics of Pierogi Assertions

We present the formal definitions of the expressions introduced in §3.2 in terms of Px86$_{\text{view}}$'s machine states.

Current and conditional views When formalising the *current* and *conditional view* expressions, we start with auxiliary functions that return the sets of observable timestamps visible to the components in question, then extract the values in memory corresponding these timestamps. To facilitate this, we define

$$\mathsf{Vals}(M, TS) \triangleq \{M[t].\mathsf{loc} \mid t \in TS\}$$

where $M \in$ Memory and TS is a set of timestamps.

Thread view To define the meaning of the thread view expression, $[x]_\tau$, we use:

$$\mathsf{TS}^{\mathsf{OF}}_\tau(\sigma, x, t) \triangleq \{t' \mid \sigma.M[t'].\mathsf{loc} = x \wedge \sigma.\vec{T}(\tau).\mathsf{coh}(x) \leq t' \wedge x \notin \sigma.M(t'..t]\}$$
$$\mathsf{TS}_\tau(\sigma, x) \triangleq \mathsf{TS}^{\mathsf{OF}}_\tau(\sigma, x, \sigma.\vec{T}(\tau).\mathsf{v_{rNew}})$$

$\mathsf{TS}^{\mathsf{OF}}_\tau(\sigma, x, t)$ returns the set of *timestamps* that are *observable from* timestamp t for thread τ to read for location x in state σ; and $\mathsf{TS}_\tau(\sigma, x)$ returns the set of *timestamps* that are *observable* for τ to read x in σ. Note that after instantiating t to $\sigma.\vec{T}(\tau).\mathsf{v_{rNew}}$ in $\mathsf{TS}^{\mathsf{OF}}_\tau(\sigma, x, t)$, we obtain the premises of the load rules in Fig. 12. Then, $[x]_\tau \triangleq \lambda\sigma.\, \mathsf{Vals}(\sigma.M, \mathsf{TS}_\tau(\sigma, x))$, i.e. is the set of values in $\sigma.M$ corresponding to the timestamps in $\mathsf{TS}_\tau(\sigma, x)$.

Persistent memory view For the persistent memory view expression, $[x]^{\mathsf{P}}$, we use:

$$\mathsf{TS}^{\mathsf{P}}(\sigma, x) = \{t \mid \sigma.M[t].\mathsf{loc} = x \wedge x \notin \sigma.M(t..\sigma.\mathsf{maxpCommit}(x)]\}$$

which returns the set of *timestamps* that are observable to the *persistent memory* for x in σ. Then, $[x]^{\mathsf{P}} \triangleq \lambda\sigma.\, \mathsf{Vals}(\sigma.M, \mathsf{TS}^{\mathsf{P}}(\sigma, x))$. Note that the second conjunct within the definition of $\mathsf{TS}^{\mathsf{P}}(\sigma, x)$ is precisely the condition that links Px86$_{\text{view}}$ states to NVM states (Definition 5). Given this definition, we have:

Proposition 1. *A non-volatile memory NVM* : Loc → Val *is possible in a state σ iff NVM$(x) \in [x]^{\mathsf{P}}(\sigma)$ for every $x \in$* Loc.

Asynchronous memory view To define the meaning of the asynchronous memory view, $[x]^{\mathsf{A}}_\tau$, we use:

$$\mathsf{TS}^{\mathsf{A}}_\tau(\sigma, x) \triangleq \{t \mid \sigma.M[t].\mathsf{loc} = x \wedge x \notin \sigma.M(t..\sigma.\vec{T}(\tau).\mathsf{v_{pAsync}}(x)]\}$$

which returns the timestamps of the asynchronous view of thread τ in location x and state σ. Then, as before, $[x]^{\mathsf{A}}_\tau \triangleq \lambda\sigma.\, \mathsf{Vals}(\sigma.M, \mathsf{TS}^{\mathsf{A}}_\tau(\sigma, x))$.

Conditional view The functions used to define conditional memory view, $\langle x, v\rangle[y]_\tau$, are slightly more sophisticated than those above. We define:

$$\mathsf{TS}^{\mathsf{OV}}_\tau(\sigma, x, v) \triangleq \left\{ t' \;\middle|\; \begin{array}{l} \exists t \in \mathsf{TS}_\tau(\sigma, x).\ \sigma.M[t].\mathsf{val} = v \wedge \\ \qquad t' = \textbf{if } t = \sigma.\vec{T}(\tau).\mathsf{coh}(x) \textbf{ then } \sigma.\vec{T}(\tau).\mathsf{v_{rNew}} \\ \qquad\qquad \textbf{else } t \sqcup \sigma.\vec{T}(\tau).\mathsf{v_{rNew}} \end{array} \right\}$$

$$\mathsf{TS}^{\mathsf{CO}}_\tau(\sigma, x, v, y) \triangleq \bigcup \{\mathsf{TS}^{\mathsf{OF}}_\tau(\sigma, y, t) \mid t \in \mathsf{TS}^{\mathsf{OV}}_\tau(\sigma, x, v)\}$$

where $\mathsf{TS}^{\mathsf{OV}}_{\tau}(\sigma, x, v)$ returns the set of timestamps that τ can observe for x with value v. Assuming t is a timestamp that τ can observe for x, and the value for x at t is v, the corresponding timestamp t' that $\mathsf{TS}^{\mathsf{OV}}_{\tau}(\sigma, x, v)$ returns is $\sigma.\vec{T}(\tau).\mathsf{v}_{\mathsf{rNew}}$ if τ's coherence view for x is t, and the maximum of t and $\sigma.\vec{T}(\tau).\mathsf{v}_{\mathsf{rNew}}$, otherwise. Given this, $\mathsf{TS}^{\mathsf{CO}}_{\tau}(\sigma, x, v, y)$ returns the timestamps that τ can observe for y, from any timestamp $t \in \mathsf{TS}^{\mathsf{OV}}_{\tau}(\sigma, x, v)$. Finally, the set of conditional values is defined by $\langle x, v\rangle[y]_{\tau} \triangleq \lambda\sigma.\, \mathsf{Vals}(\sigma.M, \mathsf{TS}^{\mathsf{CO}}_{\tau}(\sigma, x, v, y))$.

Last view assertions We use the following auxiliary definition:

$$\mathsf{Last}(M, x) \triangleq \bigsqcup\{t \mid M[t].\mathsf{loc} = x\}$$

which returns the timestamp of the last write to x in M. Then, the last view assertions are given by:

- $[\![x]\!]_{\tau} \triangleq \{\sigma \mid \mathsf{TS}_{\tau}(\sigma, x) = \{\mathsf{Last}(\sigma.M, x)\}\}$, i.e. τ's view of x in σ *is* the last write to x in σ.
- $[\![x]\!]^{\mathsf{F}}_{\tau} \triangleq \{\sigma \mid \mathsf{Last}(\sigma.M, x) \leq \sigma.\vec{T}(\tau).\mathsf{maxcoh} \sqcup \sigma.\mathsf{maxpCommit}(x)\}$, i.e. the maximum of τ's maximum coherence view and the maximum commit view of x (over all threads) is beyond the last write to x in σ. This means that executing a **flush** x operation in τ will cause the last write of x to be flushed (see FLUSH rule in Fig. 12).

Value count Finally, the value count expression is defined as follows:

$$|x, v| \triangleq \lambda\sigma.\, |\{t \mid \sigma.M[t] = \langle x := v\rangle\}|$$

5.3 Soundness of PIEROGI

Given the above building blocks, the soundness of the proposed reasoning technique is stated as follows.

Theorem 1 (Soundness of PIEROGI). *Suppose that a program Π has a valid proof outline $\langle in, ann, I, fin\rangle$. Let σ be a state of Px86$_{view}$ that is reachable in an execution of Π from some state σ_{init} of the form $\langle \lambda\tau.\, \iota, \vec{T}_{\mathsf{init}}, M_{\mathsf{init}}, G_{\mathsf{init}}\rangle$ such that $\sigma_{\mathsf{init}} \in in$. Then, the following hold:*

1) *For every $\tau \in$ TID, we have that $\sigma \in ann(\tau, \sigma.\vec{pc}(\tau))$.*
2) *If $\sigma.\vec{pc}(\tau) = \zeta$ for every $\tau \in$ TID, then $\sigma \in fin$.*
3) *Every non-volatile memory NVM that is possible in σ satisfies the crash invariant I.*

Finally, it is straightforward to show the soundness of a standard "auxiliary variable transformation" [30] which removes all auxiliary variables from a program Π (translating each command $\langle \alpha\ \textbf{goto}\ j, \hat{a} := \hat{e}\rangle$ into $\alpha\ \textbf{goto}\ j$) provided that the crash invariant and the final assertion do not contain occurrences of the auxiliary variables. Indeed, it is easy to see that the auxiliary memory G in the operational semantics in Fig. 12 serves only as an instrumentation, and does not restrict the possible runs. (Formally, if Π' is obtained from Π by removing all auxiliary variables and $\langle \vec{pc}, \vec{T}, M, G'\rangle$ is reachable in $\Rightarrow_{\Pi'}$ from some initial state, then $\langle \vec{pc}, \vec{T}, M, G\rangle$ is reachable in $\Rightarrow_{\Pi}$ from the same state for some G.)

6 Mechanisation

Perhaps the greatest strength of our development is an integrated Isabelle/HOL mechanisation providing a fully fledged semi-automated verification tool for $\text{Px86}_{\text{view}}$ programs. This mechanisation builds on the existing work on Owicki–Gries for RC11 by Dalvandi et al [11,12] applying it to the $\text{Px86}_{\text{view}}$ semantics. We start by encoding the operational semantics of Cho et al. [9], followed by the view-based assertions described in §3.2. Then, we prove correctness of all of the proof rules for the atomic statements, including those described in §3.4. These rules can be challenging to prove since they require unfolding of the assertions and examination of the low-level operational semantics and their effect on the views of different system components.

Once proved, the rules provided are highly reusable, and are key to making verification feasible. Specifically, when showing the validity of a proof outline (Definition 1), Isabelle/HOL generates the necessary proof obligations (after minor interactions) and *automatically* finds the set of high-level proof rules needed to discharge each proof obligation via the built-in sledgehammer tool [6]. This enables a high degree of experimentation and debugging of proof outlines, including the ability to reduce assertion complexity once a proof outline is validated.

The base development (semantics, view-based assertions, and soundness of proof rules) comprise ∼7000 lines of Isabelle/HOL code. With this base development in place, each example comprises 200–400 lines of code (including the encoding of the program, the annotations, and the proofs of validity). The entire development took approximately 3 months of full-time work.

7 Related Work

The soundness of Pierogi is proven relative to the $\text{Px86}_{\text{view}}$ of Cho et al. [9]; there are however other equivalent models in the literature [1,23,32,34], as well as other persistency models [33,35]. While the original persistent x86 semantics has asynchronous explicit persist instructions [34], the underlying model assumed here is due to Cho et al. [9] with synchronous persist instructions. Nevertheless, Khyzha and Lahav [23] formally proved that the two alternatives are equivalent when reasoning about states after crashes (e.g. using our "crash invariants").

As mentioned in §1, the only existing program logic for persistent programs is POG [31], which (as with Pierogi) is a descendent of Owicki–Gries [30]. Pierogi goes beyond POG by handling examples that involve $\textbf{flush}_{\text{opt}}$ instructions, which cannot be directly verified using POG. Raad et al. [31] provide a transformation technique to replace certain patterns of $\textbf{flush}_{\text{opt}}$ and **sfence** with **flush**. Specifically, given a program Π that includes $\textbf{flush}_{\text{opt}}$ instructions, provided that Π meets certain conditions, this transformation mechanism rewrites Π into an equivalent program Π' that uses **flush** instructions instead, allowing one to use POG. However, there are three limitations to this strategy: 1) the rewriting is an external mechanism that requires stepping outside the POG logic; 2) the rewriting is potentially expensive and must be done for every program that includes **flush**$_{\text{opt}}$; and 3) the transformation technique is incomplete in that not all programs meet the stipulated conditions (e.g. Epoch Persistency 2), and thus cannot be verified using this technique. PIEROGI has no such limitations, as we showed in the examples in Section 4. Moreover, POG has no corresponding mechanisation, and developing a mechanisation that also efficiently handles the program transformation for **flush**$_{\text{opt}}$ instructions would be non-trivial.

The Owicki–Gries method was first applied to non-SC memory consistency by Lahav et al. [26]. One way that their approach, which targets the release/acquire memory model, is different from ours is that they aim to use standard SC-like assertions; in order to retain soundness under a weak memory model, they had to strengthen the standard stability conditions on proof outlines. Dalvandi et al. [11, 13] took a different approach when designing their Owicki–Gries logic for the release/acquire fragment of C11: by employing a more expressive, view-based assertion language, they were able to stick with the standard stability requirement. In our work, we follow Dalvandi et al.'s approach. However, our assertions are fine-tuned to cope with the other types of view present in Px86$_{\text{view}}$, such as those corresponding to the persistent and the asynchronous views. It is interesting that some of the principles of view-based reasoning apply to different memory models, and future work could look at unifying reasoning across models.

Dalvandi et al. [13] have developed a deeper integration of their view-based logic using the Owicki–Gries encoding of Nipkow and Prensa Nieto [28] in Isabelle/HOL. Such an integration would be straightforward for PIEROGI too, allowing verification to take place without translating programs into a transition system. This would be much more difficult for POG since Owicki–Gries rules themselves are different from the standard encoding in Isabelle/HOL, in addition to the transformation required for **flush**$_{\text{opt}}$ instructions discussed above.

The idea of extending Hoare triples with crash conditions first appeared in the work of Chen et al. [8]. However, that work supports neither concurrency nor explicit flushing instructions. Related ideas are found in the works of Ntzik et al. [29] and Chajed et al. [7]. However, in contrast to PIEROGI, both of these works 1) assume sequentially consistent memory, as opposed to a weak memory model such as TSO; 2) assume strict persistency (where store and persist orders coincide); and 3) assume there is a synchronous **flush** operation, which is easier to reason about than the asynchronous **flush**$_{\text{opt}}$ operation.

Besides program logics, there have been other recent efforts to help programmers reason about persistent programs. For instance, Abdulla et al. [1] have proven that state-reachability for persistent x86 is decidable, thus opening the door to automatic verification of persistent programs, and Gorjiara et al. [18] and Kokologiannakis et al. [25] have developed model checkers for finding bugs in persistent programs. Recent works have considered durable atomic objects such as concurrent data structures [17] and transactional memory [3] and their verification [3, 14, 15], which have been designed to satisfy conditions such as durable linearizability [20, 24] and durable opacity [3]. These proofs assume persistency under SC; our work provides foundations for extending these proofs to persistent x86-TSO.

References

1. Abdulla, P.A., Atig, M.F., Bouajjani, A., Kumar, K.N., Saivasan, P.: Deciding reachability under persistent x86-TSO. Proc. ACM Program. Lang. **5**(POPL), 1–32 (2021). https://doi.org/10.1145/3434337
2. Apt, K.R., de Boer, F.S., Olderog, E.: Verification of Sequential and Concurrent Programs. Texts in Computer Science, Springer (2009). https://doi.org/10.1007/978-1-84882-745-5
3. Bila, E., Doherty, S., Dongol, B., Derrick, J., Schellhorn, G., Wehrheim, H.: Defining and verifying durable opacity: Correctness for persistent software transactional memory. In: Gotsman, A., Sokolova, A. (eds.) FORTE. Lecture Notes in Computer Science, vol. 12136, pp. 39–58. Springer (2020). https://doi.org/10.1007/978-3-030-50086-3_3
4. Bila, E.V., Dongol, B., Lahav, O., Raad, A., Wickerson, J.: Isabelle/HOL files for "View-Based Owicki-Gries Reasoning for Persistent x86-TSO" (Jan 2022). https://doi.org/10.6084/m9.figshare.18469103
5. Bila, E.V., Dongol, B., Lahav, O., Raad, A., Wickerson, J.: View-based Owicki-Gries reasoning for persistent x86-TSO (extended version) (2022), https://arxiv.org/abs/2201.05860
6. Böhme, S., Nipkow, T.: Sledgehammer: Judgement day. In: Giesl, J., Hähnle, R. (eds.) Automated Reasoning, 5th International Joint Conference, IJCAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings. LNCS, vol. 6173, pp. 107–121. Springer (2010). https://doi.org/10.1007/978-3-642-14203-1_9
7. Chajed, T., Tassarotti, J., Kaashoek, M.F., Zeldovich, N.: Verifying concurrent, crash-safe systems with perennial. In: Brecht, T., Williamson, C. (eds.) Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019. pp. 243–258. ACM (2019). https://doi.org/10.1145/3341301.3359632
8. Chen, H., Ziegler, D., Chajed, T., Chlipala, A., Kaashoek, M.F., Zeldovich, N.: Using crash hoare logic for certifying the FSCQ file system. In: Miller, E.L., Hand, S. (eds.) Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015. pp. 18–37. ACM (2015). https://doi.org/10.1145/2815400.2815402
9. Cho, K., Lee, S.H., Raad, A., Kang, J.: Revamping hardware persistency models: view-based and axiomatic persistency models for Intel-x86 and Armv8. In: Freund, S.N., Yahav, E. (eds.) PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021. pp. 16–31. ACM (2021). https://doi.org/10.1145/3453483.3454027
10. Condit, J., Nightingale, E.B., Frost, C., Ipek, E., Lee, B., Burger, D., Coetzee, D.: Better I/O through byte-addressable, persistent memory. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles. pp. 133–146. SOSP '09, ACM, New York, NY, USA (2009). https://doi.org/10.1145/1629575.1629589
11. Dalvandi, S., Doherty, S., Dongol, B., Wehrheim, H.: Owicki-Gries reasoning for C11 RAR. In: Hirschfeld, R., Pape, T. (eds.) 34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference). LIPIcs, vol. 166, pp. 11:1–11:26. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020). https://doi.org/10.4230/LIPIcs.ECOOP.2020.11
12. Dalvandi, S., Doherty, S., Dongol, B., Wehrheim, H.: Owicki-Gries reasoning for C11 RAR (artifact). Dagstuhl Artifacts Ser. **6**(2), 15:1–15:2 (2020). https://doi.org/10.4230/DARTS.6.2.15

13. Dalvandi, S., Dongol, B., Doherty, S., Wehrheim, H.: Integrating Owicki-Gries for C11-style memory models into Isabelle/HOL. J. Autom. Reason. **66**(1), 141–171 (2022). https://doi.org/10.1007/s10817-021-09610-2
14. Derrick, J., Doherty, S., Dongol, B., Schellhorn, G., Wehrheim, H.: Verifying correctness of persistent concurrent data structures. In: ter Beek, M.H., McIver, A., Oliveira, J.N. (eds.) Formal Methods - The Next 30 Years - Third World Congress, FM 2019, Porto, Portugal, October 7-11, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11800, pp. 179–195. Springer (2019). https://doi.org/10.1007/978-3-030-30942-8_12
15. Derrick, J., Doherty, S., Dongol, B., Schellhorn, G., Wehrheim, H.: Verifying correctness of persistent concurrent data structures: a sound and complete method. Formal Aspects Comput. **33**(4-5), 547–573 (2021). https://doi.org/10.1007/s00165-021-00541-8
16. Doherty, S., Dongol, B., Wehrheim, H., Derrick, J.: Verifying C11 programs operationally. In: Hollingsworth, J.K., Keidar, I. (eds.) Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2019, Washington, DC, USA, February 16-20, 2019. pp. 355–365. ACM (2019). https://doi.org/10.1145/3293883.3295702
17. Friedman, M., Herlihy, M., Marathe, V.J., Petrank, E.: A persistent lock-free queue for non-volatile memory. In: Krall, A., Gross, T.R. (eds.) Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2018, Vienna, Austria, February 24-28, 2018. pp. 28–40. ACM (2018). https://doi.org/10.1145/3178487.3178490
18. Gorjiara, H., Xu, G.H., Demsky, B.: Jaaru: efficiently model checking persistent memory programs. In: Sherwood, T., Berger, E.D., Kozyrakis, C. (eds.) ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021. pp. 415–428. ACM (2021). https://doi.org/10.1145/3445814.3446735
19. Intel Corporation: Intel 64 and IA-32 Architectures Optimization Reference Manual (2021), https://software.intel.com/content/dam/develop/external/us/en/documents-tps/64-ia-32-architectures-optimization-manual.pdf
20. Izraelevitz, J., Mendes, H., Scott, M.L.: Linearizability of persistent memory objects under a full-system-crash failure model. In: Gavoille, C., Ilcinkas, D. (eds.) Distributed Computing - 30th International Symposium, DISC 2016, Paris, France, September 27-29, 2016. Proceedings. Lecture Notes in Computer Science, vol. 9888, pp. 313–327. Springer (2016). https://doi.org/10.1007/978-3-662-53426-7_23
21. Kaiser, J., Dang, H.H., Dreyer, D., Lahav, O., Vafeiadis, V.: Strong logic for weak memory: Reasoning about release-acquire consistency in Iris. In: ECOOP (2017)
22. Kang, J., Hur, C., Lahav, O., Vafeiadis, V., Dreyer, D.: A promising semantics for relaxed-memory concurrency. In: Castagna, G., Gordon, A.D. (eds.) Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017. pp. 175–189. ACM (2017). https://doi.org/10.1145/3009837.3009850
23. Khyzha, A., Lahav, O.: Taming x86-TSO persistency. Proc. ACM Program. Lang. **5**(POPL), 1–29 (2021). https://doi.org/10.1145/3434328
24. Khyzha, A., Lahav, O.: Abstraction for crash-resilient objects. In: Programming Languages and Systems. Springer International Publishing, Cham (2022)
25. Kokologiannakis, M., Kaysin, I., Raad, A., Vafeiadis, V.: Persevere: Persistency semantics for verification under ext4. Proc. ACM Program. Lang. **5**(POPL) (jan 2021). https://doi.org/10.1145/3434324

26. Lahav, O., Vafeiadis, V.: Owicki-Gries reasoning for weak memory models. In: Halldórsson, M.M., Iwama, K., Kobayashi, N., Speckmann, B. (eds.) Automata, Languages, and Programming. pp. 311–323. Springer, Berlin, Heidelberg (2015)
27. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Computers **28**(9), 690–691 (Sep 1979). https://doi.org/10.1109/TC.1979.1675439
28. Nipkow, T., Prensa Nieto, L.: Owicki/Gries in Isabelle/HOL. In: Finance, J. (ed.) FASE. Lecture Notes in Computer Science, vol. 1577, pp. 188–203. Springer (1999). https://doi.org/10.1007/978-3-540-49020-3_13
29. Ntzik, G., da Rocha Pinto, P., Gardner, P.: Fault-tolerant resource reasoning. In: Feng, X., Park, S. (eds.) APLAS. Lecture Notes in Computer Science, vol. 9458, pp. 169–188. Springer (2015). https://doi.org/10.1007/978-3-319-26529-2_10
30. Owicki, S.S., Gries, D.: An axiomatic proof technique for parallel programs I. Acta Informatica **6**, 319–340 (1976). https://doi.org/10.1007/BF00268134
31. Raad, A., Lahav, O., Vafeiadis, V.: Persistent Owicki-Gries reasoning: a program logic for reasoning about persistent programs on Intel-x86. Proc. ACM Program. Lang. **4**(OOPSLA), 151:1–151:28 (2020). https://doi.org/10.1145/3428219
32. Raad, A., Maranget, L., Vafeiadis, V.: Extending Intel-X86 consistency and persistency: Formalising the semantics of Intel-X86 memory types and non-temporal stores. Proc. ACM Program. Lang. **6**(POPL) (jan 2022). https://doi.org/10.1145/3498683
33. Raad, A., Vafeiadis, V.: Persistence semantics for weak memory: Integrating epoch persistency with the TSO memory model. Proc. ACM Program. Lang. **2**(OOPSLA) (oct 2018). https://doi.org/10.1145/3276507
34. Raad, A., Wickerson, J., Neiger, G., Vafeiadis, V.: Persistency semantics of the Intel-x86 architecture. Proc. ACM Program. Lang. **4**(POPL), 11:1–11:31 (2020). https://doi.org/10.1145/3371079
35. Raad, A., Wickerson, J., Vafeiadis, V.: Weak persistency semantics from the ground up: Formalising the persistency semantics of ARMv8 and transactional models. Proc. ACM Program. Lang. **3**(OOPSLA) (oct 2019). https://doi.org/10.1145/3360561
36. Sewell, P., Sarkar, S., Owens, S., Nardelli, F.Z., Myreen, M.O.: x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. Commun. ACM **53**(7), 89–97 (Jul 2010). https://doi.org/10.1145/1785414.1785443