



# Verification under Intel-x86 with Persistency

PAROSH ABDULLA, Uppsala University, Sweden

MOHAMED FAOUZI ATIG, Uppsala University, Sweden

AHMED BOUAJJANI, Université Paris Cité, France

K. NARAYAN KUMAR, Chennai Mathematical Institute and IRL ReLaX, India

PRAKASH SAIVASAN, Institute of Mathematical Sciences, HBNI and IRL ReLaX, India

The full semantics of the Intel-x86 architecture has been defined by Raad et al in POPL 2022, extending the earlier formalization based on the TSO memory model incorporating persistency. This new semantics involves an intricate combination of the SC, TSO, and PSO models to account for the diverse features of the enlarged instruction set. In this paper we investigate the reachability problem under this semantics, including both its consistency and persistency aspects each of which requires reasoning about unbounded operation reorderings. Our first contribution is to show that reachability under this model can be reduced to reachability under a model without the persistency component. This is achieved by showing that the persistency semantics can be simulated by a finite-state protocol running in parallel with the program. Our second contribution is to prove that reachability under the consistency model of Intel-x86 (even without crashes and persistency) is undecidable. Undecidability is obtained as soon as one thread in the program is allowed to use both TSO variables and two PSO variables. The third contribution is showing that for any fixed bound on the alternation between TSO writes (write-backs), and PSO writes (non-temporal writes), the reachability problem is decidable. This defines a complete parametrized schema for under-approximate analysis that can be used for bug finding.

CCS Concepts: • **Theory of computation** → **Verification by model checking**; • **Software and its engineering** → **Formal software verification**.

Additional Key Words and Phrases: model checking, program verification, TSO memory model, persistency

## ACM Reference Format:

Parosh Abdulla, Mohamed Faouzi Atig, Ahmed Bouajjani, K. Narayan Kumar, and Prakash Saivasan. 2024. Verification under Intel-x86 with Persistency. *Proc. ACM Program. Lang.* 8, PLDI, Article 195 (June 2024), 24 pages. <https://doi.org/10.1145/3656425>

## 1 INTRODUCTION

The semantics of a modern concurrent memory system can be quite complex and hard to comprehend. One component of such semantics is the consistency model which determines the values that read operations may return along computations. The simplest such model is the sequential consistency (SC) model where instructions within each thread are executed in the order in which they are issued (program order) and where memory writes are instantaneously visible to all other threads. In general, for performance reasons, consistency models allow violations of both of these properties: they permit reordering of operations, and they may also make the result of a write visible at different times to different threads (a simple example of this is when the writing thread may read

---

Authors' addresses: [Parosh Abdulla](#), Uppsala University, Uppsala, Sweden, [parosh@it.uu.se](mailto:parosh@it.uu.se); [Mohamed Faouzi Atig](#), Uppsala University, Uppsala, Sweden, [mohamed\\_faouzi.atig@it.uu.se](mailto:mohamed_faouzi.atig@it.uu.se); [Ahmed Bouajjani](#), Université Paris Cité, Paris, France, [abou@irif.fr](mailto:abou@irif.fr); [K. Narayan Kumar](#), Chennai Mathematical Institute and IRL ReLaX, Chennai, India, [kumar@cmi.ac.in](mailto:kumar@cmi.ac.in); [Prakash Saivasan](#), Institute of Mathematical Sciences, HBNI and IRL ReLaX, Chennai, India, [prakshs@imsc.res.in](mailto:prakshs@imsc.res.in).

---



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2024 Copyright held by the owner/author(s).

ACM 2475-1421/2024/6-ART195

<https://doi.org/10.1145/3656425>

the written value before it is available to other threads). These features result in behaviours that do not meet the strong consistency guarantees of the SC model and the set of reachable memory configurations is larger than under SC. Some well known examples of such *relaxed/weakly* memory models are the Total Store Order (TSO) and Partial Store Order (PSO) models, that form part of the model studied in this paper.

In addition, when the memory system includes a mechanism for ensuring resilience to crashes, by the use of non-volatile storage, the semantics includes a second component that determines the order in which writes to memory take effect in such a storage (*persistent* storage). For performance reasons, this order may differ from the order reflected in the consistency model and thus resulting in another dimension of reordering of operations. The result of this second type of reordering is visible after recovering from a crash – the values still available in memory are only those recorded in the persistent memory prior to the crash. As a result, the set of memory configurations reachable along runs with crashes and recoveries include some that are not reachable based only on the consistency model.

Typically, a variety of instructions (fences) are provided in the instruction set which programmers may use in their code to force ordering of instructions, both w.r.t. the consistency and the persistent stages. The behaviour of these fences is another important component of the semantics. (Notice that fences are expensive as they go against the purpose of the reorderings which is to provide better performance.)

These aspects of a modern memory system (consistency, fences and persistency) interact in non-trivial ways and thus programming concurrent applications under such models is hard and error prone. There is a need to develop verification methods applicable to this setting. In this work, we consider the issue of the decidability of the safety verification problem (reducible to solving the state reachability problem) which is fundamental for the development of automatic verification algorithms. We address this issue on a concrete instance that is significant enough to show the main issues that arise in this context – we consider the case of verifying concurrent programs running over the Intel-x86 architecture with persistent memory for which a formal semantics has been defined in [Raad et al. 2022]. We assume in this work that programs have a finite data domain, and concentrate on the decidability issues related only to concurrency and the effect of reordering among operations.

The decidability of reachability verification under weak consistency and persistency has not yet been investigated extensively so far in the literature. The only work we are aware of in this context is [Abdulla et al. 2021a] where the authors prove the decidability of this problem for Persistent x86-TSO (PTSO) [Raad et al. 2020] which is an extension of the TSO (Total Store Order) consistency model with a persistency model. However, PTSO does not capture faithfully the semantics of the Intel-x86 architecture with persistency as has been pointed out in [Raad et al. 2022]. Our aim in this paper is to address the decidability of the verification of the reachability problem for the full semantics of the Intel-x86 architecture with persistency as it was defined in [Raad et al. 2022].

According to the TSO model, write operations issued by a thread are placed in an unbounded FIFO store buffer where they remain pending till committed to (volatile) memory. During this time, the written value is visible only to the writing thread. Commit to memory makes it visible to all other threads simultaneously. While a write is pending in the store buffer, later reads, in program order, by the same thread to other locations can be executed fetching values from the memory. The relaxation provided by TSO model may still be unnecessarily strict for some applications – for instance bulk transfers of video buffers may not need the preservation of ordering between writes on distinct locations.

To provide flexibility the Intel-x86 architecture provides instructions to be used for different types of consistency requirements, ranging from strong SC like to very weak ones. Its subset of

instructions with TSO semantics corresponds actually to the one targeting the so-called *write-back* (wb) memory. There are others. Among the available memory types, uc (for strong-uncacheable) memory allows only updates that are immediately committed, like write operations in the SC model. On the other hand, wc (for write-combining) memory allows reorderings between writes on different variables issued by a same thread, as it can happen in the PSO (Partial Store Order) model (though wc reads differ from PSO reads). In addition to having different semantics for different memory types, writes operations can be declared to be *non-temporal*, which changes their original semantics. Roughly, non-temporal writes (ntw) behave similarly to wc memory updates, allowing reordering between writes on different variables, even if these writes are on the write-back memory. The permitted reorderings of reads in these different memory types have subtle differences and further there are also intricate rules governing reorderings of operations to distinct memory types. The different memory types, non-temporal writes, and the interactions between these features induce a consistency model eTSO (for extended TSO) that is more relaxed and way more complex than the TSO model.

Apart from the consistency stage, the Intel-x86 architecture also includes a persistency stage. Operationally, the consistency model is extended with an additional unbounded buffer to which writes can be moved from the volatile stage observing certain ordering rules. Naturally these rules differ for the different types of memory operations. This model, which we call *epTSO* model, is formalized in [Raad et al. 2022] generalizing the one [Raad et al. 2020] taking into account all the features of the Intel-x86 architectures mentioned above (various types of memories, non-temporal writes, and associated fence operations).

Even for finite-state threads (i.e., the data domain is finite and each thread has a finite control), proving decidability (of reachability) in this context is hard because of the unbounded reorderings of operations permitted by the complex consistency model, the persistency model and their interaction. All this hints at undecidability. Yet, despite the infinity sources in the semantics the models could still satisfy some properties that make their analysis possible. In fact, reachability under both TSO and PSO consistency models for instance is decidable [Atig et al. 2010, 2012]. This has been proved by reduction to reachability problems in well-structured systems [Abdulla et al. 1996; Finkel and Schnoebelen 2001] (a class of systems for which it is known that this problem is decidable). More recently, the reachability problem for PTSO has also been proved to be decidable by a quite sophisticated reduction to well-structured systems. This makes the decidability of reachability an interesting and challenging problem.

Our first contribution is to prove that the reachability problem under epTSO can be reduced to the reachability problem under eTSO. To achieve that, we proceed in two steps. The first and easier step is to eliminate crashes from our analysis. A computation with crashes is decomposed into a sequence of crash-less phases separated by occurrences of crashes. Then the sequence of persistent states at the interfaces between these phases are guessed, reducing the original reachability problem to checking reachability between two given persistent memory states via computations without crashes. The second and involved step is to reduce this crash-free reachability between persistent memory states to reachability without the persistency stage.

Now, to solve the crash-less reachability problem (between persistent memory states), we first observe that (as far as reachability is concerned), wb and ntw updates together with atomic read-writes and fences, can simulate all other types of write operations. Then, the main reduction consists in showing that reachability under the model with persistency can be reduced to reachability under a model without persistency that is an extension of TSO with non-temporal writes (only). The reduction is code-to-code, in the sense that given a program for which a reachability question is stated, this question is reduced to another reachability question for a program that is derived from the original one. The main idea behind the construction is to consider a *manager* running as

an additional thread which, in cooperation with the other threads of the program, implements a protocol that simulates the persistency semantics. Basically the manager guesses for each variable the value that will be recovered from the persistent memory upon the next crash, and then checks the validity of the guess by ensuring that no operation which overwrites the persistent memory in a manner that invalidates the guessed persistent memory state occurs. That this manager is just a thread, and not an external finite state observer synchronizing with all updates, is important – otherwise, the TSO buffers are FIFO, and such a manager in cooperation with the threads can solve undecidable problems such as PCP. This makes interaction of the manager with the program threads quite subtle. Each thread helps the manager in its guessing by marking, for each variable, a write operation that it considers as its last write that persists. The manager takes one of these values as the guessed value for that variable. Then, by watching the operations done by each thread, subsequent to this point, it can check that his guess was right. This requires an intricate protocol of signaling between the threads and the manager. The important and surprising fact is that, the manager can be defined as a *finite-control* process, i.e., it needs to track only a finite amount of extra variables and yet we get rid of the unbounded buffer usually employed to model the persistency stage, e.g., in [Raad et al. 2020] and [Raad et al. 2022]. Another important point is that the manager uses only SC variables. This implies that our reduction can be used already for SC extended with persistency, as well as for TSO or PSO extended with persistency. In particular, this construction remarkably simplifies the models introduced in [Abdulla et al. 2021a; Khyzha and Lahav 2021] to reason about reachability in PTSO.

The question then is whether the reachability problem for Intel-x86 without persistency (the eTSO consistency model) is decidable or not? Our second contribution is to prove its undecidability. This is surprising – the operations on each of the Intel-x86 memory types and non-temporal writes, when considered in isolation, have semantics corresponding to either the SC, TSO, or PSO models, and reachability is known to be decidable under each of these models. However, we prove that very limited interactions between operations in these models, as allowed in eTSO, leads to undecidability. The reachability problem becomes undecidable for programs where all threads are running according to SC except one that uses TSO variables and two PSO variables.

Given the undecidability, a natural approach to this problem is to seek a decidable parametrized bounded under-approximate analysis schema. This means defining some appropriate bounding parameter  $k$  such that the reachability problem is decidable under each fixed bound, i.e., by considering only the program behaviors satisfying the bounding constraint. This is a classical and widely adopted approach in the context of bug finding, particularly for concurrent programs where a number of bounding concepts such as context-bounding [Qadeer and Rehof 2005] and delay-bounding [Emmi et al. 2011] have been introduced. Ideally, the parametrized bounding schema should be complete in the sense that the union of all program behaviors under all bounds is the set of all its possible behaviors, which implies that if there is a bug in the program, there must exist a bound  $k$  where it will be observed.

We take as bounding parameter the number of alternations between wb writes and ntw's along computations. Clearly this bounding schema is complete. We prove that for any fixed bound on such alternations, the reachability problem is decidable. The proof is by a reduction to the reachability problem under PSO which is known to be decidable [Atig et al. 2012]. Our reduction is formulated as a code-to-code translation to PSO. In our reduction, each thread is now represented by  $2k$  threads, one for each of its rounds (in each round it performs either only wb writes or only ntw writes), which run in parallel. The TSO rounds are handled by suitable insertion of fences. There are a number of subtle requirements imposed by fences that require orderings to be enforced between operations among these  $2k$  threads. There is also the flow of information, through read own writes, between these threads that has to be managed. Both of these are handled by a dedicated manager

thread (i.e. we have one manager to handle the  $2k$  threads corresponding to one eTSO thread). Interestingly, this manager is finite-state and needs to track only a finite amount of information at the interface between the different computation rounds.

**Related Work.** Our work uses the formal semantics of Intel-x86 memory types and non-temporal stores that has been proposed in [Raad et al. 2022]. For the related work on formal semantics of weak memory models, the reader is referred to [Raad et al. 2022].

In the following, we focus on the decidability and complexity results for the verification problems of programs running under weak memory models. The decidability and complexity of the reachability problem for program under Total Store Ordering (TSO) has been studied in [Abdulla et al. 2016, 2018; Atig et al. 2010, 2012], Partial Store Ordering (PSO) in [Abdulla et al. 2015b; Atig et al. 2012], POWER architecture in [Abdulla et al. 2020b], Release-Acquire in [Abdulla et al. 2019] and its variants Strong Release/Acquire (SRA) and Weak Release/Acquire (WRA) in [Lahav and Boker 2020, 2022] and promising semantics in [Abdulla et al. 2021b]. The work [Abdulla et al. 2022] studies the reachability problem for TSO programs with dynamic thread creation. The parameterized verification (i.e., the verification of an arbitrary number of identical threads) for TSO has been addressed in [Abdulla et al. 2016, 2018, 2023, 2020a] and Release Acquire in [Krishna et al. 2022].

The robustness problem (which can be seen as a stronger problem than reachability) for programs running under weak memory models has been addressed for TSO [Bouajjani et al. 2013, 2011], POWER architecture [Derevenetc and Meyer 2014] and fragments of C11 memory model [Lahav and Margalit 2019; Margalit and Lahav 2021]. A closer problem of the robustness problem called persistence has been studied in [Abdulla et al. 2015a].

All these works do not consider persistency. [Abdulla et al. 2021a] is the only work (as far as we know) that addresses the decidability and complexity of programs running weak memory models with persistency. However, the considered formal model in [Abdulla et al. 2021a] uses the formal semantics of Intel-x86 persistency that was introduced in [Raad et al. 2020]. This formal semantics considers only write-backs memory and does not model non-temporal stores as it is the case of [Raad et al. 2022]. Our results are different from [Abdulla et al. 2021a]: First, the reachability problem for program under Px86 was shown to be decidable in [Abdulla et al. 2021a] using the framework of well structured systems [Abdulla 2010; Abdulla et al. 1996; Finkel and Schnoebelen 2001] while we show that this problem is undecidable for the full Intel-x86 consistency model regardless of the persistency feature. Furthermore, we show, in this paper, that the reachability problem under the full Intel-86 architecture with persistency can be reduced to the reachability problem under a consistency model without persistency. Finally, our decidability result of the reachability problem when bounding the number of alternation between non-temporal writes and temporal write-back operations is more general than the decidability result of [Abdulla et al. 2021a] and the proof is done by reduction to the reachability problem for PSO.

## 2 PRELIMINARIES

### 2.1 Notation

Let  $\Sigma$  be an alphabet,  $\Sigma^*$  (resp.  $\Sigma^+$ ) denote the set of (non-empty) finite words over  $\Sigma$ . Let  $\epsilon$  denote the empty word. Consider  $w$  a word over  $\Sigma$ , we use  $|w|$  to denote the length of  $w$ . For  $i : 1 \leq i \leq |w|$ , we write  $w[i]$  to denote the  $i^{th}$  letter of  $w$ . For any  $a \in \Sigma$ , we write  $a \in w$  to denote that there exists  $i : 1 \leq i \leq |w|$  such that  $w[i] = a$ . Given two words  $w_1, w_2 \in \Sigma^*$ ,  $w_1 \cdot w_2$  stands for the concatenation of  $w_1$  and  $w_2$ . Given a word  $w \in \Sigma^*$  and  $\Sigma' \subseteq \Sigma$ , we use  $w \downarrow_{\Sigma'}$  to mean the word obtained by deleting from  $w$  all the letters not in  $\Sigma'$ .

```

prog ::= gvars* (thread lvars* instr)* x ∈ gvars, b, a ∈ lvars, e ∈ expr, d ∈  $\mathbb{D}$ ,
instr ::= lbl : stmt
stmt ::= x :=wb a | x :=ntw a | x :=rmw a, b | a := x | a := d | flushopt(x) | flush(x) |
mf | sf | assume e | if e then  $\ell$  | goto  $\ell$ 

```

Fig. 1. Syntax of concurrent programs.

Given two sets  $A$  and  $B$ , we use  $[A \rightarrow B]$  to denote the set of all functions from  $A$  to  $B$  and we write  $f : A \rightarrow B$  to denote that  $f \in [A \rightarrow B]$ . We write  $f[a \leftarrow b]$  to denote the function  $f'$  where  $f'(a) = b$ , and  $f'(a') = f(a')$  if  $a' \neq a$ .

## 2.2 Transition System

Let  $\Sigma$  be a finite alphabet (called also the set of *events*) which contains a special *empty event* (denoted  $\tau$ ). A word  $h \in \Sigma^*$  is called a *history* over  $\Sigma$  if the empty event  $\tau$  does not occur in  $h$ . Let  $w$  be a word over  $\Sigma$ . We use  $\tau^-(w)$  to denote the history  $w \downarrow_{\Sigma \setminus \{\tau\}}$  (obtained from  $w$  by deleting all the occurrences of the empty event).

A *transition system*  $\mathcal{A}$  is defined by a tuple  $\langle \Gamma, \Gamma_{init}, \Sigma_{in}, \Sigma_{out}, \rightarrow \rangle$  where  $\Gamma$  is a set of *configurations*,  $\Gamma_{init} \subseteq \Gamma$  is the set of *initial configurations*,  $\Sigma_{in}$  (resp.  $\Sigma_{out}$ ) is the set of *input* (resp. *output*) events, and  $\rightarrow \subseteq \Gamma \times \Sigma_{in} \times \Sigma_{out} \times \Gamma$  is the *transition relation* of  $\mathcal{A}$ . We use  $\gamma \xrightarrow{e/e'} \gamma'$  to denote that  $\langle \gamma, e, e', \gamma' \rangle \in \rightarrow$ .

We also use  $\gamma_1 \rightarrow \gamma_2$  to denote that there are  $e_1 \in \Sigma_{in}$  and  $e_2 \in \Sigma_{out}$  such that  $\gamma_1 \xrightarrow{e_1/e_2} \gamma_2$ . Let  $\rightarrow^*$  be the reflexive transitive closure of  $\rightarrow$ .

Given a set of configurations  $G \subseteq \Gamma$ , we use  $G \models \gamma'$  to denote that there is a configuration  $\gamma \in G$  such that  $\gamma \xrightarrow{*} \gamma'$ . We abuse the notation and use  $\gamma \models \gamma'$  for  $\{\gamma\} \models \gamma'$ . We say  $\mathcal{A} \models \gamma$  when  $\Gamma_{init} \models \gamma$ . This definition is extended to sets of configurations as expected, denoted by  $\mathcal{A} \models F$  where  $F \subseteq \Gamma$ .

A *run*  $\rho$  of  $\mathcal{A}$  is a sequence of transitions of the form  $\gamma_0 \xrightarrow{e_1/e'_1} \gamma_1 \xrightarrow{e_2/e'_2} \gamma_2 \xrightarrow{e_3/e'_3} \dots \xrightarrow{e_n/e'_n} \gamma_n$ , with  $\gamma_0 \in \Gamma_{init}$  is an initial configuration. Let  $\text{Runs}(\mathcal{A})$  be the set of all runs of  $\mathcal{A}$ . Let  $\llbracket \rho \rrbracket := \langle \tau^-(e_1 \cdot e_2 \cdot e_3 \dots e_n), \tau^-(e'_1 \cdot e'_2 \cdot e'_3 \dots e'_n) \rangle$  and  $\llbracket \mathcal{A} \rrbracket := \{\llbracket \rho \rrbracket \mid \rho \in \text{Runs}(\mathcal{A})\}$ .

In the following, let  $\mathcal{A}_1 \otimes \mathcal{A}_2 = \langle \Gamma, \Gamma_{init}, \Sigma_{in}, \Sigma_{out}, \rightarrow \rangle$  denote the composition of two given transition systems  $\mathcal{A}_1 = \langle \Gamma^1, \Gamma_{init}^1, \Sigma_{in}^1, \Sigma_{out}^1, \rightarrow_1 \rangle$  and  $\mathcal{A}_2 = \langle \Gamma^2, \Gamma_{init}^2, \Sigma_{in}^2, \Sigma_{out}^2, \rightarrow_2 \rangle$  as follows: (1)  $\Gamma = \Gamma_1 \times \Gamma_2$ ,  $\Gamma_{init} = \Gamma_{init}^1 \times \Gamma_{init}^2$ ,  $\Sigma_{in} = \Sigma_{in}^1$ , and  $\Sigma_{out} = \Sigma_{out}^2$ . The transition relation  $\rightarrow$  is defined as the smallest relation such that  $\langle \gamma_1, \gamma_2 \rangle \xrightarrow{e_1/e_2} \langle \gamma_3, \gamma_4 \rangle$  if one of the cases is satisfied: (i)  $\gamma_1 \xrightarrow{e_1/\tau} \gamma_3$ ,  $e_2 = \tau$ , and  $\gamma_2 = \gamma_4$ , (ii)  $\gamma_2 \xrightarrow{\tau/e_2} \gamma_4$ ,  $e_1 = \tau$ , and  $\gamma_1 = \gamma_3$ , or (iii)  $\gamma_1 \xrightarrow{e_1/e} \gamma_3$  and  $\gamma_2 \xrightarrow{e/e_2} \gamma_4$  for some  $e \neq \tau$ . The composition operator  $\otimes$  can be extended to multiple transition systems  $\mathcal{A}_1 \otimes \mathcal{A}_2 \otimes \dots \otimes \mathcal{A}_n$  in the straightforward manner.

## 3 EPTSO– FORMAL SEMANTICS

In this section, we present the formal semantics of concurrent programs running under the ePTSO semantics (following the style of [Abdulla et al. 2021a; Raad et al. 2022]). We assume a finite data domain  $\mathbb{D}$ , which also contains the special value 0. To define a program, we define a simple programming language in Fig. 1. A program  $\mathcal{P}$  is then any code that conforms to this programming language. Notice that a program then contains a set of shared (global) variables (say  $\mathbb{X}$ ) and a set of threads (say  $\Theta$ ). We sometimes refer to a program as  $\mathcal{P} = \langle \Theta, \mathbb{D}, \mathbb{X} \rangle$ . A thread  $\theta \in \Theta$  declares a set  $\text{lvars}_\theta$  of local variables, followed by its code. Let  $\text{lvars} = \bigcup_{\theta \in \Theta} \text{lvars}_\theta$  be the set of all local variables. We assume that the local variables and the global variables range over the data domain  $\mathbb{D}$ . An *instruction* is of the form  $\ell : \text{stmt}$  where  $\ell$  is the *label* and *stmt* is the *statement* of the

instruction. A label occurs at most once in  $\mathcal{P}$ , and hence, for a given label  $\ell$ , the instruction and the thread to which  $\ell$  belongs are uniquely defined. We use  $\Lambda$  to denote the set of all labels. We assume a set  $\text{expr}$  of Boolean expressions involving local and global variables. The set of thread's instructions includes reading, writing, atomic-read-write instructions on shared and local variables, and branching instructions. Moreover, we allow flush operations (**flush<sub>opt</sub>** and **flush**), and fence instructions (**mfence** and **sfence**). There are two types of write instructions, *write-back* (wb) and *non-temporal writes* (ntw).

### 3.1 Handling other type of operations

In Intel-x86 architecture there are several other kinds of reads and writes [Raad et al. 2022]. Further, the memory is partitioned into different types and the operations permitted on a location are determined by its type. We allow only one memory type which corresponds exactly to the wb-type in [Raad et al. 2022]. We also restrict our write operations to ntw and wb writes and rmw. We describe below, how to handle other type of operations using only these operations.

The wc-type memory permits wc writes which have exactly the same behaviour as ntw writes on the pending and persistency stages, so they can be simulated by ntw writes. However, wc reads behave differently from reads on wb type memory. They do not overtake any operation (and reads are not overtaken by any operation). In effect, it is like a wb read but should execute only when the pending buffer is empty. One may be tempted to simulate it by a **mfence** followed by an wb read and this matches the behaviour of wc read in the pending stage. However, memory fence has the effect of flushing the persistent buffers while wc read entails no such flushing. The correct solution is the following and uses an additional helper thread **man** and a new variable  $\blacktriangleleft \text{empty} \blacktriangleright$  which takes values over  $\{\top, \perp\}$ . To simulate a  $\text{rd}_{\text{wc}}(x, a)$  we execute:

**flush** ( $\blacktriangleleft \text{empty} \blacktriangleright$ );  $\blacktriangleleft \text{empty} \blacktriangleright :=_{\text{wb}} \top$ ; **assume** ( $\blacktriangleleft \text{empty} \blacktriangleright = \perp$ );  $a := x$

The helper thread **man** is a thread that non-deterministically executes  $\blacktriangleleft \text{empty} \blacktriangleright :=_{\text{rmw}} \top, \perp$ . Observe that the flush at the beginning of the sequence ensures that no prior writes (in particular ntw writes) can be delayed beyond this point. The write followed by read on  $\blacktriangleleft \text{empty} \blacktriangleright$ , in conjunction with the helper thread, verifies the emptiness of the pending buffer.

The other type of memory considered in [Raad et al. 2022] is the uc-memory. The behaviour of uc-read is exactly identical of that wc-read. The uc write however behaves differently from wb and ntw writes. Its effect on the pending buffer is the same as a **mfence** followed by wb write followed by a **mfence**, but it has no effect on the persistent buffer.

We use the same idea as above but in addition we have to ensure the emptiness of the buffer after the write completes. This can be arranged by executing the above protocol after the write.

We can also comply with the typing of memory by syntactically identifying memory locations with specific memory types. With this translation we can restrict our attention to wb and ntw writes to prove our results.

### 3.2 Semantics

In the following, we give the operational semantics of a program in the epTSO semantics as a composition of three transition systems namely the program, the volatile memory, and the persistent memory transition systems. These transition systems are defined below. For the purpose of these definitions, we fix the threads of program to be  $\Theta$  and the shared variables to be  $\mathbb{X}$ .

**3.2.1 The Program Transition System.** A program  $\mathcal{P}$  induces an transition system  $\mathcal{A}^{\mathcal{P}} = \langle \Gamma^{\mathcal{P}}, \Gamma_{\text{init}}^{\mathcal{P}}, \Sigma_{\text{in}}^{\mathcal{P}}, \Sigma_{\text{out}}^{\mathcal{P}}, \rightarrow^{\mathcal{P}} \rangle$  defined as follows: A configuration of  $\mathcal{A}^{\mathcal{P}}$  is a pair  $\langle \mathcal{L}, \mathcal{R} \rangle$  where  $\mathcal{L} : \Theta \rightarrow \Lambda$  returns, for each thread, the label of the next instruction to be executed, and  $\mathcal{R} : \text{lvars} \rightarrow \mathbb{D}$

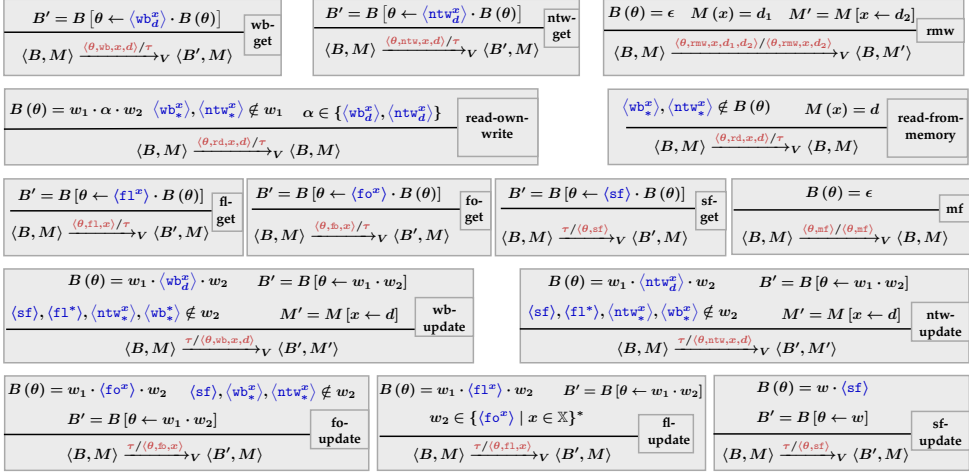


Fig. 2. The volatile stage transition relation.

gives the values of the local variables. There is only one single initial configuration  $\langle \mathcal{L}_{init}, \mathcal{R}_{init} \rangle$  where  $\mathcal{L}_{init}(\theta)$  returns the label of the initial instruction, and  $\mathcal{R}_{init}(a) = 0$  for every  $a \in \text{lvars}$ . The empty event is the only input event (we assume here that our program does not take any input). Let  $\theta \in \Theta$  be a thread,  $x \in \mathbb{X}$  be a variable, and  $d, d_1, d_2 \in \mathbb{D}$  be any values. An output event can be either (1) a *write-back*  $\langle \theta, \text{wb}, x, d \rangle$ , (2) a *non-temporal write*  $\langle \theta, \text{ntw}, x, d \rangle$ , (3) a *read*  $\langle \theta, \text{rd}, x, d \rangle$ , (4) an *rmw*  $\langle \theta, \text{rmw}, x, d_1, d_2 \rangle$ , (5) an *optimized flush*  $\langle \theta, \text{fo}, x \rangle$ , (6) a *flush*  $\langle \theta, \text{fl}, x \rangle$ , (7) a *store fence*  $\langle \theta, \text{sf} \rangle$ , (8) a *memory fence*  $\langle \theta, \text{mf} \rangle$ , or (9) the empty event  $\tau$ . The transition relation is defined in the straightforward manner. We let  $\mathcal{A}^P[\mathcal{L}] = \langle \Gamma^P, \Gamma_{init}^P[\mathcal{L}], \Sigma_{in}^P, \Sigma_{out}^P \rightarrow_P \rangle$  where  $\Gamma_{init}^P[\mathcal{L}] = \{ \langle \mathcal{L}, \mathcal{R}_{init} \rangle \}$  i.e. the transition system obtained by setting  $\mathcal{L}$  as the starting locations.

**3.2.2 The Volatile Stage Transition System.** The *volatile stage transition system*  $\mathcal{A}^V = \langle \Gamma^V, \Gamma_{init}^V, \Sigma_{in}^V, \Sigma_{out}^V \rightarrow_V \rangle$  describes how we handle each instruction of the program in the the volatile stage of ePTSO. Here,  $\Sigma_{in}^V = \Sigma_{out}^V$ , that is, the set of input events is equal to the set of output events from the program. Furthermore, the set of output events  $\Sigma_{out}^V$  is equal to  $\{ \langle \theta, \text{wb}, x, d \rangle, \langle \theta, \text{ntw}, x, d \rangle, \langle \theta, \text{rmw}, x, d, d_1 \rangle \mid (\theta \in \Theta) \wedge (x \in \mathbb{X}) \wedge (d, d_1 \in \mathbb{D}) \} \cup \{ \langle \theta, \text{sf} \rangle, \langle \theta, \text{mf} \rangle \mid \theta \in \Theta \} \cup \{ \langle \theta, \text{fl}, x \rangle, \langle \theta, \text{fo}, x \rangle \mid (\theta \in \Theta) \wedge (x \in \mathbb{X}) \}$ . A configuration of the system is a pair of the form  $\langle B, M \rangle$  where: (1) the map  $B : \Theta \rightarrow A^*$ , with  $A := \{ \langle \text{fl}^x \rangle \mid x \in \mathbb{X} \} \cup \{ \langle \text{fo}^x \rangle \mid x \in \mathbb{X} \} \cup \{ \text{sf} \} \cup \{ \langle \text{wb}_d^x \rangle \mid (x \in \mathbb{X}) \wedge (d \in \mathbb{D}) \} \cup \{ \langle \text{ntw}_d^x \rangle \mid (x \in \mathbb{X}) \wedge (d \in \mathbb{D}) \}$  is the *buffer alphabet*, corresponds to the operations delayed by the corresponding thread, and (2) the map  $M : \mathbb{X} \rightarrow \mathbb{D}$  gives the value of each shared variable in the volatile memory. In the sequel, we refer to  $B$  as the *pending* (or *store buffer*) and the elements residing in it as *messages*. For instance, we say a *wb*-message on  $x$  in the buffer of  $\theta$ , or an *fo*-message, etc. We may refer to messages by their types, e.g., a message of type *wb*, *fo*, etc. We define the initial configurations as:  $\Gamma_{init}^V := \{ \langle B_{init}, M_{init} \rangle \}$ , where  $B_{init}(\theta) := \epsilon$  for any  $\theta \in \Theta$  and  $M_{init}(x) := 0$  for any  $x \in \mathbb{X}$ . We will sometimes use  $\mathcal{A}^V[M] = \langle \Gamma^V, \Gamma_{init}^V[M], \Sigma_{in}^V, \Sigma_{out}^V \rightarrow_V \rangle$  where  $\Gamma_{init}^V[M] = \{ \langle B_{init}, M \rangle \}$  to mean the transition system obtained by modifying the initial volatile memory.

We define the transition relation according to the inference rules of Fig. 2. We classify the set of inference rules in four categories: (i) In the rules *wb-get*, *ntw-get*, *sf-get*, *fl-get*, *fo-get*, the transition system gets the corresponding events from the program. We append the corresponding message

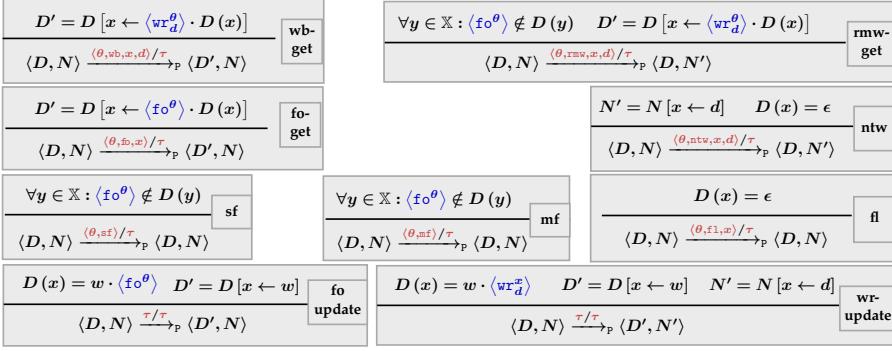


Fig. 3. The persistency stage transition relation.

to the tail of the store buffer. (ii) The transition described in the rule **rmw** can only be executed if the store buffer is empty, the value of  $x$  in the volatile memory is  $d_1$ . The execution of this rule will set the value of  $x$  in the volatile memory to  $d_2$ . Observe that the rule **mf** requires also that the store buffer is empty. (iii) Reading the value of a shared variable  $x$  can be performed by two rules. In **read-own-write**, the store buffer contains a write message on  $x$ . Then, the most recent pending write message on  $x$  is read. In **read-from-memory**, there is no write message on  $x$  in the store buffer. In such case, we read the value of  $x$  directly from the volatile memory. (iv) This category of transitions concerns updates (i.e. propagating the messages from the store buffer). We carry out the updates of **sf** in-order (i.e. it does not re-order with any other instructions). The cases of **ntw** and **wb** are dealt with separately in the rules **wb-update** and **ntw-update**, respectively. In case the message to be updated is of the form  $\langle \text{wb}_d^x \rangle$  then it is propagated only if the following messages are absent in front of it in the buffer: (i)  $\langle \text{sf} \rangle$ - and **f1**-messages, (ii) **ntw**-write messages on the same variable  $x$ , and (iii) **wb**-writes on any variable. In case the message to be update is of the form  $\langle \text{ntw}_d^x \rangle$  then it is propagated only if the following messages are absent in front of it in the buffer: (i)  $\langle \text{sf} \rangle$ - and **f1**-messages, (ii) write-messages (whether **wb** or **ntw**) on the variable  $x$ . If the messages are of the form  $\langle \text{f1}^x \rangle$  or  $\langle \text{fo}^x \rangle$ , then the rules **f1-update** and **fo-update** are used respectively.

**3.2.3 The Persistent Stage Transition System.** We capture the behavior of the persistency stage by the transition system  $\mathcal{A}^P := \langle \Gamma^P, \Gamma_{init}^P, \Sigma_{in}^P, \Sigma_{out}^P, \rightarrow_P \rangle$ . Here  $\Sigma_{in}^P = \Sigma_{out}^P$  and  $\Sigma_{out}^P = \{\tau\}$ . In other words, the transition system gets its input events from the pending stage transition systems. The configurations  $\Gamma^P$  are of the form  $\langle D, N \rangle$  where (1)  $D : \mathbb{X} \rightarrow A^*$ , with  $A := \{ \langle \text{wr}_d^\theta \rangle \mid (\theta \in \Theta) \wedge (d \in \mathbb{D}) \} \cup \{ \langle \text{fo}^\theta \rangle \mid \theta \in \Theta \}$  is the *buffer alphabet*, is the content of the persistency buffer for every shared variable, and (2)  $N : \mathbb{X} \mapsto \mathbb{D}$  gives the value of each variable in the persistent memory. The set of initial configuration is defined by  $\Gamma_{init}^P = \{ \langle D_{init}, N_{init} \rangle \}$  where  $D_{init}(x) = \epsilon$  and  $N_{init}(x) = 0$  for any shared variable  $x$ . We use  $\mathcal{A}^P[N] = \langle \Gamma^P, \Gamma_{init}^P[N], \Sigma_{in}^P, \Sigma_{out}^P, \rightarrow_P \rangle$  where  $\Gamma_{init}^P[N] = \{ \langle D_{init}, N \rangle \}$  to refer to the transition system obtained by replacing the initial persistent memory. The transition relation  $\rightarrow_N$  is given in Fig. 3.

The **wb-get** handles **wb** write messages arriving from the volatile stage. It appends the corresponding message to the tail of the persistency buffer for the relevant variable. The rule **fo-get** concerns **fo**-messages. Both the **wr** and **fo** messages are removed from the buffer in-order. The **ntw** case is handled by the rule **ntw**. It is only enabled if the buffer of the corresponding variable is empty. In this case the value is directly propagated to the persistent memory. The case of **f1** is similar, i.e. the rule **f1** is enabled only if the corresponding buffer of the variable is empty. The case of **rmw**-messages is described in **rmw**. The transition is enabled only if there are no **fo** messages on

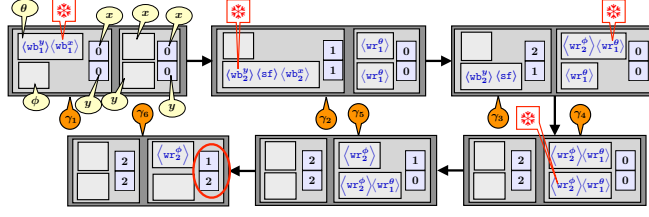


Fig. 4. A run of  $\mathcal{P}_1$  in Fig. 5 that persists  $x = 2$  and  $y = 1$ .

$\theta$  in any of the buffers, where the  $\theta$  is the thread that executed the instruction. The case of **sf** and **mf** are also similar and is given in the rules **sf** and **mf** respectively.

REMARK 1. When only the program transition system and the volatile stage transition systems are involved, we refer to their composition as the *eTSO* system. That is, given a program  $\mathcal{P}$ , we refer to  $\mathcal{A}^{\mathcal{P}} \otimes \mathcal{A}^V$  as the *eTSO* transition system.

### 3.3 The Reachability Problems

Let  $\mathcal{P}$  be a program with a finite set  $\Theta$  of threads and with a set  $\Lambda$  of labels. Let  $\mathcal{A}^{\mathcal{P}}$  be the transition system associated with the program. Let  $\mathcal{L}$  be a labelling function and  $N_1, N_2$  be a pair of persistent memories. The *Crash-Free Reachability Problem* (CRP) asks whether for a given configuration  $\gamma$  of  $\mathcal{A}^{\mathcal{P}}$ , if  $\gamma$  can be reached (i.e.  $(\mathcal{A}^{\mathcal{P}}[\mathcal{L}]) \otimes (\mathcal{A}^V[N_1]) \otimes (\mathcal{A}^P[N_1]) \models \langle \gamma, \gamma_1, \gamma_2 \rangle$  for some  $\gamma_1$  and  $\gamma_2$ ). The *Crash-Free Persistent Reachability Problem* (CPRP) asks whether for a given  $N_2$ , there is a  $\gamma_2 = \langle D, N_2 \rangle$  for some  $D$  such that  $(\mathcal{A}^{\mathcal{P}}[\mathcal{L}]) \otimes (\mathcal{A}^V[N_1]) \otimes (\mathcal{A}^P[N_1]) \models \langle \gamma, \gamma_1, \gamma_2 \rangle$  for some  $\gamma$  and  $\gamma_1$ . That is, whether a configuration with the persistent memory as  $N_2$  can be reached when starting from  $N_1$  in both the persistent memory and the volatile memory. With an abuse of notation, we denote this as  $(\mathcal{A}^{\mathcal{P}}[\mathcal{L}]) \otimes (\mathcal{A}^V[N_1]) \otimes (\mathcal{A}^P[N_1]) \models N_2$ .

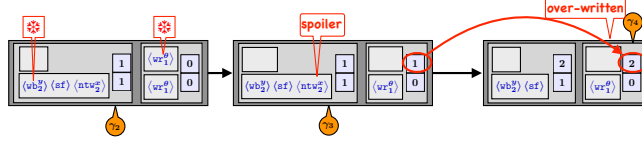
Let  $\text{Rec} : [\mathbb{X} \rightarrow \mathbb{D}] \rightarrow [\Theta \rightarrow \Lambda]$  be a recovery function that associates for each valuation of the persistent memory a labelling function. Intuitively, the recovering function defines the new initial labels of the threads after a crash of the system. We assume w.l.o.g that  $\text{Rec}(N_{\text{init}}) = \mathcal{L}_{\text{init}}$  and that the recovery function is computable. Let  $N$  be an valuation of the persistent memory. We define  $\mathcal{P} \square N$  to be the event transition system  $(\mathcal{A}^{\mathcal{P}}[\text{Rec}(N)]) \otimes (\mathcal{A}^V[N]) \otimes (\mathcal{A}^P[N])$ . The *Full-Reachability Problem* (FRP) asks whether, for a given configuration  $\gamma$  of  $\mathcal{A}^{\mathcal{P}}$  and a recovery procedure  $\text{Rec}$ , there exists a finite sequence  $\mathcal{A}_0 N_0 \mathcal{A}_1 N_1 \dots \mathcal{A}_n$  such that: (1)  $\mathcal{A}_0 = \mathcal{A}^{\mathcal{P}} \otimes \mathcal{A}^V \otimes \mathcal{A}^P$ , (2)  $\mathcal{A}_i \models N_i$  and  $\mathcal{A}_{i+1} = \mathcal{P} \square N_i$  for all  $i < n$ , and (3)  $\mathcal{A}_n \models \langle \gamma, \gamma_1, \gamma_2 \rangle$  for some  $\gamma_1$  and  $\gamma_2$ .

To solve the full-reachability problem, it is sufficient to guess the intermediary valuations of the persistent memory  $N_0, N_1, \dots, N_n$ , then solve the crash-free persistent reachability problems  $\mathcal{A}_i \models N_i$  and the crash-free reachability problem  $\mathcal{A}_n \models \langle \gamma, \gamma_1, \gamma_2 \rangle$ .

THEOREM 3.1. The full-reachability problem is reducible to the crash-free persistent and crash free reachability problems.

## 4 REMOVING THE PERSISTENCY STAGE

In this section we show informally how we can eliminate the persistency stage, while preserving correctness modulo reachability. More precisely, given a program  $\mathcal{P}$ , we translate  $\mathcal{P}$  to a new program  $\mathcal{P}'$  such that the CRP of  $\mathcal{P}'$  is equivalent to the CPRP of  $\mathcal{P}$ . We provide an overview of the translation, through examples in Sub-sections 4.1 to 4.5.2 and then give the formal reduction in 4.6

Fig. 6. An ntw-spoiled run of  $\mathcal{P}_2$  in Fig. 5.

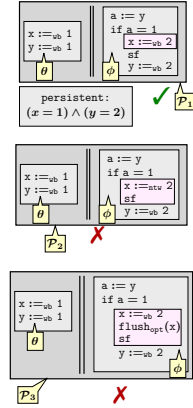
#### 4.1 Speculation

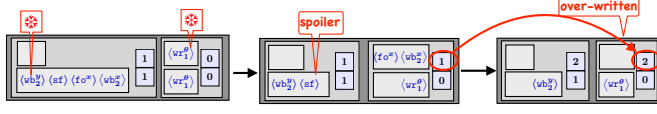
At the heart of our translation scheme is a *speculation procedure*: for each variable  $x$ , the protocol *guesses* and *freezes* the value  $d$  of an arbitrary write message on  $x$ . The protocol then ensures that the value of  $x$  in the persistent memory is  $d$  just before the next crash. We implement the protocol with the help of an extra thread, the *manager*, that acts according to SC, and that verifies the consistency of these guesses. In implementing such a protocol, we need to handle these challenges:

- *Soundness*: We must preserve the behavior of the input program  $\mathcal{P}$  up to reachability.
- *Freezing*: The manager and the other threads must agree on a freezing point for each variable.
- *Non-Spoiling*: the frozen values should not be *spoiled*, i.e., they should not be overwritten in the persistent memory.

Consider the program  $\mathcal{P}_1$  of Fig. 5 consisting of the threads  $\theta$  and  $\phi$ , in which we would like to persist the values  $x = 1$  and  $y = 2$ . In Fig. 4, we give a run that persists these values. In  $\gamma_1$ , the thread  $\theta$  has executed its instructions and has placed the corresponding messages in its pending buffer. The run defines the freezing point of  $x$  to be the message  $\langle wb_1^x \rangle$ . In  $\gamma_2$ , the two messages of  $\theta$  have updated the volatile memory and crossed to the persistency stage. In particular, the message  $\langle wr_1^x \rangle$  is induced by the message that was the freezing point of  $x$ . Also, in  $\gamma_2$ , the thread  $\phi$  has performed its three instructions and placed the corresponding messages in its pending buffer. The configurations  $\gamma_3$  and  $\gamma_4$  show these messages update the volatile memory and cross to the persistency buffers. In  $\gamma_5$ , the message  $\langle wr_1^x \rangle$  has updated the persistent memory. Since this message is a freezing point, the run can no longer write to  $x$  in the persistent memory; otherwise, we would overwrite a frozen value. In  $\gamma_6$ , we have transferred the messages in the  $y$ -persistency buffer to the persistent memory, and, in particular, we have obtained the desired values of  $x$  and  $y$ .

Consider the program  $\mathcal{P}_2$  of Fig. 5, which we get from  $\mathcal{P}_1$  by replacing the instruction  $x :=_{wb} 1$  by the instruction  $x :=_{ntw} 1$ . Also, consider the run of  $\mathcal{P}_2$  depicted in Fig. 6. The configurations  $\gamma_1$  and  $\gamma_2$  are similar to Fig. 4. We notice that, due to the presence of `(sf)`, the message  $\langle wb_2^y \rangle$  will reach the end of the pending buffer after the message  $\langle ntw_2^x \rangle$  (this is true even in the case of  $\mathcal{P}_1$ ). The main difference is that the  $\langle ntw_2^x \rangle$  cannot cross to the persistency stage until the  $x$ -persistency buffer is empty. Furthermore, the `ntw` must hit the persistent memory immediately without passing through the  $x$ -persistency buffer. This means that the frozen value  $x = 1$  will be overwritten in the persistent memory. We say that the  $\langle ntw_2^x \rangle$  acts as a *spoiler*. Sometimes, we explicitly refer to the spoiler's type, so, in this case, we say that  $\langle ntw_2^x \rangle$  is an *ntw-spoiler*.

Fig. 5. Three programs with threads  $\theta$  and  $\phi$ . The differences between program codes are highlighted in pink.

Fig. 7. An SFW-spoiled run of  $\mathcal{P}_2$  in Fig. 5.

In our construction, we will rely on the fact that to be able to persist a set of write messages, we should be able to find a run along which (i) we can freeze the values of the variables in *some* order, and (ii) once the value of a given variable is frozen, then the value will not be spoiled (overwritten in the persistent memory). In general, different program runs may use different spoilers to overwrite the variable values in the persistent memory. In Fig. 4, we demonstrated that  $\mathcal{P}_1$  has such a run that allows freezing the correct value without spoiling. In particular,  $\langle wb_2^x \rangle$  is not a spoiler in the run of  $\mathcal{P}_1$  in Fig. 4 since it can be transferred to the persistent memory only after all the correct variable values have persisted.

Next, we consider another type of spoiler, namely sf-fo-wr-spoilers (SFW-spoilers for short). We illustrate SFW-spoilers using the program  $\mathcal{P}_3$  of Fig. 5. In any program run, the three highlighted instructions of  $\phi$  will act as a spoiler. The three instructions generate the following sequence of messages  $\langle sf \rangle \langle fo^x \rangle \langle wb_2^x \rangle$ , see Fig. 7. According to the epTSO semantics, these messages cannot be re-ordered in the pending buffer. Therefore,  $\langle wb_2^x \rangle$  will first enter the x-persistence buffer as the message  $\langle wr_2^0 \rangle$ . Next, the message  $\langle fo^x \rangle$  will also enter the x-persistence buffer, and it cannot be re-ordered with  $\langle wb_2^x \rangle$ . Finally, when the message  $\langle sf \rangle$  reaches the end of the pending buffer, it forces the message  $\langle fo^x \rangle$  to leave the x-persistence buffer, which, in turn, causes the message  $\langle wb_2^x \rangle$  to persist. In general, to be a spoiler, an operation or combination of operations should be able to force a new value to be persisted (after the freezing). There are also other types of spoilers namely fl-wr, mf-fo-wr and rmw-fo-wr-spoilers, which are similar.

## 4.2 Soundness and Visibility

As we mentioned in Section 4.1, spoiler detection is a vital component of the speculation protocol. We must enable the manager to detect the spoilers that reach the shared memory. Recall that we no longer have the persistency stage; hence, we need to make all decisions based on messages that reach the volatile memory. The manager cannot detect spoilers only by inspecting the values of the variables in the volatile memory. The reason is twofold: (i) ntw-spoilers are difficult to detect since we cannot distinguish between values written by wb or ntw instructions. A value, say  $x = 2$  in the memory, does not reveal whether the writer was of type wb or ntw. (ii) We cannot detect WFS or WFF spoilers since fences and barriers do not modify the memory in the first place. To solve this problem, we “divert the traffic” from the threads to the memory and make it pass through the manager. More precisely, we now have additional copies of the memory locations. The threads write to one of the copy and the manager moves

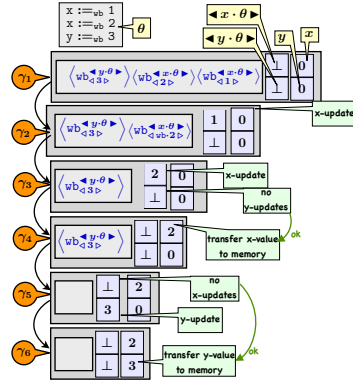


Fig. 8. The Writing Protocol, illustrated on a single thread  $\theta$  operating on two shared variables  $x$  and  $y$ . The text boxes, with a green background, on the right-hand side of the figure describe the manager's actions. We also show part of the shared memory: the variables  $x$ ,  $y$ ,  $\langle x \cdot \theta \rangle$ , and  $\langle y \cdot \theta \rangle$ .

the written value to the main memory. Together with the normal data values, we augment the messages travelling through the store buffers with additional information that helps the manager to detect freezing points, the types of writes, etc. Furthermore, we make fences and barriers visible to the manager by replacing them with write instructions on special variables. The manager inspects the arriving messages looking for frozen messages and spoilers and then copies the relevant data, i.e., the variable's values, to the shared memory. A crucial challenge is to ensure that replacing instructions in this manner is “sufficiently precise” to preserve the epTSO semantics. We achieve this objective using the protocols of the following subsections. The manager processes exactly one message from one thread at a time. So messages are processed atomically, preserving order within threads (however some messages may be missed) and interleaving across threads.

### 4.3 The Writing Protocol

Consider the single-thread program  $\mathcal{P}_1$  of Fig. 8 performing three wb instructions. The writing protocol ensures that the manager observes and transfers enough write messages to preserve the epTSO semantics (up to reachability). We only need to preserve *sufficiently many* messages, but not necessarily *all* messages, since, in a similar manner to the classical TSO semantics, the epTSO semantics is *almost lossy* (but not *entirely lossy*). In a wb message sequence, on the same variable, in a store buffer, we can lose all but the last message without compromising the semantics. As far as the manager is concerned, it needs to observe the last message in such a sequence, while it may or may not see the rest. We call this the *last-message* guarantee. In Fig. 8, we need to ensure that the manager observes the second write instruction, but not necessarily the first. If the manager missed the second write instruction, we would allow a memory configuration where the values of  $x$  and  $y$  are 1 and 3, respectively. This memory configuration is not reachable in the epTSO semantics. For each thread  $\theta \in \Theta$  and variable  $x \in \mathbb{X}$ , the protocol uses a shared variable  $\blacktriangleleft x \cdot \theta \blacktriangleright$ . In  $\gamma_1$ , the thread  $\theta$  has executed all its instructions, resulting in the three messages we show in the figure. At the other end of the buffer, the manager will wait for the variables  $\blacktriangleleft x \cdot \theta \blacktriangleright$  and  $\blacktriangleleft y \cdot \theta \blacktriangleright$  to be populated. The first message will update the value of  $\blacktriangleleft x \cdot \theta \blacktriangleright$  (configuration  $\gamma_2$ ). In this program run, the manager will not notice this message since its value is overwritten by the next message (configuration  $\gamma_3$ ). In  $\gamma_3$ , the shared variable  $\blacktriangleleft x \cdot \theta \blacktriangleright$  carries the last written value on  $x$  before a write message on another variable ( $y$  in this case) arrives. The one at a time feature means that the manager must process this last written value on  $x$  before the message on  $y$  arrives — it fetches the value 2 of  $x$  verifying that no other variable updates have occurred. It thus provides the last-message guarantee. The rest of the simulation similarly transfers the write message on  $y$ .

### 4.4 The Freezing Protocol

The aim of the *freezing protocol* is twofold. Firstly it allows, for each shared variable  $x \in \mathbb{X}$ , to guess and *freeze* the value  $d$  of a particular write instruction on  $x$ . The intuition here is that (i)  $d$  will persist and (ii)  $d$  will not be overwritten in the persistent memory until the next crash occurs. It also facilitates each thread to guess within its execution, the position where the freeze occurs.

Fig. 9 shows the simulation of the protocol for a program with two threads  $\theta$  and  $\phi$  sharing two variables  $x$  and  $y$ . Each thread guesses, for each variable  $x \in \mathbb{X}$ , an  $x$ -*freezing point*. The freezing point is defined by a write message which we enrich by an extra flag  $\ast$ . A typical example of such a message is  $\langle \text{wb}_{\blacktriangleleft 1 \cdot \ast \blacktriangleright}^{\blacktriangleleft x \cdot \theta \blacktriangleright} \rangle$  that is issued by thread  $\theta$  in Fig. 9. This will tell the manager that  $\theta$  expects the value of  $x$  to be frozen after the message is fetched from the store buffer but before the next message of the form  $\langle \text{wb}_{\blacktriangleleft d \blacktriangleright}^{\blacktriangleleft x \cdot \theta \blacktriangleright} \rangle$  is fetched from the buffer. The manager, for its part, waits until the freezing points for  $x$  have arrived from all the threads. At that point, the manager *freezes* the value of  $x$  currently in the shared memory, we refer to this point in simulation as  $x$ -freeze.

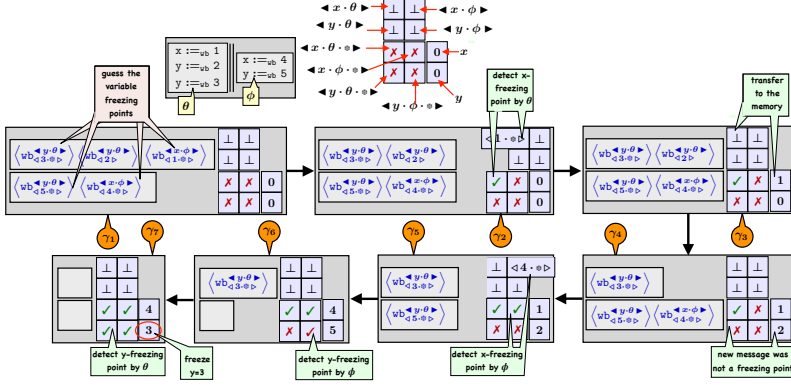


Fig. 9. The Freezing Protocol. We simulate the protocol on the program shown in the top-left corner. Together with the pending buffers, we depict part of the shared memory, namely the variables  $x$ ,  $y$ , and the variables  $\langle x \cdot \theta \rangle$ ,  $\langle y \cdot \theta \rangle$ ,  $\langle x \cdot \phi \rangle$ , and  $\langle y \cdot \phi \rangle$ . We also show some of the manager's freeze variables, namely the local variables  $\langle x \cdot \theta \rangle$ ,  $\langle y \cdot \theta \rangle$ ,  $\langle x \cdot \phi \rangle$ , and  $\langle y \cdot \phi \rangle$ . The pink text boxes describe the thread's actions, while the green ones describe the manager's actions.

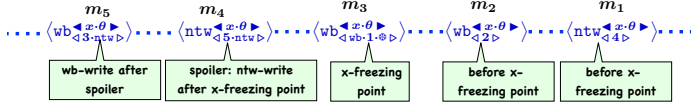


Fig. 10. The ntw Protocol.

In Fig. 9, the threads guess the freezing points for  $x$  to be the first messages in their respective buffers and guess the freezing points for  $y$  to be the last messages in the buffers (configuration  $\gamma_1$ ). In  $\gamma_2$ , the  $x$ -freezing point of  $\theta$  has reached the memory, to which the manager reacts by ticking its local variable  $\langle x \cdot \theta \rangle$ . The manager copies the value 1 of  $x$  carried by the message to the variable  $x$  in the shared memory (configuration  $\gamma_3$ ). The next message fetched from the buffer (corresponding to the second instruction of  $\theta$ ) is not a freezing point, so the freezing protocol does not react to it. However, the value will still be transferred to the memory (configuration  $\gamma_4$ ). In  $\gamma_5$ , the manager receives the  $x$ -freezing point of  $\phi$ . At this point, the manager has received the  $x$ -freezing points of all threads. The manager freezes the value of  $x$  in the shared memory, which in  $\gamma_5$  is equal to 4. Notice that although the manager requires all the threads to propose their own  $x$ -freezing points, it freezes only a single value, namely the value accompanying the last  $x$ -freezing point. In this case, the  $x$ -freezing point of  $\phi$  arrived last (i.e., after the  $x$ -freezing point of  $\theta$ ), and hence the value 4 was frozen (rather than 1). We might get the impression that, for a given variable  $x$ , it is sufficient that only one of the threads guesses the freezing of  $x$ . After all, only the last freezing point for  $x$  is taken to consideration. However, as we see below, the per-thread freezing points for  $x$  are needed so that the threads can help the manager to handle potential  $x$ -spoilors.

#### 4.5 Handling Spoilers

**4.5.1 The ntw Protocol.** We recall that an ntw-spoiler is an ntw-message on  $x$  in the buffer of a thread  $\theta$ , that occurs after the  $x$ -freezing point of  $\theta$ . The goal of the ntw-protocol is to enable the manager to detect NTW-spoilers (cf. Fig. 10). To that end, we enrich the data domain by values of the form  $\langle d \cdot \text{ntw} \rangle$  where  $d \in \mathbb{D}$ , i.e., we tag the written values by an ntw-flag. Before the  $x$ -freezing point, we generate write messages using the standard (un-tagged) values (the messages  $m_1$  and  $m_2$

in Fig. 10. After the  $x$ -freezing of  $\theta$  occurs (the message  $m_3$  in Fig. 10, we watch out for the first ntw-instruction on  $x$  by  $\theta$  (the message  $m_4$  in Fig. 10). The message  $m_4$  is an ntw-spoiler of  $x$ . Instead of generating a message  $\langle \text{wb}_{\langle 5 \rangle}^{\langle x \cdot \theta \rangle} \rangle$ , the thread will generate the message  $\langle \text{ntw}_{\langle 5 \cdot \text{ntw} \rangle}^{\langle x \cdot \theta \rangle} \rangle$ . In other words, the value 5 of the message is tagged with the ntw-flag. When the manager sees the message, it knows from the tag that it is an ntw-spoiler. From this point on, the threads tags all write messages on  $x$ , whether of type wb or type ntw, with the ntw-flag, i.e., it only generates messages of the form  $\langle \text{wb}_{\langle \text{ntw} \cdot d \rangle}^{\langle x \cdot \theta \rangle} \rangle$  or for the form  $\langle \text{ntw}_{\langle \text{ntw} \cdot d \rangle}^{\langle x \cdot \theta \rangle} \rangle$ . The reason is that these subsequent messages, e.g., the message  $m_5$  in Fig. 10, may overwrite the spoiler. For instance, assume we do not tag  $m_5$  with ntw. A possible scenario is that the manager misses  $m_4$  and only reads the memory after  $m_5$  has arrived (this is an allowed behavior according to the almost-lossiness property we described above). This means that when the manager reads the memory, the message  $m_5$  has already overwritten the value  $\langle 5 \cdot \text{ntw} \rangle$ , written by  $m_4$ , and replaced it with the un-tagged value 4; and hence the manager have missed the fact that a spoiler has occurred. With the tagging of the subsequent messages, this scenario will not occur. More precisely, whenever the manager sees the ntw-flag in a write message on  $x$ , it knows that either the message itself or a preceding message is an  $x$ -spoiler. Two further remarks: First, the manager halts the program execution whenever it sees an ntw-tagged message since such a message indicates the existence of an ntw-spoiler. Therefore, these tagged messages never reach the (volatile) memory. As for read-own-operations, the thread strips off the ntw-flag and treats the message as a regular write operation. Second, the tagged messages preserves the message type (wb- or ntw-type message), and hence the protocol does not affect the re-ordering of messages inside the buffers.

**4.5.2 The SFW Protocols.** We describe the idea behind detecting any SFW spoiler in detail. Such a spoiler involves a pattern consisting of an  $x$ -freezing point, a write to  $x$ , an  $\text{flush}_{\text{opt}}$ -message on  $x$ , and an  $\text{sf}$ -message. One difference with the case of ntw-spoilers is that that the write to  $x$  may be performed by another thread. We use this thread to reveal the possibility of such a spoiler to the manager.

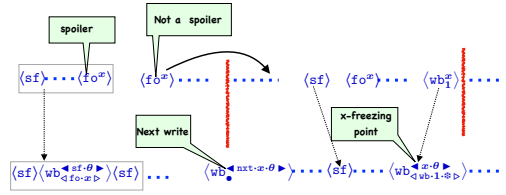


Fig. 11. The SFW Protocol.

The thread waits until it is past the freezing point for  $x$  before activating the SFW-protocol. Once we are past this point, the thread switches to helping the manager detect a possible SFW spoiler by going three phases, (i), (ii) and (iii), as follows. In phase (i), it guesses the position where it expects the violating write on  $x$  to occur (which is possibly from another thread) and marks it with a write to a special variable  $\langle \text{next} \cdot x \cdot \theta \rangle$ . This guess can be verified by the manager. After inserting this message the thread changes its behaviour and enters phase (ii). It no longer executes the  $\text{flush}_{\text{opt}}(x)$  but looks for one which can be a potential spoiler. Note that the  $\langle \text{fo}^* \rangle$  can re-order with other messages, hence not all the  $\text{flush}_{\text{opt}}(x)$  in phase (ii) are potential spoilers. It remembers, in its local state, if a potential spoiler indeed occurs. In phase (iii) the thread tracks  $\langle \text{sf} \rangle$ -messages. The reason is that an  $\text{flush}_{\text{opt}}$ -instruction would contribute a spoiler only if it is followed by an  $\langle \text{sf} \rangle$ -message. As mentioned earlier, there is no way for the manager to observe an  $\text{sf}$ -message since it does not modify the memory. Hence, if such a  $\langle \text{sf} \rangle$ -message were to be generated, the thread inserts a write on the variable  $\langle \text{sf} \cdot \theta \rangle$  with a special value  $\langle \text{flush}_{\text{opt}} \cdot x \rangle$  as a signal to the manager that a SFW spoiler has occurred. If this message reaches the manager it detects the spoiler and aborts. Replacing an  $\langle \text{sf} \rangle$  by a write to the variable  $\langle \text{sf} \cdot \theta \rangle$  can allow more behaviours since a write can re-order with other operations where as an  $\text{sfence}$  cannot. We remedy this by guarding the write with  $\langle \text{sf} \rangle$  messages. The other spoilers are handled using similar ideas.

#### 4.6 Formal Translation

Recall that in CPRP, we are given a program  $\mathcal{P} = (\Theta, \mathbb{D}, \mathbb{X})$  and a persistent memory  $N$ , we ask if  $(\mathcal{A}^{\mathcal{P}}) \otimes (\mathcal{A}^V) \otimes (\mathcal{A}^P) \models N$ . We show in this section that we can construct another program  $\llbracket \mathcal{P} \rrbracket = \langle \llbracket \Theta \rrbracket, \llbracket \mathbb{D} \rrbracket, \llbracket \mathbb{X} \rrbracket \rangle$  and a configuration  $\gamma$  of  $\mathcal{A}^{\llbracket \mathcal{P} \rrbracket}$  such that  $\gamma$  can be reached if and only if  $N$  can be reached, as stated in the below theorem.

**THEOREM 4.1.** *Given a program  $\mathcal{P}$  and a persistent memory  $N$ , we can construct another program  $\llbracket \mathcal{P} \rrbracket$  and a configuration  $\gamma$  such that*

$$(\mathcal{A}^{\mathcal{P}}) \otimes (\mathcal{A}^V) \otimes (\mathcal{A}^P) \models N \iff \exists \gamma_1, \gamma_2 \left( \mathcal{A}^{\llbracket \mathcal{P} \rrbracket} \otimes (\mathcal{A}^V) \otimes (\mathcal{A}^P) \models \langle \gamma, \gamma_1, \gamma_2 \rangle \right)$$

*That is, the CPRP reachability in  $\mathcal{P}$  reduces to CRP reachability in  $\llbracket \mathcal{P} \rrbracket$ .*

We show in the Figure 12, the translation of the given program  $\mathcal{P} = \langle \Theta, \mathbb{D}, \mathbb{X} \rangle$  into  $\llbracket \mathcal{P} \rrbracket = \langle \llbracket \Theta \rrbracket, \llbracket \mathbb{D} \rrbracket, \llbracket \mathbb{X} \rrbracket \rangle$ , in particular we have

- $\llbracket \Theta \rrbracket = \{ \llbracket \theta_1 \rrbracket, \dots, \llbracket \theta_n \rrbracket \} \cup \{ \theta_{\text{man}} \}$ , where  $\theta_{\text{man}}$  is the manager thread that acts as a validator.
- $\llbracket \mathbb{X} \rrbracket = \mathbb{X} \cup \mathbb{X}' \cup \mathbb{V}_{\text{instr}}$  with  $\mathbb{X}' = \{ x' \mid x \in \mathbb{X} \}$  and  $\mathbb{V}_{\text{instr}} = \{ \blacktriangleleft x \cdot \theta \blacktriangleright, \blacktriangleleft \text{sf} \cdot \theta \blacktriangleright, \blacktriangleleft \text{fl} \cdot x \blacktriangleright, \blacktriangleleft \text{fo} \cdot x \cdot \theta \blacktriangleright, \blacktriangleleft \text{nxt} \cdot x \cdot \theta \blacktriangleright \mid x \in \mathbb{X}, \theta \in \Theta \}$ .
- $\llbracket \mathbb{D} \rrbracket = \mathbb{D}_{\text{wr}} \cup \mathbb{D}_{\text{sync}} \cup \mathbb{D}$ , where  $\mathbb{D}_{\text{wr}} = \{ \blacktriangleleft d \cdot s \blacktriangleright \mid s \in \{ \otimes, \text{ntw} \}, d \in \mathbb{D} \}$ ,  $\mathbb{D}_{\text{sync}} = \{ \blacktriangleleft \text{rmw} \blacktriangleright, \blacktriangleleft \text{rmw} \cdot \otimes \blacktriangleright, \blacktriangleleft \text{fo} \cdot x \blacktriangleright, \bullet \}$

**Threads.** The thread  $\llbracket \theta_i \rrbracket$  is obtained through a line by line translation of the thread  $\theta_i$ , as shown in the Figure 12. That is, if  $\theta_i = \ell_1^i : \text{st}_1^i; \ell_2^i : \text{st}_2^i; \dots; \ell_m^i : \text{st}_m^i$ , where  $\ell_i : \text{st}_i$  is the label, statement pair, then  $\llbracket \theta_i \rrbracket = \llbracket \ell_1^i : \text{st}_1^i \rrbracket; \llbracket \ell_2^i : \text{st}_2^i \rrbracket; \dots; \llbracket \ell_m^i : \text{st}_m^i \rrbracket; \ell_{\text{end}}^i : \text{halt}$ . For any statement  $\ell : \text{st}$ , its translation  $\llbracket \ell : \text{st} \rrbracket$  will be defined soon.

$$\begin{aligned} \llbracket \mathcal{P} \rrbracket &:= \llbracket \mathbb{X} \rrbracket \cdot (\llbracket \Theta \rrbracket)_{\theta \in \Theta} \cdot \theta_{\text{man}} \\ \llbracket \mathbb{X} \rrbracket &:= \mathbb{X} \cup \mathbb{X}' \cup \mathbb{V}_{\text{instr}} \\ \llbracket \text{id lvars } \ell_1^\theta : \text{st}_1^\theta \dots \ell_m^\theta : \text{st}_m^\theta \rrbracket &:= \\ &\quad \text{id } \llbracket \text{lvars} \rrbracket \llbracket \ell_1^\theta : \text{st}_1^\theta \rrbracket \dots \llbracket \ell_m^\theta : \text{st}_m^\theta \rrbracket \cdot \ell_{\text{end}}^\theta : \text{halt} \\ \llbracket \text{lvars} \rrbracket &:= \text{lvars} \cup \mathbb{I}_{\text{prg}} \end{aligned}$$

Fig. 12. Program translation

**Shared Variables and Data.** Let us examine the shared variables and data-values listed above and explain their roles. We begin with the variables in  $\mathbb{V}_{\text{instr}}$ . The variable  $\blacktriangleleft x \cdot \theta \blacktriangleright$  is used by the thread  $\theta$  to implement the wb or ntw writes in such a way that the manager is aware of it, this is as explained in Sections 4.3. The variables  $\blacktriangleleft \text{sf} \cdot x \cdot \theta \blacktriangleright$ , is used to make visible the occurrences of SFW spoiler to the manager. Recall that an **sfence** is replaced by a write to this variable in case the thread identifies it as a spoiler, this was discussed in section 4.5.2. Further, in the phase-i of the protocol, each thread speculated where it expects a violation and writes to a variable  $\blacktriangleleft \text{nxt} \cdot x \cdot \theta \blacktriangleright$  at that position. We also have other variable that will be used as part to detect the other spoilers.

Next we turn our attention to the data values. As indicated in Section 4.4, the freeze protocol required each thread to speculate a freeze point. This was achieved by tagging the data value with a  $\otimes$  tag. Similarly the ntw protocol described in Section 4.5.1, on detecting an ntw spoiler required that the data value is tagged with an ntw tag. The data values  $\mathbb{D}_{\text{wr}}$  serve this purpose. The values in  $\mathbb{D}_{\text{sync}}$  will be used to handle the synchronisations due to rmw, sf, fl instructions.

**Local variables.** To implement the freezing protocol from Section 4.4 each thread  $\theta$  has a variable  $\blacktriangleleft x \cdot \text{lfrz} \blacktriangleright$  to indicate if it has issued the freezing write on  $x$ . The process uses the variable  $\blacktriangleleft x \cdot \text{ntw} \blacktriangleright$  to remember an ntw spoiler, the threads then ensure that any write that follows this is also tagged as a spoiler (see Section 4.5.1). To implement the SFW protocol (and other spoiler protocol) correctly, each thread  $\theta$ , has a local variable  $\blacktriangleleft x \cdot \text{lpfo} \blacktriangleright, \blacktriangleleft x \cdot \text{fo} \blacktriangleright, \blacktriangleleft \text{nxt} \cdot x \blacktriangleright$ . The need for the local variable  $\blacktriangleleft x \cdot \text{lpfo} \blacktriangleright$  is important and subtle. It is used to arrange the re-orderings of the **fo** with the other messages. Recall we mentioned that every **flush<sub>opt</sub>** in phase ii of the SFW protocol need not be a potential spoiler. This is because the **fo** can be removed from the

store buffer before the write to  $\langle \text{nxt} \cdot x \rangle$  is removed from the buffer. The the local variable  $\langle x \cdot \text{lpfo} \rangle$  allows for such re-orderings. It is set as soon as  $\langle \text{nxt} \cdot x \rangle$  is written to and any  $\text{flush}_{\text{opt}}$  when it is set is deemed not a potential spoiler. The variable  $\langle x \cdot \text{lfo} \rangle$  is used to remember a potential **fo** that is a spoiler. Thus the set of local variables used by each thread is given by  $\mathbb{L}_{\text{prg}} = \{\langle x \cdot \text{lpfo} \rangle, \langle x \cdot \text{lfo} \rangle, \langle x \cdot \text{lfrz} \rangle, \langle x \cdot \text{ntw} \rangle, \langle \text{nxt} \cdot x \rangle \mid x \in \mathbb{X}\}$ .

Next we examine the local variables used by the manager thread. It uses a local variable  $\langle x \cdot \theta \cdot \text{lfrz} \rangle$  to remember whether the freeze marked write on variable  $x$  from thread  $\theta$  has been observed and  $\langle x \cdot \text{lfrz} \rangle$  to remember if all the freeze markers are observed. To detect the spoilers listed in Section 4.1, and to verify that the threads have guessed the next write positions correctly, it uses the variable  $\langle x \cdot \theta \cdot \text{lwr} \rangle$ . This variable is set when it encounter a write to a frozen variable  $x$  from thread  $\theta$ . We have the set of local variables of the manager,  $\mathbb{L}_{\text{man}} = \{\langle x \cdot \theta \cdot \text{lfrz} \rangle, \langle x \cdot \theta \cdot \text{lwr} \rangle, \langle x \cdot \text{lfrz} \rangle \mid x \in \mathbb{X}, \theta \in \Theta\}$ .

**Algorithm 1:**  $\llbracket x :=_{\text{wb}} d \rrbracket$ 

```

1 assume ( $\neg \langle x \cdot \text{lpfo} \rangle$ )
2 assume ( $\neg \langle x \cdot \text{lfrz} \rangle \vee \langle x \cdot \text{nxt} \rangle$ )
3  $a := d$ 
4 if  $\langle x \cdot \text{ntw} \rangle$  then
5    $a := \langle d \cdot \text{ntw} \rangle$ 
6 if  $\star \wedge (\neg \langle x \cdot \text{lfrz} \rangle)$  then
7    $\langle x \cdot \text{lfrz} \rangle := \text{true}$ 
8    $a := \langle d \cdot \otimes \rangle$ 
9  $\langle x \cdot \theta \rangle :=_{\text{wb}} a$ ; ND()

```

**Algorithm 2:**  $\llbracket x :=_{\text{ntw}} d \rrbracket$ 

```

1 assume ( $\neg \langle x \cdot \text{lpfo} \rangle$ )
2 assume ( $\neg \langle x \cdot \text{lfrz} \rangle \vee \langle x \cdot \text{nxt} \rangle$ )
3  $a := d$  if  $\langle x \cdot \text{lfrz} \rangle$  then
4    $\langle x \cdot \text{ntw} \rangle := \text{true}$ 
5    $a := \langle d \cdot \text{ntw} \rangle$ 
6 if  $\star \wedge (\neg \langle x \cdot \text{lfrz} \rangle)$  then
7    $\langle x \cdot \text{lfrz} \rangle := \text{true}$ 
8    $a := \langle d \cdot \otimes \rangle$ 
9  $\langle x \cdot \theta \rangle :=_{\text{ntw}} a$ ; ND()

```

**Algorithm 3:**  $\llbracket \text{sfence} \rrbracket$ 

```

1  $\bigwedge_{x \in \mathbb{X}} \text{assume} (\neg \langle x \cdot \text{lpfo} \rangle)$ 
2  $\text{flg} := \bigvee_{x \in \mathbb{X}} \langle x \cdot \text{lfo} \rangle$ 
3 if  $\text{flg}$  then
4   sfence;  $\langle \text{sf} \cdot \theta \rangle :=_{\text{wb}} \langle \text{fo} \rangle$ 
5 sfence; ND()

```

**Algorithm 4:**  $\llbracket \text{flush}_{\text{opt}}(x) \rrbracket$ 

```

1 if  $\langle x \cdot \text{nxt} \rangle \wedge \neg \langle x \cdot \text{lpfo} \rangle$  then
2    $\langle x \cdot \text{lfo} \rangle := \text{true}$ 
3 ND()

```

*Translation.* We are now ready to describe translation and the behaviour of the manager, we only provide the implementation relevant to the ntw spoiler and SFW spoiler, the rest of the implementations are similar. The algorithm 1 through 4 are for the implementation of the thread and rest of the algorithms are the implementation of the manager.

*Program code.* We now describe the code to code translations. We describe in the Algorithm 3, how to simulate the **sfence**. In this case, it is firstly checked if the  $\langle x \cdot \text{lpfo} \rangle$  is set, this guards any  $\text{flush}_{\text{opt}}$  from re-ordering with it. In case there was a  $\text{flush}_{\text{opt}}$  that was a potential spoiler (line 2), then the variable  $\langle x \cdot \text{fo} \rangle$  would be set, this also indicates an SFW spoiler. In this case, we write to  $\langle \text{sf} \cdot x \cdot \theta \rangle$  (line 4) as described in the Section 4.5.2. Otherwise the instruction is simulated as it is. Notice the invocation of the procedure ND(), we will describe this in sequel. The simulation of the instruction **flush** and **rmw** are similar, their implementation is guided by how to handle the fl-wr spoiler and rmw-fo-wr-spoilers respectively.

Next we examine Algorithm 1 which provides the translation of wb-writes. Firstly we guard against re-ordering with an  $\text{flush}_{\text{opt}}$ , this is in line 1. We also ensure that we do not write before we speculate the next write, this is done in line 2. If the flag  $\langle x \cdot \text{ntw} \rangle$  is set then we tag the data value with ntw, indicating that this write is preceded by a spoiler. As explained in Section 4.5.2 this ensures that the manager never misses an ntw-type spoiler. It can also nondeterministically choose to set the freeze marker, if it was not set before. Finally it simulates the instruction as a wb write to the variable  $\langle x \cdot \theta \rangle$ .

Algorithm 2, for ntw-writes, is similar but is also tasked with setting the  $\langle x \cdot \text{ntw} \rangle$  if the thread has already used its freeze marker for  $x$ . The rest of the code follows the pattern in Algorithm 1.

The  $\text{flush}_{\text{opt}}(x)$  instruction is described in Algorithm 4. The instruction is simply ignored as long as it is not a potential spoiler. It is deemed a spoiler if it cannot re-ordered before the write to

**Algorithm 0:** ND ()

```

1 if  $\langle x \cdot \text{lfrz} \rangle$  then
2    $\langle \text{nxt} \cdot x \cdot \theta \rangle = \bullet$ 
3    $\langle x \cdot \text{nxt} \rangle := \text{true}$ 
4    $\langle x \cdot \text{lpfo} \rangle := \text{true}$ 
5 if  $\star$  then
6   assume ( $\langle x \cdot \text{lpfo} \rangle$ )
7    $\langle x \cdot \text{lpfo} \rangle := \text{false}$ 
8 if  $\star$  then
9   Reset(lvars)
10 goto  $\ell_{\text{end}}^{\theta}$ 

```

$\blacktriangleleft \text{nxt} \cdot x \blacktriangleright$ . In this case, it is remembered in the  $\blacktriangleleft x \cdot \text{lfo} \blacktriangleright$  flag. A read on  $x$  is now simulated by first reading the  $\langle x, \theta \rangle$  variable and if its value is  $\perp$ , the value is fetched from  $x$  directly.

What is remaining to explain in the thread implementation is the procedure  $\text{ND}()$ . The need for this non-deterministic procedure is as follows. It non-deterministically guesses the position of the violating write (see Section 4.5.2) and implements a write to  $\blacktriangleleft \text{nxt} \cdot x \cdot \theta \blacktriangleright$  (line 1-4) and sets  $\blacktriangleleft x \cdot \text{lpo} \blacktriangleright$  flag to true to allow for the  $\text{fo}$  to re-order. It also non-deterministically stops the thread simulation by jumping to the final halt location of the program. It resets the values in the local variables, the need for this will become clear soon.

*Manager.* The manager is responsible processing the updates. It non-deterministically invokes a procedure from  $\{\text{updRMW}, \text{updWr}, \text{updNxtWr}\}$  and executing it. It does this until values in all the variables are persisted. Finally it also ensures that the persisted values are from  $N$ . The manager also employs a procedure  $\text{testInit}()$  to ensure the last write guarantee explained in Section 4.3. It simply checks if all the thread specific variables in  $\mathbb{V}_{\text{instr}}$  are  $\perp$ , as shown in the Algorithm 8.

**Algorithm 8:**  $\text{testInit}()$ 

```

1 for  $\text{var} \in \mathbb{X}, \theta \in \Theta$  do
2   assume ( $\blacktriangleleft \text{var} \cdot \theta \blacktriangleright = \perp$ )
3   assume ( $\blacktriangleleft \text{var} \cdot \theta \cdot \text{nxt} \blacktriangleright = \perp$ )
4   assume ( $\blacktriangleleft \text{var} \cdot \theta \cdot \text{fl} \blacktriangleright = \perp$ )
5   assume ( $\blacktriangleleft \text{var} \cdot \theta \cdot \text{rmw} \blacktriangleright = \perp$ )
6   assume ( $\blacktriangleleft \theta \cdot \text{sf} \blacktriangleright = \perp$ )

```

**Algorithm 6:**  $\text{updNxtWr}()$ 

```

1 Let  $\text{var} \in \mathbb{X}, \theta \in \Theta$ 
2 assume ( $\neg \blacktriangleleft \text{var} \cdot \text{lwr} \blacktriangleright$ )
3  $\text{rmw}(\blacktriangleleft \text{var} \cdot \theta \cdot \text{nxt} \blacktriangleright, \perp, \perp)$ 

```

**Algorithm 7:**  $\text{updWr}()$ 

```

1 Let  $\text{var} \in \mathbb{X}, \theta \in \Theta$ 
2  $a := \blacktriangleleft \text{var} \cdot \theta \blacktriangleright$ 
3 assume ( $\text{ntw} \notin a$ )
4  $\text{rmw}(\blacktriangleleft \text{var} \cdot \theta \blacktriangleright, a, \perp)$ 
5 if  $\ast \in a$  then
6    $\blacktriangleleft \text{var} \cdot \theta \cdot \text{lfrz} \blacktriangleright := \text{true}$ 
7    $\text{var}' := \text{var}$ 
8 if  $\blacktriangleleft \text{var} \cdot \theta \cdot \text{lfrz} \blacktriangleright = \text{true}$  then
9   assume ( $\blacktriangleleft \text{var} \cdot \text{lfrz} \blacktriangleright$ )
10   $\blacktriangleleft \text{var} \cdot \theta \cdot \text{lwr} \blacktriangleright := \text{true}$ 
11  $\text{var} :=_{\text{atn}} \text{Value}(a)$ 

```

**Algorithm 5:**  $\text{Manager}()$ 

```

1 while  $\text{flg}$  do
2   Let  $\text{func} \in \{\text{updRMW}, \text{updWr}, \text{updNxtWr}\}$ 
3    $\text{testInit}(); \text{func}(); \text{testInit}()$ 
4    $\text{flg} := \text{true}$ 
5   for  $\text{var} \in \mathbb{X}$  do
6      $\blacktriangleleft \text{var} \cdot \text{lfrz} \blacktriangleright := \bigwedge_{\theta \in \Theta} \blacktriangleleft \text{var} \cdot \theta \cdot \text{lfrz} \blacktriangleright$ 
7      $\text{flg} := \text{flg} \wedge \blacktriangleleft \text{var} \cdot \text{lfrz} \blacktriangleright$ 
8    $\bigwedge_{x \in \mathbb{X}} \text{assume}(x' = N(x))$ 
9   Halt

```

The Algorithm 7 describes the procedure  $\text{updWr}$ , this is used by the manager to handle the writes by the threads. The procedure is invoked non-deterministically, upon its invocation, the value in  $\blacktriangleleft x \cdot \theta \blacktriangleright$  is read. It is verified that the value is not an ntw spoiler (line 3). Further if this write is guessed to be one of the last writes by the process, then it is recorded (line 5-7). It is also ensure in line 9 that the current write respects the freeze point and records the same as the next write. The value is finally propagated to  $x$  in line 11.

The procedure  $\text{updNxtWr}$  ensures that  $\blacktriangleleft x \cdot \text{lwr} \blacktriangleright$  is not set while processing any writes to  $\blacktriangleleft \text{nxt} \cdot x \cdot \theta \blacktriangleright$ , this ensures that the guessed position of the next write by any process is well before the actual next write. The  $\text{updRMW}$  procedure implements a handshake protocol.

Finally, we provide the  $\gamma$  as required by the Theorem 4.6. We let  $\gamma = \langle \mathcal{L}, \mathcal{R}_{\text{init}} \rangle$ , where for any thread  $\theta \in \Theta$ ,  $\mathcal{L}(\theta) \ell_{\text{end}}^\theta$  and for the manager,  $\mathcal{L}(\text{man}) = 10$ . Notice that our  $\text{ND}()$  procedure resets the values of the local variable and hence we can afford to let  $\mathcal{R}_{\text{init}}$  in  $\gamma$ .

**REMARK 2.** In our translation, we have made an assumption that every thread writes to every variable. This ensures that the freeze point of each thread is communicated to the manager. We make this assumption to simplify the construction. A given program can easily be transformed to confirm to the assumption by the following procedure. Each thread at the very beginning of its execution, re-writes the initial memory for each variable with the same value by means of an atomic read write.

## 5 UNDECIDABILITY

We reduce the well known *Post correspondence problem* (PCP) [Post 1946], which is known to be undecidable, to crash-free reachability in our model. Given a set of words  $U = \{u_1, \dots, u_\ell\}$  and  $V = \{v_1, \dots, v_\ell\}$  with  $u_i, v_i \in \Sigma^*$ , PCP asks if there is a sequence of indices  $i_1, \dots, i_n \in [1..\ell]$  such that  $u_{i_1} \cdot u_{i_2} \cdot \dots \cdot u_{i_n} = v_{i_1} \cdot v_{i_2} \cdot \dots \cdot v_{i_n}$ . We show how to reduce this problem to crash free reachability problem in our setting. We fix an PCP instance  $U = \{u_1, \dots, u_\ell\}$  and  $V = \{v_1, \dots, v_\ell\}$  over an alphabet  $\Sigma$ .

Towards the reduction, we develop a programs whose crash free reachability is ensured if and only if the PCP has a solution. The program consists of two threads (PCPGen, PCPVerif), as shown in Algorithms 1, 2. The first thread, the generator, proceeds in a sequence of rounds, in each round it guesses an index  $i \in \{1, \dots, \ell\}$ , and ntw-writes the letters of  $u_i$  to the variable  $s$  and  $v_i$  to the variable  $t$  (this is indicated in the lines 5,6). In the program, suppose  $u = a_1 \dots a_n$ , we use  $s :=_{\text{ntw}} u$  to mean the sequence of instructions  $s :=_{\text{ntw}} a_1 \dots s :=_{\text{ntw}} a_n$ . It non-deterministically stops this process after iterating a certain number of times. The other thread, the verifier, reads alternately from  $s$  and  $t$  to verify that the values read match (see line 7,8 in Algorithm-2). But it has to do so without skipping or re-reading any letter. It uses rmws to prevent any re-reads. To prevent skipping, the first thread also wb-writes  $|u_i| + |v_i|$  many 1s to two variables  $x$  and  $y$  (line 7 of Algorithm-1). The verifier, in each iteration, uses rmw instructions to consume these 1's from the two variables, in a manner resembling the alternating bit protocol to prevent skipping. It reads 1 from  $x$  while ensuring  $y = 0$ , reads 1 from  $y$  while ensuring  $x = 0$  (see lines 4, 5 and 9, 10). This way, the number of times PCPVerif program executes the while loop starting in line 3 is exactly same as the number of times PCPGen executes its while loop. Hence, the location 12 is reached in PCPVerif if and only if  $u_{i_1} \dots u_{i_k} = v_{i_1} \dots v_{i_k}$ .

Note that in our construction, a pair of variables are written to using ntw write exclusively and another pair is written to using wb writes exclusively. In fact, the undecidability holds even when we only use wb writes on two variables and ntw write on another. We chose the former reduction since it is simpler. We briefly outline how to obtain the latter reduction. The reduction in this case is from the reachability problem for systems with a FIFO channel. In it, we have a thread that simulates the execution of the given system, it uses its store buffer to simulate the channel. An enqueue operation is simulated as a wb write to variables  $x, y$  existence of two such variables allows us to implement an alternating bit protocol as described above. We additionally have a manager thread which is responsible to ensure that exactly one update updates (of  $x, y$ ) is performed while simulating a dequeue operation. To dequeue an element, the thread guesses the value to be dequeued and writes the value using an ntw write to another variable  $z$ . This allows for synchronization with the manager. The manager thread is used to ensure that exactly one write to the variables  $x, y$  is updated and that the value there is the value that it obtained in  $z$ .

#### Algorithm 1: PCPGen

```

1 Global Vars  $x, y, s, t$ 
2 Local Vars  $i, j, \text{flg} := \text{true}$ 
3 while  $\star$  do
4   Let  $i \in [1, \ell]$ 
5    $s :=_{\text{ntw}} u_i$ 
6    $t :=_{\text{ntw}} v_i$ 
7    $j := |u_i| + |v_i|$ 
8   while  $j > 0$  do
9      $x :=_{\text{wb}} 1$ 
10     $y :=_{\text{wb}} 1$ 
11     $j = j - 1$ 
12  $x :=_{\text{wb}} \#$ 

```

#### Algorithm 2: PCPVerif

```

1 Global Vars  $x, y, s, t$ 
2 Local Vars  $a, b$ 
3 while  $(a \neq \#)$  do
4    $x :=_{\text{rmw}} 1, 0$ 
5    $y :=_{\text{rmw}} 0, 0$ 
6   Let  $b \in \Sigma$ 
7    $s :=_{\text{rmw}} b, 0$ 
8    $t :=_{\text{rmw}} b, 0$ 
9    $y :=_{\text{rmw}} 1, 0$ 
10   $x :=_{\text{rmw}} 0, 0$ 
11   $a := x$ 
12 Halt

```

**THEOREM 5.1.** *Given a program  $\mathcal{P}$ , the crash free reachability problem for it is undecidable.*

## 6 ALTERNATION BOUNDED RUNS – THROUGH EXAMPLES

Given the undecidability results, we consider a restriction of the problem that we refer to as the alternation-bounded reachability problem. In fact, we restrict ourselves to the eTSO semantics since persistence plays no role in the crash free reachability problem. In alternation-bounded reachability, we restrict the runs of the program by putting a bound  $k$  on the number of alternations between wb and ntw instructions. A run of the program will now be a sequence of the form  $\rho_1 \cdot \rho_{1\frac{1}{2}} \cdot \rho_2 \cdot \rho_{2\frac{1}{2}} \cdot \dots \cdot \rho_k \cdot \rho_{k\frac{1}{2}}$ , where the sequences  $\rho_i$  do not contain ntw-writes and the sequences  $\rho_{i\frac{1}{2}}$  do not have wb-writes. We refer to each  $\rho_i$  or  $\rho_{i\frac{1}{2}}$  as a *phase* of the run. We translate the alternation-bounded reachability problem to the *ntw-reachability problem*, i.e., the reachability

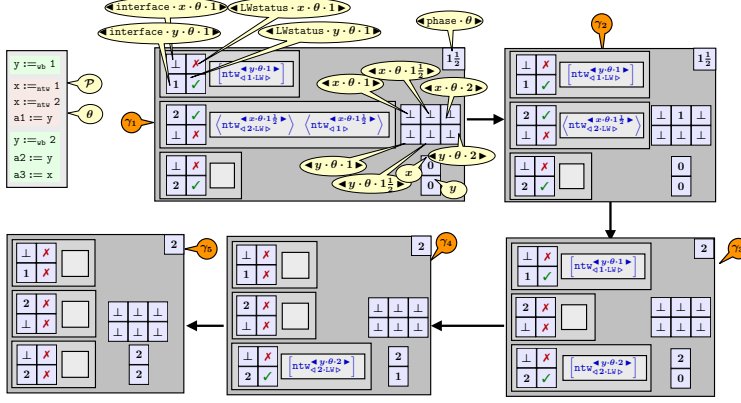


Fig. 13. An input program  $\mathcal{P}$  consisting of a single thread  $\theta$ . The highlighted parts of the code give the instructions executed in the phases 1,  $1\frac{1}{2}$ , and 2 in the execution of  $\mathcal{P}'$ , respectively.

problem where we do not use wb instructions. Given a program  $\mathcal{P}$  and a bound  $k$ , we translate  $\mathcal{P}$  to new program  $\mathcal{P}'$  such the ntw reachability problem for  $\mathcal{P}'$  is equivalent to the  $k$ -alternation bounded reachability problem for  $\mathcal{P}$ . We illustrate the ideas using the program  $\mathcal{P}$  of Fig. 13. For simplicity, we let the program have a single thread  $\theta$  without loops. In general, our framework deals with multiple threads and with loop constructs.

Forbidding wb-write instructions means that we need to simulate such instructions by ntw-instructions. To preserve equivalence with the eTSO-semantics, we need to keep the allowed orderings between messages, i.e., (i) We do not allow wb-messages to overtake each other. We simulate a wb-write instruction by an ntw instruction, and we encapsulate the latter by sf instructions, i.e., we put an sf instruction both before and after the ntw-instruction in our translation. The encapsulation will help prevent forbidden re-orderings of wb-messages. (ii) We allow ntw-messages to overtake wb-messages even on the same variable. To that end, we simulate each phase by a separate thread in  $\mathcal{P}'$ . Since the threads have different buffers, the respective messages may now overtake each other. (iii) We do not allow wb-messages to overtake ntw-messages on the same variable; nor do we allow them to overtake sf- or fl-messages. To that end, we add a manager thread, and implement a protocol, that we refer to as the *interface protocol*. The protocol lets the manager, in collaboration with the other threads, ensure that write messages are updated to the memory in the correct order, and that read instructions see the correct values. In the rest of the section, we describe how we implement the interface protocol, by giving the set of threads, variables, data domains, updates and reads.

*Threads.* For the program of Fig. 13, we will consider a 2-alternating run in which  $\theta$  executes the following instructions: (i) the first instruction in phase 1, (iii) the next three instructions in phase  $1\frac{1}{2}$ , and (iii) the last three instructions in phase 2. We simulate each phase  $i$  of  $\theta$  in  $\mathcal{P}'$  by a separate thread which we call the  $\langle \theta, i \rangle$ -thread, i.e., the threads are  $\langle \theta, 1 \rangle$ ,  $\langle \theta, 1\frac{1}{2} \rangle$ , and  $\langle \theta, 2 \rangle$ . The program  $\mathcal{P}'$  runs the instructions of the different phases one after one. The current phase of  $\theta$  is given by the shared variable  $\blacktriangleleft \text{phase} \cdot \theta \blacktriangleright$ . Although the instructions are run sequentially, updating the messages belonging to different phases of the same thread may now interleave since we are using separate threads to simulate them. Therefore, we need to ensure that message updates and the values seen by read instructions faithfully mimic the behavior of the input program  $\mathcal{P}$ . We do this by guessing the *interfaces*, i.e., the memory contents between the various phases of the same thread, and then run a protocol, the *interface protocol*, that ensures that the inter-phase interaction

is carried out correctly. We implement the protocol using a *manager*, which is an extra thread to which we divert the traffic between the threads and the memory (similar to the case of Section 4).

*Interfaces.* For each phase  $i$ , thread  $\theta$ , and variable  $x$ , we guess the value of the last write operation on  $x$  by  $\theta$  during  $i$ . For instance, in Fig. 13, the  $\langle \theta, 1 \rangle$ -interface is defined by  $x = \perp$  and  $y = 1$ . These values are given by the shared variables  $\blacktriangleleft \text{interface} \cdot x \cdot \theta \cdot 1 \blacktriangleright$  and  $\blacktriangleleft \text{interface} \cdot y \cdot \theta \cdot 1 \blacktriangleright$ , respectively. The above values tell us that (i)  $\theta$  will not perform any write instruction on  $x$  during phase 1, and (ii) that the last write instruction of  $\theta$  on  $y$ , during phase 1, assigns the value 1 to  $y$ . The other interfaces are interpreted similarly.

*Variables and Data.* The variable  $\blacktriangleleft \text{phase} \cdot \theta \blacktriangleright$  gives the current phase of  $\theta$ . In  $\gamma_1$ , the current phase is  $1\frac{1}{2}$ , which means that the thread  $\langle \theta, 1 \rangle$  has already executed its (only) instruction (it did so in phase 1, which we do not show in the figure.) It has put the corresponding message in the buffer. To simulate a wb-write instruction by an ntw instruction, we encapsulate it by sf instructions, i.e., we put an sf instruction both before and after it in our translation. The encapsulation will help prevent forbidden re-orderings. To simplify the notation, we replace the sequence  $\langle \text{sf} \rangle \langle \text{ntw}_{\beta}^{\alpha} \rangle \langle \text{sf} \rangle$  by  $\left[ \text{ntw}_{\beta}^{\alpha} \right]$ , for any  $\alpha$  and  $\beta$ . The enriched message  $\left[ \text{ntw}_{\blacktriangleleft 1 \cdot \text{LW} \blacktriangleright}^{\blacktriangleleft y \cdot \theta \cdot 1 \blacktriangleright} \right]$  tells us that  $\theta$  has performed wb-write on  $y$  in phase 1. The written value is 1; furthermore, the LW attribute tells us is the last write message on  $y$  by thread  $\theta$ . Also, in  $\gamma_1$ , the thread  $\langle \theta, 1\frac{1}{2} \rangle$  has performed its first two instructions and has generated the corresponding messages  $\left[ \text{ntw}_{\blacktriangleleft 2 \cdot \text{LW} \blacktriangleright}^{\blacktriangleleft x \cdot \theta \cdot 1\frac{1}{2} \blacktriangleright} \right] \left[ \text{ntw}_{\blacktriangleleft 1 \blacktriangleright}^{\blacktriangleleft x \cdot \theta \cdot 1\frac{1}{2} \blacktriangleright} \right]$ . The first message does not have the flag LW in its value since it is not the last write by the thread  $\langle \theta, 1\frac{1}{2} \rangle$  on  $x$ . In  $\gamma_3$ , we have entered phase 2 and the thread  $\langle \theta, 2 \rangle$  has executed its write instruction resulting in the message  $\left[ \text{ntw}_{\blacktriangleleft 2 \cdot \text{LW} \blacktriangleright}^{\blacktriangleleft y \cdot \theta \cdot 2 \blacktriangleright} \right]$ .

*Correct Updates.* We need to guarantee that updates to the memory are performed in the correct order. We need to ensure that, within any thread, the following properties are satisfied: (i) For any given variable  $x$ , the ntw write messages are not re-ordered within the same thread. (ii) the wb-write messages are not re-ordered. We let message updates go through the manager. For instance, from in the transition from  $\gamma_1$  to  $\gamma_2$ , we transfer the message  $\left[ \text{ntw}_{\blacktriangleleft 1 \blacktriangleright}^{\blacktriangleleft x \cdot \theta \cdot 1\frac{1}{2} \blacktriangleright} \right]$  to the variable  $\blacktriangleleft x \cdot \theta \cdot 1\frac{1}{2} \blacktriangleright$ . This enables the manager to inspect the value before transferring the correct value to the memory (the last step is not shown in the figure; in  $\gamma_3$ , we have already moved the next message of the buffer to get the value  $x = 2$  in the memory). We provide, for each variable  $x$ , thread  $\theta$ , and phase  $i$ , the shared variable  $\blacktriangleleft \text{LWstatus} \cdot x \cdot \theta \cdot i \blacktriangleright$ . The latter is a Boolean flag that tells whether the last write message on  $x$  generated by the thread  $\langle \theta, i \rangle$  is still in the buffer (the value true) or has left the buffer (the value false). For instance, in  $\gamma_1$  of Fig. 13, the last write on  $y$  in phase 1 is still in the buffer, whence the value true of the corresponding. The value is false for  $x$  since there is no write on  $x$  in phase 1. The manager changes the value of the LWstatus-flag to false when it receives a write message whose value contains the LW-flag. For instance, in  $\gamma_3$ , the manager has switched the value of the flag  $\blacktriangleleft \text{LWstatus} \cdot x \cdot \theta \cdot 1\frac{1}{2} \blacktriangleright$  since it has received the the message  $\left[ \text{ntw}_{\blacktriangleleft \text{LW} \cdot 2 \blacktriangleright}^{\blacktriangleleft x \cdot \theta \cdot 1\frac{1}{2} \blacktriangleright} \right]$  (which has LW as part of its value). In this manner, the manager can record whether a given buffer contains an outstanding write message on a given variable and approve variable updates only if they do not violate the semantics. As an example, in  $\gamma_3$ , the manager would not accept to update the message  $\left[ \text{ntw}_{\blacktriangleleft 2 \cdot \text{LW} \blacktriangleright}^{\blacktriangleleft y \cdot \theta \cdot 2 \blacktriangleright} \right]$  from the buffer of the thread  $\langle \theta, 2 \rangle$  since the buffer of the thread  $\langle \theta, 1 \rangle$  still contains a the message  $\left[ \text{ntw}_{\blacktriangleleft 1 \cdot \text{LW} \blacktriangleright}^{\blacktriangleleft y \cdot \theta \cdot 1 \blacktriangleright} \right]$ . Such an update would correspond to re-ordering two ntw-writes messages on the same variable, which is forbidden in the eTSO semantics. On the other

hand, it allowed updating the messages of phase  $1\frac{1}{2}$  since they represent wb-messages overtaking ntw messages on different variables, which is permitted under eTSO.

*Correct Reads.* Similarly to updates, the interface protocol allows to read the correct values. Assume that the thread  $\theta$  needs to read the variable  $x$  value during phase  $i$ . Then, the thread  $\langle \theta, i \rangle$  in  $\mathcal{P}'$  will perform the following sequence of actions: (i) It checks whether it has performed a write on  $\blacktriangleleft x \cdot \theta \cdot i \blacktriangleright$  (this information is maintained locally in  $\langle \theta, i \rangle$ ). In such a case, it reads the value. (ii) Otherwise, it inspects the value of the last write on  $x$  in the previous phase  $j = i - \frac{1}{2}$ . If  $\blacktriangleleft \text{interface} \cdot x \cdot \theta \blacktriangleright j = \perp$ , then the previous phase never performed a write on  $x$ , and hence we continue to the phase  $i - 1$  (until we possibly reach the memory). If  $\blacktriangleleft \text{interface} \cdot x \cdot \theta \blacktriangleright j \neq \perp$ , then  $\theta$  has performed a write on  $x$  during phase  $j$ . We check whether this last written value is still in the buffer. More precisely,  $\langle \theta, i \rangle$  checks the value of the variable of  $\blacktriangleleft \text{LWstatus} \cdot x \cdot \theta \cdot j \blacktriangleright$ . If that variable contains a value different from  $\perp$ , it fetches the value from  $\blacktriangleleft \text{interface} \cdot x \cdot \theta \cdot j \blacktriangleright$ ; otherwise, it reads the variable's value in the memory. For instance, in  $\gamma_2$ , if  $\theta$  executes the instruction  $a1 := y$ , it will read the value of  $\blacktriangleleft \text{interface} \cdot y \cdot \theta \cdot 1 \blacktriangleright$  which is 1. If  $\theta$  executes the instruction  $a2 := y$  in  $\gamma_4$  it sees its own last write on  $y$ , which is 2. If  $\theta$  executes the instruction  $a3 := x$  in  $\gamma_5$  it needs to go all the way to the memory to find the value 2.

Finally we invoke the result in [Atig et al. 2012] which states that reachability problem under PSO which is known to be decidable to obtain the following theorem.

**THEOREM 6.1.** *Given a program  $\mathcal{P}$ , the alternation bounded reachability problem for it is decidable.*

## 7 CONCLUSION

We have investigated the decidability of the reachability problem under the Intel-x86 semantics defined in [Raad et al. 2022].

We have first provided a reduction that allows to take into account persistency without using an unbounded memory to model the persistency stage. The reduction is based on a program instrumentation that augment the given program with an extra finite-state thread, which allows to reduce the original reachability problem to verifying reachability under the consistency model only. The reduction is valid in particular when the consistency model is SC, TSO, or PSO. This allows in particular to provide a simpler decidability proof for the reachability problem under PTSO than the one in [Abdulla et al. 2021a] that uses an unbounded buffer for the persistency stage. An interesting issue is to investigate the class of storage systems for which such a finite-memory instrumentation is possible to encode the persistency semantics.

We have also shown that mixing operations obeying to various consistency models with decidable reachability problems may lead to undecidability. However, we have provided for the case we consider a condition under which verifying reachability becomes decidable: bounding the number of alternation between wb writes and ntw's in computations. This result is interesting as it provides a complete parametrized bounded analysis schema for bug finding in the setting we consider. Other types of restrictions could be investigated, based on commonly adopted patterns for the use of operations on different memory types and for their interactions.

## ACKNOWLEDGMENTS

This research was partially supported by Infosys (India), DST-VR funded Indo Swedish Project P-04/2019, the MATRICS grant (MTR/2022/000312), the Swedish Research Council (Sweden) and the Project AdeCoDS of the French National Research Agency ANR (France).

## REFERENCES

- Parosh Aziz Abdulla. 2010. Well (and better) quasi-ordered transition systems. *Bull. Symb. Log.* 16, 4 (2010), 457–515. <https://doi.org/10.2178/bsl/1294171129>
- Parosh Aziz Abdulla, Jatin Arora, Mohamed Faouzi Atig, and Shankara Narayanan Krishna. 2019. Verification of programs under the release-acquire semantics. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019.*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 1117–1132. <https://doi.org/10.1145/3314221.3314649>
- Parosh Aziz Abdulla, Mohamed Faouzi Atig, Ahmed Bouajjani, Egor Derevenetc, Carl Leonardsson, and Roland Meyer. 2020b. Safety Verification under Power. In *NETYS 2020 (Lecture Notes in Computer Science)*. Springer.
- Parosh Aziz Abdulla, Mohamed Faouzi Atig, Ahmed Bouajjani, K. Narayan Kumar, and Prakash Saivasan. 2021a. Deciding reachability under persistent x86-TSO. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–32. <https://doi.org/10.1145/3434337>
- Parosh Aziz Abdulla, Mohamed Faouzi Atig, Ahmed Bouajjani, K. Narayan Kumar, and Prakash Saivasan. 2022. Verifying Reachability for TSO Programs with Dynamic Thread Creation. In *Networked Systems - 10th International Conference, NETYS 2022, Virtual Event, May 17-19, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13464)*, Mohammed-Amine Koulali and Mira Mezini (Eds.). Springer, 283–300. [https://doi.org/10.1007/978-3-031-17436-0\\_19](https://doi.org/10.1007/978-3-031-17436-0_19)
- Parosh Aziz Abdulla, Mohamed Faouzi Atig, Ahmed Bouajjani, and Tuan Phong Ngo. 2016. The Benefits of Duality in Verifying Concurrent Programs under TSO. In *CONCUR (LIPIcs, Vol. 59)*. Schloss Dagstuhl, 5:1–5:15.
- Parosh Aziz Abdulla, Mohamed Faouzi Atig, Ahmed Bouajjani, and Tuan Phong Ngo. 2018. A Load-Buffer Semantics for Total Store Ordering. *Logical Methods in Computer Science* 14, 1 (2018). [https://doi.org/10.23638/LMCS-14\(1:9\)2018](https://doi.org/10.23638/LMCS-14(1:9)2018)
- Parosh Aziz Abdulla, Mohamed Faouzi Atig, Florian Furbach, Adwait Amit Godbole, Yacoub G. Hendi, Shankara Narayanan Krishna, and Stephan Spengler. 2023. Parameterized Verification under TSO with Data Types. In *Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22-27, 2023, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 13993)*, Sriram Sankaranarayanan and Natasha Sharygina (Eds.). Springer, 588–606. [https://doi.org/10.1007/978-3-031-30823-9\\_30](https://doi.org/10.1007/978-3-031-30823-9_30)
- Parosh Aziz Abdulla, Mohamed Faouzi Atig, Adwait Godbole, Shankara Narayanan Krishna, and Viktor Vafeiadis. 2021b. The Decidability of Verification under PS 2.0. In *Programming Languages and Systems - 30th European Symposium on Programming, ESOP 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12648)*, Nobuko Yoshida (Ed.). Springer, 1–29. [https://doi.org/10.1007/978-3-030-72019-3\\_1](https://doi.org/10.1007/978-3-030-72019-3_1)
- Parosh Aziz Abdulla, Mohamed Faouzi Atig, Magnus Lång, and Tuan Phong Ngo. 2015b. Precise and Sound Automatic Fence Insertion Procedure under PSO. In *Networked Systems - Third International Conference, NETYS 2015, Agadir, Morocco, May 13-15, 2015, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 9466)*, Ahmed Bouajjani and Hugues Fauconnier (Eds.). Springer, 32–47. [https://doi.org/10.1007/978-3-319-26850-7\\_3](https://doi.org/10.1007/978-3-319-26850-7_3)
- Parosh Aziz Abdulla, Mohamed Faouzi Atig, and Ngo Tuan Phong. 2015a. The Best of Both Worlds: Trading Efficiency and Optimality in Fence Insertion for TSO. In *ESOP*.
- Parosh Aziz Abdulla, Mohamed Faouzi Atig, and Rojin Rezvan. 2020a. Parameterized verification under TSO is PSPACE-complete. *PACMPL* 4, POPL (2020).
- Parosh Aziz Abdulla, Karlis Cerans, Bengt Jonsson, and Yih-Kuen Tsay. 1996. General Decidability Theorems for Infinite-State Systems. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, USA, July 27-30, 1996*. IEEE Computer Society, 313–321. <https://doi.org/10.1109/LICS.1996.561359>
- Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. 2010. On the verification problem for weak memory models. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, Manuel V. Hermenegildo and Jens Palsberg (Eds.). ACM, 7–18. <https://doi.org/10.1145/1706299.1706303>
- Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. 2012. What’s Decidable about Weak Memory Models?. In *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012, Proceedings (Lecture Notes in Computer Science, Vol. 7211)*, Helmut Seidl (Ed.). Springer, 26–46. [https://doi.org/10.1007/978-3-642-28869-2\\_2](https://doi.org/10.1007/978-3-642-28869-2_2)
- Ahmed Bouajjani, Egor Derevenetc, and Roland Meyer. 2013. Checking and Enforcing Robustness against TSO. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013, Proceedings (Lecture Notes in Computer Science, Vol. 7792)*, Matthias Felleisen and Philippa Gardner (Eds.). Springer, 533–553. [https://doi.org/10.1007/978-3-642-37036-6\\_29](https://doi.org/10.1007/978-3-642-37036-6_29)
- Ahmed Bouajjani, Roland Meyer, and Eike Möhlmann. 2011. Deciding Robustness against Total Store Ordering. In *Automata, Languages and Programming - 38th International Colloquium, ICALP 2011, Zurich, Switzerland, July 4-8, 2011, Proceedings*,

- Part II (Lecture Notes in Computer Science, Vol. 6756), Luca Aceto, Monika Henzinger, and Jiri Sgall (Eds.). Springer, 428–440. [https://doi.org/10.1007/978-3-642-22012-8\\_34](https://doi.org/10.1007/978-3-642-22012-8_34)
- Egor Derevenetc and Roland Meyer. 2014. Robustness against Power is PSpace-complete. In *Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part II* (Lecture Notes in Computer Science, Vol. 8573), Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias (Eds.). Springer, 158–170. [https://doi.org/10.1007/978-3-662-43951-7\\_14](https://doi.org/10.1007/978-3-662-43951-7_14)
- Michael Emmi, Shaz Qadeer, and Zvonimir Rakamaric. 2011. Delay-bounded scheduling. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 411–422. <https://doi.org/10.1145/1926385.1926432>
- Alain Finkel and Philippe Schnoebelen. 2001. Well-structured transition systems everywhere! *Theor. Comput. Sci.* 256, 1-2 (2001), 63–92. [https://doi.org/10.1016/S0304-3975\(00\)00102-X](https://doi.org/10.1016/S0304-3975(00)00102-X)
- Artem Khyzha and Ori Lahav. 2021. Taming x86-TSO persistency. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–29. <https://doi.org/10.1145/3434328>
- Shankaranarayanan Krishna, Adwait Godbole, Roland Meyer, and Soham Chakraborty. 2022. Parameterized Verification under Release Acquire is PSPACE-complete. In *PODC '22: ACM Symposium on Principles of Distributed Computing, Salerno, Italy, July 25 - 29, 2022*, Alessia Milani and Philipp Woelfel (Eds.). ACM, 482–492. <https://doi.org/10.1145/3519270.3538445>
- Ori Lahav and Udi Boker. 2020. Decidable verification under a causally consistent shared memory. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 211–226. <https://doi.org/10.1145/3385412.3385966>
- Ori Lahav and Udi Boker. 2022. What's Decidable About Causally Consistent Shared Memory? *ACM Trans. Program. Lang. Syst.* 44, 2 (2022), 8:1–8:55. <https://doi.org/10.1145/3505273>
- Ori Lahav and Roy Margalit. 2019. Robustness against release/acquire semantics. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 126–141. <https://doi.org/10.1145/3314221.3314604>
- Roy Margalit and Ori Lahav. 2021. Verifying observational robustness against a c11-style memory model. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–33. <https://doi.org/10.1145/3434285>
- Emil L. Post. 1946. A variant of a recursively unsolvable problem. *Bull. Amer. Math. Soc.* 52 (1946), 264–268.
- Shaz Qadeer and Jakob Rehof. 2005. Context-Bounded Model Checking of Concurrent Software. In *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3440)*, Nicolas Halbwachs and Lenore D. Zuck (Eds.). Springer, 93–107. [https://doi.org/10.1007/978-3-540-31980-1\\_7](https://doi.org/10.1007/978-3-540-31980-1_7)
- Azalea Raad, Luc Maranget, and Viktor Vafeiadis. 2022. Extending Intel-x86 consistency and persistency: formalising the semantics of Intel-x86 memory types and non-temporal stores. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–31. <https://doi.org/10.1145/3498683>
- Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis. 2020. Persistency semantics of the Intel-x86 architecture. *Proc. ACM Program. Lang.* 4, POPL (2020), 11:1–11:31. <https://doi.org/10.1145/3371079>

Received 2023-11-16; accepted 2024-03-31