What's Decidable about Weak Memory Models? (Extended Version)

M. F. Atig[1], A. Bouajjani[2], S. Burckhardt[3], and M. Musuvathi[3]

[1] Uppsala University, Sweden, `mohamed_faouzi.atig@it.uu.se`
[2] LIAFA, Paris Diderot Univ. & CNRS, France, `{atig,abou}@liafa.jussieu.fr`
[3] Microsoft Research Redmond, USA, `{sburckha,madanm}@microsoft.com`

Abstract. We investigate the decidability of the state reachability problem in finite-state programs running under weak memory models. In [3], we have shown that this problem is decidable for TSO and its extension with the write-to-write order relaxation, but beyond these models nothing is known to be decidable. Moreover, we have shown that relaxing the program order by allowing reads or writes to overtake reads leads to undecidability.
In this paper, we refine these results by sharpening the (un)decidability frontiers on both sides. On the positive side, we introduce a new memory model NSW (for non-speculative writes) that extends TSO with the write-to-write relaxation, the read-to-read relaxation, and support for partial fences. We present a backtrack-free operational model for NSW, and prove that it does not allow causal cycles (thus barring pathological out-of-thin-air effects). On the negative side, we show that adding the read-to-write relaxation to TSO causes undecidability, and that adding non-atomic writes to NSW also causes undecidability.
Our results establish that NSW is the first known hardware-centric memory model that is relaxed enough to permit both delayed execution of writes and early execution of reads for which the reachability problem is decidable.

1 Introduction

The memory consistency model (or simply, the memory model) of a shared-memory multiprocessor is a low-level programming abstraction that defines when and in what order writes performed by one processor become visible to other processors. The simplest memory model, sequential consistency [16], requires that the operations performed by the processors should appear as if these operations are interleaved in a consistent global order. Despite its simplicity and appeal, most contemporary hardware platforms support Weak (relaxed) Memory Models for performance reasons [2, 13].

The effects of weal memory models can be counterintuitive and difficult to understand even for very small programs. Not surprisingly, relaxed memory models are an active research area today. Much progress has been made to aid programmers, in the form of verification or model-checking algorithms [8, 15, 28, 17, 4], testing tools [11, 19], analyses that check whether programs are exposed to specific relaxations [7, 9, 21], fence insertion tools [14, 15, 18], verified compilation [10, 26, 25], and formal models that closely approximate commercial multiprocessors [22, 24, 27].

Nevertheless, many foundational questions about weak memory models remain. For instance, given a finite-state concurrent program under weak memory model, what is the complexity of deciding if a particular erroneous state can be reached? What is the most relaxed model for which the safety or liveness verification problem is decidable? Understanding the answers to these questions is necessary when designing automated analysis tools for low-level system software exposed to weak memory models.

$\mathsf{w} \rightarrow \mathsf{r}$ (Write-to-read order). The effect of a write may be delayed past a subsequent read. This relaxation enables the use of per-processor *write buffers*. Specifically, when executing a write, a processor may buffer the value to be written in its local buffer and continue executing before the buffered value becomes globally visible.

$\mathsf{w} \rightarrow \mathsf{w}$ (Write-to-write order). A processor may swap the order of two writes. For instance, if using a write buffer as described above, writes may exit the buffer in a different order than they entered.

$\mathsf{r} \rightarrow \mathsf{r}/\mathsf{w}$ (Read-to-read/write order). A processor may change the order of a read and a subsequent read or write. This enables out-of-order execution techniques that help to hide latency of memory accesses. We further distinguish between $\mathsf{r} \rightarrow \mathsf{r}$ (read-to-read) and $\mathsf{r} \rightarrow \mathsf{w}$ (read-to-write) relaxations.

RLWE (Read local writes early). A processor may read its own writes even if they are not globally visible yet (i.e. before the exit the buffer). For example, if a processor executes a read from a location for which there are pending writes in the local buffer, it can immediately forward the value of the last such write from the buffer to the read.

RRWE (Read remote writes early). A processor may read other processors' writes even if they are not globally visible yet. For example, a write in a local buffer may be directly forwarded to some remote processors before it exits the buffer.

RWF (read-read and write-write fences). A processor may issue a read-read (write-write) fence to prevent reordering of reads (writes) that precede the fence with reads (writes) that succeed it.

Fig. 1. Definition Acronyms that represent relaxations/features, following the terminology in [2].

In prior work [3], we have presented some early decidability results for relaxed memory models. We have shown that the reachability problem for finite state machines is decidable for two simple relaxations to SC: (1) the TSO (total store order) model that allows write-to-read relaxation, and (2) a memory model that extends TSO with the write-to-write relaxation. In addition, the prior work also shows that the reachability problem is undecidable for a memory model that allows all four combinations of read and write relaxations. In this paper, we refine these results with a precise study of relaxations that lead to the undecidability of memory models. Fig. 1 describes the relaxations studied in this paper and Fig. 2 summarizes our results and comparison with prior work.

Our results show (perhaps surprisingly) that relaxations that are commonly considered as counter-intuitive by programmers coincide with those that lead to undecidability. For instance, we show that adding the read-to-write relaxation to TSO (total store order) results in an undecidable memory model. In such a relaxation, a processor eagerly makes a write visible to other processors before a prior read has completed. Such speculative writes can result in causal cycles, a well known memory model hazard [12, 20]. On the other hand, a memory model that avoids this relaxation but otherwise remains

Memory Model	Name	Reach. Problem
$\{\mathsf{w} \to \mathsf{r},\ \text{RLWE}\}$	TSO	decidable [3]
TSO$\cup\{\mathsf{w} \to \mathsf{w}\}$	-	decidable [3]
TSO$\cup\{\mathsf{w} \to \mathsf{w},\ \text{RWF}\}$	PSO	decidable [**new**]
PSO$\cup\{\mathsf{r} \to \mathsf{r}\}$	NSW	decidable [**new**]
TSO$\cup\{\mathsf{r} \to \mathsf{r}/\mathsf{w}\}$	-	undecidable [3]
TSO$\cup\{\mathsf{r} \to \mathsf{w}\}$	-	undecidable [**new**]
NSW$\cup\{\text{RRWE}\}$	-	undecidable [**new**]

Fig. 2. Summary of previously known and unknown results about the decidability of the reachability problem on weak memory models. The acronyms are defined in Fig. 1.

general by allowing read-to-read, write-to-read, and write-to-write relaxations together with read-read and write-write fences is actually decidable. We call this memory model NSW (non speculative writes) and study its properties. Finally, we show that adding non-atomic writes to NSW results in an undecidable memory model. Such non-atomic writes can lead to counter-intuitive IRIW (independent reads of independent writes) effects [6].

Memory designers have to reconcile the conflicting goals of being weak enough to allow performance optimizations while simple enough for programmers to understand. We hope that characterizing decidability of memory models will help designers to make the right performance/programmability tradeoff. As an example, the Power memory model specification [24] allows the read-to-write relaxation. However, extensive experimentation performed on real hardware implementations have not found evidence of this relaxation [24]. This suggests that while this optimization is currently not implemented and not crucial for performance, hardware designers would still like to keep the specification flexible for future needs. Our results quantifies the cost of this flexibility.

Along the same vein, we show that NSW, which is the most relaxed model known to be decidable, exhibits the following desirable properties:

– NSW enables significant optimizations; specifically, (1) it permits a write to be moved down (later) in the program execution past any other read or write (by delaying it in a buffer), and (2) it permits reads to be moved up (earlier) in the program execution, before any other read or write (even before a read on whose value it depends).
– The performance impact of prohibiting the read-to-write relaxation (which is the only ordering relaxation remaining in NSW) can be ameliorated by write buffers: even if we disallow writes to become visible to other processors (i.e. exit the write buffer) before all preceding reads have completed, we may still allow writes to enter into the buffer while older reads are still pending.
– Since NSW does not permit writes to become visible to other processors before all older loads by the same processor have completed, causal cycles and out-of-thin-air behaviors are impossible. We formalize and prove this fact in Section 3.7.
– In operational memory models, reordering of dependent memory accesses is usually modeled by nondeterministically guessing the read value and validating it later.

In some sense, such models are not very constructive as they may require backtracking if a guess can not be validated later on. We discovered a way to eliminate all such guesses from our operational model for NSW, obtaining an alternative operational model that is backtrack-free (Section 5).

– The relaxations in NSW do not depend on any notion of data/control-dependencies. Not only does this greatly simplify the formalism, but it also avoids subtle soundness problems with compiler optimizations that may break dependencies [5].

To establish that the state reachability problem for NSW is decidable, we proceed in two steps. First, we define an operational model for NSW where reads do not need to be stored, but still allowing the precise simulation of all their possible reorderings due to the read-to-read relaxation (section 5). The key idea for tackling this issue consists, roughly speaking, in using a buffer storing the history of all the past memory states, in addition to informations about the most recent value read by each process on each variable. The whole model has actually three levels of buffers, each of them related to one of the considered relaxations (write-to-write, write-to-read, and finally read-to-read). We think that this step has its own interest from the point of view of modeling and of understanding the effects of each of the considered relaxations, regardless from the decidability issue. Then, in a second step (section 6), we prove that the defined operational model can be transformed, while preserving state reachability, into a system that is monotonic w.r.t. a well quasi-ordering on the set of its configurations. This allows to deduce that the model has a decidable state reachability problem, using [1]. Both steps are nontrivial and are based on new and quite subtile constructions.

2 Preliminary definitions and notations

Let $k \in \mathbb{N}$ such that $k \geq 1$. Then, we denote by $[k]$ the set $\{1,\ldots,k\}$. Let Σ be a finite alphabet. We denote by Σ^* the set of all *words* over Σ, and by ε the empty word. The length of a word $w \in \Sigma^*$ is denoted by $\mathit{length}(w)$. (We assume that $\mathit{length}(\varepsilon) = 0$.) For every $i \in [\mathit{length}(w)]$, let $w(i)$ denote the symbol at position i in w. For $a \in \Sigma$ and $w \in \Sigma^*$. We write $a \in w$ if a appears in w, i.e., $\exists i \in [\mathit{length}(w)]$ such that $a = w(i)$.

Given a sub-alphabet $\Theta \subseteq \Sigma$ and a word $u \in \Sigma^*$, we denote by $u|_\Theta$ the *projection* of u over Θ, i.e., the word obtained from u by erasing all the symbols that are not in Θ.

Let $k \geq 1$ be an integer and E be a set. Let $\mathbf{e} = (e_1,\ldots,e_k) \in E^k$ be a k-dim vector over E. For every $i \in [k]$, we use $\mathbf{e}[i]$ to denote the i-th component of $\mathbf{e}$ (i.e., $\mathbf{e}[i] = e_i$). For every $j \in [k]$ and $e' \in E$, we denote by $\mathbf{e}[j \leftarrow e']$ the k-dim vector $\mathbf{e}'$ over E defined as follows: $\mathbf{e}'[j] = e'$ and $\mathbf{e}'[l] = \mathbf{e}[l]$ for all $l \neq j$.

Let E and F be two sets. We denote by $[E \rightarrow F]$ the set of all mappings from E to F. Assume that E is finite and that $E = \{e_1,\ldots,e_k\}$ for some integer $k \geq 1$. Then, we sometimes identify a mapping $\mathbf{g} \in [E \rightarrow F]$ with a k-dim vector over F (i.e., we consider that $\mathbf{g} \in F^k$ with $\mathbf{g}[i] = e_i$ for all $i \in [k]$).

3 Weak Memory Models

3.1 Shared memory concurrent systems

Let D be a finite data domain, and $X = \{x_1, \ldots, x_m\}$ a finite set of variables valued in D. Let M denote the set D^m, i.e., the set of all possible valuations of the variables in X.

For a given finite set of process identities I, let $\Omega(I,X,D)$ be the set of operations of the form: (1) *"no operation"*: nop, (2) *read*: $\mathsf{r}(i,j,d)$, (3) *write*: $\mathsf{w}(i,j,d)$, (4) *atomic read-write*: $\mathsf{arw}(i,j,d,d')$, (5) *read_fence*: $\mathsf{rfence}(i)$, and (6) *write_fence*: $\mathsf{wfence}(i)$, where $i \in I$, $j \in [m]$, and $d, d' \in D$.

A *concurrent system* over D and X is a tuple $\mathcal{N} = (\mathcal{P}_1, \ldots, \mathcal{P}_n)$ such that for every $i \in [n]$, $\mathcal{P}_i = (P_i, \Delta_i)$ is a finite-state process where (1) P_i is a finite set of control states, and (2) $\Delta_i \subseteq P_i \times \Omega(\{i\}, X, D) \times P_i$ is a finite set of labeled transitions.

Let $\mathbf{P} = P_1 \times \ldots \times P_n$. For convenience, we write $p \xrightarrow{op}_i p'$ instead of $(p, op, p') \in \Delta_i$, for any $p, p' \in P_i$ and $op \in \Omega(\{i\}, X, D)$. We denote by $\Omega(\mathcal{N}) \subseteq \Omega([n], X, D)$ the set of operations used in $\mathcal{N}$. Given an operation $\omega = op(i,j,d)$ with $op \in \{\mathsf{r}, \mathsf{w}\}$, $i \in [n]$, $j \in [m]$, and $d \in D$, let $proc(\omega) = i$, $var(\omega) = j$, and $data(\omega) = d$.

3.2 Memory models

The executions of a concurrent system are obtained by interleaving the executions of operations issued by its different processes. In the Sequential Consistency (SC) model, the program order between operations of a same process is preserved. Relaxations of this program order lead to the definition of various weak memory models. However, fences (called also barriers) can be used in order to impose the serialization of some operations at some points of the execution. An operation $\mathsf{arw}(i,j,d,d')$ is equivalent to the atomic execution of the sequence $\mathsf{r}(i,j,d);\mathsf{w}(i,j,d')$, with the additional assumption that this operation is never reordered with any other operation of the same process. Therefore, this operation can emulate a fence w.r.t all kind of operations, i.e., two operations by the same process occurring before and after (in program order) the fence cannot be swapped. The operation $\mathsf{wfence}(i)$ (resp. $\mathsf{rfence}(i)$) is a fence for write (resp. read) operations only, i.e., writes (resp. reads) that occur before and after a write_fence (resp. read_fence) cannot be swapped.

3.3 A Semantics based on Rewrite Rules

We consider memory models corresponding to a set of program order relaxations defined by permutation rules between the operations. Given read/write operations $op_1, op_2 \in \{\mathsf{w}, \mathsf{r}\}$, relaxing the $\mathbf{op_1}$ **to** $\mathbf{op_2}$ order consists in allowing that operations of the class op_2 are allowed to overtake operations of the class op_1 in a computation, provided that these operations are issued by the same process, and that they are acting on *different* variables. This corresponds to defining a set of rewrite rules:

$$op_1(i,j,d)op_2(i,k,d') \hookrightarrow op_2(i,k,d')op_1(i,j,d) \tag{1}$$

for any $i \in [n]$, $j, k \in [m]$, and $d, d' \in D$.

In addition to permutations between reads and writes, we consider that reads and write_fences issued by the same process can always be swapped, and the same holds concerning writes and read_fences. Then, we consider the following set of rewrite rules RWF defining the semantics of read/write fences: For any $i \in [n]$, $j \in [m]$, $d \in D$,

$$\begin{aligned} \mathsf{wfence}(i)\mathsf{r}(i,j,d) &\hookrightarrow \mathsf{r}(i,j,d)\mathsf{wfence}(i) \\ \mathsf{r}(i,j,d)\mathsf{wfence}(i) &\hookrightarrow \mathsf{wfence}(i)\mathsf{r}(i,j,d) \\ \mathsf{rfence}(i)\mathsf{w}(i,j,d) &\hookrightarrow \mathsf{w}(i,j,d)\mathsf{rfence}(i) \\ \mathsf{w}(i,j,d)\mathsf{rfence}(i) &\hookrightarrow \mathsf{rfence}(i)\mathsf{w}(i,j,d) \end{aligned} \tag{2}$$

We also consider the following set RLWE (Read Local Write Early) of rewrite rules:

$$\mathsf{w}(i,j,d)\mathsf{r}(i,j,d) \hookrightarrow \mathsf{w}(i,j,d) \tag{3}$$

for any $i \in [n]$, $j \in [m]$, $d \in D$. These rules say that a read that occurs after a write of the same value on the same variable by the same process can be validated immediately.

Then, we consider that a memory model M is defined by the choice of a set of rewrite rules defining the allowed relaxations of the program order. For instance, we give in the table on the right the definition in this framework of well known memory models.

Model	Rewrite Rules
SC	$\emptyset$
TSO	$\mathsf{RWF} \cup \mathsf{RLWE} \cup \{\mathsf{w} \to \mathsf{r}\}$
PSO	$\mathsf{RWF} \cup \mathsf{RLWE} \cup \{\mathsf{w} \to \mathsf{r}, \mathsf{w} \to \mathsf{w}\}$

Clearly, SC can be simulated under both TSO and PSO by inserting a fence after each operation. It is also possible to simulate TSO under PSO by inserting a wfence before each write operation. Notice that the use of read_fences in TSO and PSO is not relevant since reads cannot be swapped in these models. Similarly, the use of write_fences in TSO is not relevant. But the possibility of using write_fences in PSO is important. Without this operation, it is not possible to simulate TSO under PSO.

Given a process $\mathcal{P}_i$ of the system $\mathcal{N}$, and two control state $p, p' \in P_i$, a computation trace of $\mathcal{P}_i$ from p to p' is a finite sequence $\tau = \omega_0 \cdots \omega_{\ell-1} \in \Omega(\{i\}, X, D)^*$ such that there is a sequence of control states $p_0 \cdots p_\ell \in P_i^*$ such that $p = p_0$, $p' = p_\ell$, and for every $j \in \{0, \ldots, \ell-1\}$, $(p_j, \omega_i, p_{j+1}) \in \Delta_i$. The set of computations traces of $\mathcal{P}_i$ from p to p' is denoted by $\mathcal{T}(\mathcal{P}_i, p, p')$.

Let R be a set of rewrite rules over traces defining a memory model M. Given a rewrite rule $\rho = \alpha \hookrightarrow \beta$, where $\alpha, \beta \in \Omega(\mathcal{N})^*$, and a computation trace $\tau \in \Omega(\mathcal{N})^*$, we define a rewriting relation $\hookrightarrow_\rho$ between traces as follows: $\tau \hookrightarrow_\rho \tau'$ if $\tau = \tau_1 \alpha \tau_2$ and $\tau' = \tau_1 \beta \tau_2$ for some $\tau_1, \tau_2 \in \Omega(\mathcal{N})^*$. As usual, $\hookrightarrow_\rho^*$ denotes the reflexive-transitive closure of $\hookrightarrow_\rho$. These definitions are generalized in the obvious way to sets of rules and sets of computation traces. Given a set of rewrite rules R, the closure of a set of traces T, denoted by $[T]_R$, is the smallest set containing T and which is closed under the application of the rules in R, i.e., $[T]_R = \{\tau' \in \Omega(\mathcal{N})^* \,:\, \tau \in T \wedge \tau \hookrightarrow_R^* \tau'\}$.

Given two traces τ_1 and τ_2, the shuffle of the two traces is the set of traces obtained by interleaving the elements of τ_1 and τ_2 while preserving the original order between elements of each trace. Formally, the operator $||$ is defined inductively as follows: (1) $\varepsilon || \tau = \tau || \varepsilon = \tau$, and (2) $\omega_1 \tau_1 || \omega_2 \tau_2 = \omega_1(\tau_1 || \omega_2 \tau_2) \cup \omega_2(\omega_1 \tau_1 || \tau_2)$ for every

$\omega_1, \omega_2 \in \Omega(\mathcal{N})$, and for every $\tau, \tau_1, \tau_2 \in \Omega(\mathcal{N})^*$. The definition can be extended in a straightforward manner to a finite number of traces.

Given two vector of control states $\mathbf{p}, \mathbf{p}' \in \mathbf{P}$, the set of computation traces in $\mathcal{N}$ from $\mathbf{p}$ to $\mathbf{p}'$ in the memory model M (defined by R), denoted by $\mathcal{T}_{\mathsf{M}}(\mathcal{N}, \mathbf{p}, \mathbf{p}')$, is defined by

$$[\mathcal{T}(\mathcal{P}_1, \mathbf{p}[1], \mathbf{p}'[1])]_R \parallel \ldots \parallel [\mathcal{T}(\mathcal{P}_n, \mathbf{p}[n], \mathbf{p}'[n])]_R$$

We define a relation $[\,\rangle$ between memory states corresponding to the execution of operations in $\Omega(\mathcal{N})$. Given $\mathbf{d}, \mathbf{d}' \in M$, we have, for every $i \in [n]$ and for every $j \in [m]$:

- $\mathbf{d}[\mathsf{w}(i, j, d)\rangle \mathbf{d}'$ if $\mathbf{d}' = \mathbf{d}[j \leftarrow d]$,
- $\mathbf{d}[\mathsf{r}(i, j, d)\rangle \mathbf{d}'$ if $\mathbf{d}[j] = d$ and $\mathbf{d} = \mathbf{d}'$,
- $\mathbf{d}[\mathsf{arw}(i, j, d, d')\rangle \mathbf{d}'$ if $\mathbf{d}[j] = d$ and $\mathbf{d}' = \mathbf{d}[j \leftarrow d']$,
- $\mathbf{d}[op\rangle \mathbf{d}'$ with $op \in \{\mathsf{nop}, \mathsf{wfence}(i), \mathsf{rfence}(i)\}$, if $\mathbf{d} = \mathbf{d}'$.

We extend this definition to sequences of operations, and therefore to computation traces. A *state* of $\mathcal{N}$ is a pair $\langle \mathbf{p}, \mathbf{d} \rangle$ where $\mathbf{p} \in \mathbf{P}$ and $\mathbf{d} \in M$. For a given memory model M, we define a reachability relation $Reach^{\mathsf{M}}_{\mathcal{N}}$ between states of $\mathcal{N}$ as follows. Let $s = \langle \mathbf{p}, \mathbf{d} \rangle$ and $s' = \langle \mathbf{p}', \mathbf{d}' \rangle$ be two states of $\mathcal{N}$. We consider that $Reach^{\mathsf{M}}_{\mathcal{N}}(s, s')$ holds if there exists a trace $\tau \in \mathcal{T}_{\mathsf{M}}(\mathcal{N}, \mathbf{p}, \mathbf{p}')$ such that $\mathbf{d}[\tau\rangle \mathbf{d}'$.

3.4 The State Reachability Problem

The state reachability problem for a memory model M consists in, given a concurrent system $\mathcal{N}$ and two state s and s' of $\mathcal{N}$, checking whether $Reach^{\mathsf{M}}_{\mathcal{N}}(s, s')$ holds. We have:

Theorem 1 ([3]). *The state reachability problem for* TSO *is decidable.*

We have also proved in [3] the decidability of the state reachability problem for a model with both $\mathsf{w} \to \mathsf{w}$ and $\mathsf{w} \to \mathsf{r}$ relaxations, but without considering write_fences. Therefore, the so-called PSO model in [3] is incomparable with TSO, and is strictly less expressive than the PSO model as defined in this paper. We show also in [3] that the state reachability problem is undecidable for the model where all four read/write relaxations are considered. We prove here the following stronger result:

Theorem 2. *The state reachability problem for* $\mathsf{TSO} \cup \{\mathsf{r} \to \mathsf{w}\}$ *is undecidable.*

The proof is by a reduction of Post's Correspondence Problem to our problem. It follows a similar schema as the one we used in [3] for $\mathsf{TSO} \cup \{\mathsf{r} \to \mathsf{r/w}\}$, although the encoding is quite different. The proof can be found in Appendix A.

3.5 NSW: A Model with Non Speculative Writes

We have seen in Section 3.4 that including the $\mathsf{r} \to \mathsf{w}$ relaxation to TSO results in a memory model with an undecidable state reachability problem. Motivated by this, we introduce a memory model called NSW (for Non Speculative Writes) obtained by discarding this relaxation, i.e., by considering the following set of rules:

$$\mathsf{NSW} = \mathsf{RLWE} \cup \mathsf{RWF} \cup \{\mathsf{w} \to \mathsf{r},\ \mathsf{w} \to \mathsf{w},\ \mathsf{r} \to \mathsf{r}\}$$

Clearly, the NSW model subsumes TSO and PSO, and since it allows out-of-order reads, it is actually a strictly more relaxed model than PSO. Notice that PSO can be simulated under NSW by inserting a rfence after each read operation.

We show later that the state reachability problem problem for NSW is decidable. In the next section, we discuss another desirable property of the NSW memory model.

3.6 Expressive Power of NSW

Clearly, the NSW model subsumes TSO and PSO, and since it allows out-of-order reads, it is actually a strictly more relaxed model than PSO. We show hereafter examples of behaviors that are allowed under this model.

SB (Store Buffering): This example is an abstract version of Dekker's exclusion mutual protocol. Process $\mathcal{P}_1$ writes a 1 to x, and then checks that y is still equal 0 is order to proceed (to the critical section). Symmetrically, process $\mathcal{P}_2$ writes a 1 to y and then checks that x is 0. Under the SC model, it is not possible to execute the reads of both processes in a same execution.

$x = y = 0$	
$\mathcal{P}_1$	$\mathcal{P}_2$
(1) $\mathsf{w}(x,1)$ (2) $\mathsf{r}(y,0)$	(3) $\mathsf{w}(y,1)$ (4) $\mathsf{r}(x,0)$
$x = y = 1$	

In NSW this behavior is allowed since reads can overtake writes. So, it is possible to swap (1) and (2) for instance, and get the following computation: $\mathsf{r}(y,0)\mathsf{w}(y,1)\mathsf{r}(x,0)\mathsf{w}(x,1)$. Symmetrically, it is possible to swap (3) and (4) and obtain: $\mathsf{r}(x,0)\mathsf{w}(x,1)\mathsf{r}(y,0)\mathsf{w}(y,1)$. Notice that these computations are already allowed in TSO.

MP (Message Passing): Process $\mathcal{P}_1$ assigns the value 2 to x, and signals this fact to $\mathcal{P}_2$ by writing 1 on y. However, even if $\mathcal{P}_2$ reads this 1, this does not guarantee that it will read the value 2 on x.

$x = y = 0$	
$\mathcal{P}_1$	$\mathcal{P}_2$
(1) $\mathsf{w}(x,2)$ (2) $\mathsf{w}(y,1)$	(3) $\mathsf{r}(y,1)$ (4) $\mathsf{r}(x,0)$
$x = y = 1$	

First, this behavior is possible because the write operations (1) and (2) can be permuted, and this leads to the computation: $\mathsf{w}(y,1)\mathsf{r}(y,1)\mathsf{r}(x,0)\mathsf{wrel}(x,1)$. Notice that this behavior is already possible in PSO. To forbid this permutation, a write_fence must be inserted between (1) and (2). Even in this case, still the behavior is possible since two read operations (3) and (4) can be permuted, leading to the computation: $\mathsf{r}(x,0)\mathsf{wrel}(x,1)\mathsf{w}(y,1)\mathsf{r}(y,1)$. To forbid this latter computation a read_fence between (3) and (4) is needed.

WRC (Write to Read Causality): Process $\mathcal{P}_1$ writes 1 on x, and then $\mathcal{P}_2$ reads this value and writes 1 on y. Then, when $\mathcal{P}_3$ reads 1 on y it not guaranteed that it can read 1 on x.

$x = y = 0$		
$\mathcal{P}_1$	$\mathcal{P}_2$	$\mathcal{P}_3$
(1) $\mathsf{w}(x,1)$	(2) $\mathsf{r}(x,1)$ (3) $\mathsf{w}(y,1)$	(4) $\mathsf{r}(y,1)$ (5) $\mathsf{r}(x,0)$
$x = y = 1$		

Similarly to the previous case, this behavior is allowed in NSW by permuting the two reads (4) and (5), which leads to the following computation: $\mathsf{r}(x,0)\mathsf{w}(x,1)\mathsf{r}(x,1)\mathsf{w}(y,1)\mathsf{r}(y,1)$. Again, to forbid this computation, it is possible to insert a read_fence between (4) and (5).

IRIW (Independent Reads of Independent Writes): Processes $\mathcal{P}_3$ writes 1 to x and $\mathcal{P}_4$ writes 1 to y. In parallel, $\mathcal{P}_1$ observes that x has been modified before y, whereas $\mathcal{P}_2$ observes that y is modified before x.

$x = y = 0$			
$\mathcal{P}_1$	$\mathcal{P}_2$	$\mathcal{P}_3$	$\mathcal{P}_4$
(1) $\mathsf{r}(x,1)$ (2) $\mathsf{r}(y,0)$	(3) $\mathsf{r}(y,1)$ (4) $\mathsf{r}(x,0)$	(5) $\mathsf{w}(x,1)$	(6) $\mathsf{w}(y,1)$
$x = y = 1$			

This behavior is possible in NSW either by permuting (1) and (2), or by permuting (3) and (4). For instance, in the latter case this leads to the following execution: $\mathsf{r}(x,0)\mathsf{w}(x,1)\mathsf{r}(x,1)\mathsf{r}(y,0)\mathsf{w}(y,1)\mathsf{r}(y,1)$.

Forbidding this behavior is of course possible by inserting read_fences between (1) and (2) *and* between (3) and (4).

3.7 Absence of Causality Cycles in NSW

Let po denote the *program order* relation corresponding the order in which operations of each thread are issued by the program. Now, one can define a dependency relation between the operations of a same process that reflects the data and control dependencies. For instance, if a value read by a read operation is used to evaluate the branch condition, then all instructions subsequent to the branch are control dependent on the read operation. Similarly, if a value read in a read operation is used to compute the address of the memory location read in a subsequent read operation, or to compute the value written in a subsequent write, these two subsequent operations are data dependent on the first read operation.

While the exact definition of the dependency relation is based on the semantics of the programming language, we define a conservative definition by considering that all operations occurring after, in the program order, a read operation is considered dependent on this read. Formally, this corresponds to the following dependency relation.

$$\mathsf{dep} = \mathsf{po} \cap (\{\mathsf{r}\} \times \{\mathsf{r}, \mathsf{w}, \mathsf{arw}\}) \tag{4}$$

Second, we consider a *read-from* relation, denoted rf, that associates with each read event of the computation a write event such that $\mathsf{w}(i,k,d) \rightarrow_{\mathsf{rf}} \mathsf{r}(j,k,d)$ if the $\mathsf{r}(j,k,d)$ operation issued by process $\mathcal{P}_j$ takes the value d that has been written by the operation $\mathsf{w}(i,k,d)$ issued by process $\mathcal{P}_i$ on the variable x_k.

Then, the causality relation corresponding to the considered computation is defined by $\mathsf{c} = \mathsf{dep} \cup \mathsf{rf}$. It can be seen that under the model SC $\cup \{\mathsf{r} \rightarrow \mathsf{w}\}$, there are programs having computations with a cyclic causality relation. An example of such a program is given on the right.

$x = y = 0$	
$\mathcal{P}_1$	$\mathcal{P}_2$
(1) $\mathsf{r}(x,1)$	(3) $\mathsf{r}(y,1)$
(2) $\mathsf{w}(y,1)$	(4) $\mathsf{w}(x,1)$
$x = y = 1$	

It is clear that under the SC model, the four operations of this program cannot belong to a same computation leading to the configuration $x = y = 1$ starting from $x = y = 0$. However, when the $\mathsf{r} \rightarrow \mathsf{w}$ relaxation is admitted, it is possible for instance, by permuting (1) and (2), to execute the four operations in the following order $\mathsf{w}(y,1)\mathsf{w}(x,1)\mathsf{r}(x,1)\mathsf{r}(y,1)$. This computation contains the causality cycle: $\mathsf{r}(x,1) \rightarrow_{\mathsf{dep}} \mathsf{w}(y,1) \rightarrow_{\mathsf{rf}} \mathsf{r}(y,1) \rightarrow_{\mathsf{dep}} \mathsf{w}(x,1) \rightarrow_{\mathsf{rf}} \mathsf{r}(x,1)$. Intuitively, by executing the operation $\mathsf{w}(y,1)$ before $\mathsf{r}(x,1)$, process $\mathcal{P}_1$ speculates on the success of the read operation $\mathsf{r}(x,1)$ in the future. But process $\mathcal{P}_2$ can read the 1 on y that was speculatively written, and then write the value 1 to x, allowing this way the validation of the speculation of $\mathcal{P}_1$. We prove that by avoiding the $\mathsf{r} \rightarrow \mathsf{w}$ relaxation, NSW avoids causal cycles.

Theorem 3. *Every computation of any concurrent system under the* NSW *model has an acyclic causality relation.*

The proof of Theorem 3 can be found in Appendix B. Notice that since this theorem relies on the conservative definition of dependency relation defined above (4), it also holds for any refinement of the dependency relation.

4 An Operational Model for NSW

We provide an operational model for NSW where configurations are formed by a vector of control states, one per process, a memory state giving the valuation of the shared variables, and an *event structure* where pending operations, issued by the different processes but not yet executed, are stored. This event structure defines a partial order between these operations reflecting the constraints imposed by the memory model on the order of their execution.

We start by defining formally the notion of event structure. Then, we define a first operational model where the stored operations can be reads, writes, or write_fences. (Nop's, atomic read-writes, and read_fences do not need to be stored.)

4.1 Event structures

Let $\mathcal{E}$ be an enumerable set of of events. An *event structure* over an alphabet Σ is a tuple $\mathcal{S} = (E, \rightsquigarrow, \lambda)$ where E is a finite subset of $\mathcal{E}$, $\rightsquigarrow \subseteq E \times E$ is a partial order over E, and $\lambda : E \to \Sigma$ is a mapping associating with each event a symbol in Σ.

Given an event $e \in \mathcal{E} \setminus E$ and a symbol $a \in \Sigma$, we denote by $\mathcal{S} \lhd [e \leftarrow a]$ the structure $(E \cup \{e\}, \rightsquigarrow, \lambda')$ such that $\lambda'(e) = a$ and $\lambda'(e') = \lambda(e')$ for all $e' \in E$. Given an event $e \in E$, we denote by $\mathcal{S} \rhd e$ the structure $(E' = E \setminus \{e\}, \rightsquigarrow|_{E'}, \lambda|_{E'})$. Moreover, given $e, e' \in E$, we denote by $\mathcal{S} \oplus e \rightsquigarrow e'$ the event structure $(E, (\rightsquigarrow \cup \{(e, e')\})^*, \lambda)$. These notations can be generalized to sets (of events and transitions) in the obvious way.

Given a concurrent system $\mathcal{N} = (\mathcal{P}_1, \ldots, \mathcal{P}_n)$, an *event structure* $\mathcal{S}$ over $\mathcal{N}$ is an event structure over $\Omega(\mathcal{N})$. Given $i \in [n]$ and $j \in [m]$, let $E_{(i,j)} = \{e \in E \ : \ \exists d \in D.\ \exists op \in \{\mathsf{w}, \mathsf{r}\}.\ \lambda(e) = op(i, j, d)\}$. An event structure over $\Omega(N)$ is *well-formed* if, for every i and j, the relation $\rightsquigarrow|_{E_{(i,j)}}$ is a total order. We assume in the rest of the paper that all event structures over $\mathcal{N}$ are well-formed. This condition corresponds to the fact that read/write operations on the same variable should not be reordered.

Let $\widehat{E}_{(i,j)} = E_{(i,j)} \cup \{e \in E \ : \ \lambda(e) = \mathsf{wfence}(i)\}$. For every $i \in [n]$ and $j \in [m]$, let $RE(i, j) = \{e \in E \ : \ \exists d \in D.\ \lambda(e) = \mathsf{r}(i, j, d)\}$, and let $WE(i, j) = \{e \in E \ : \ \exists d \in D.\ \lambda(e) = \mathsf{w}(i, j, d)\}$. For every $e \in E$, we use $data(e)$ to denote $data(\lambda(e))$.

4.2 An Operational Model with Stored Reads

We associate with the concurrent system $\mathcal{N}$ a transition system $(Conf_{\mathcal{N}}, \Rightarrow_{\mathcal{N}})$ where $Conf_{\mathcal{N}}$ is a set of configurations, and $\Rightarrow_{\mathcal{N}} \subseteq Conf_{\mathcal{N}} \times Conf_{\mathcal{N}}$ is a transition relation between configurations.

A *configuration* of $\mathcal{N}$ (an element of $Conf_{\mathcal{N}}$) is any triple $(\mathbf{p}, \mathbf{d}, \mathcal{S})$ where $\mathbf{p} \in \mathbf{P}$, $\mathbf{d} \in M$, and $\mathcal{S}$ is an event structure over $\mathcal{N}$. The transition relation $\Rightarrow_{\mathcal{N}}$ is the smallest relation such that for every $\mathbf{p}, \mathbf{p}' \in \mathbf{P}$, for every $\mathbf{d}, \mathbf{d}' \in M$, and for every $\mathcal{S} = (E, \rightsquigarrow, \lambda)$, $\mathcal{S}' = (E', \rightsquigarrow', \lambda')$ two event structures over $\mathcal{N}$, we have $(\mathbf{p}, \mathbf{d}, \mathcal{S}) \Rightarrow_{\mathcal{N}} (\mathbf{p}', \mathbf{d}', \mathcal{S}')$ if there is an $i \in [n]$, and there are $p, p' \in P_i$, such that $\mathbf{p}[i] = p$, $\mathbf{p}' = \mathbf{p}[i \leftarrow p']$, and one of the following cases hold:

1. Nop: $p \xrightarrow{\mathsf{nop}}_i p'$, $\mathbf{d} = \mathbf{d}'$, and $\mathcal{S} = \mathcal{S}'$.
2. Write: $p \xrightarrow{\mathsf{w}(i,j,d)}_i p'$, $\mathbf{d} = \mathbf{d}'$, and $\exists e \in \mathcal{E} \setminus E$ such that $\mathcal{S}' = ((\mathcal{S} \lhd [e \leftarrow \mathsf{w}(i, j, d)]) \oplus \{e' \rightsquigarrow e \ : \ e' \in max(\widehat{E}_{(i,j)})\}$.
3. RLWE: $p \xrightarrow{\mathsf{r}(i,j,d)}_i p'$, $\mathbf{d} = \mathbf{d}'$, $\mathcal{S}' = \mathcal{S}$, $WE(i, j) \neq \emptyset$ with $e_m = max(WE(i, j))$, $\nexists e \in RE(i, j).\ e_m \rightsquigarrow e$, and $data(e_m) = d$.
4. Read: $p \xrightarrow{\mathsf{r}(i,j,d)}_i p'$, $\mathbf{d} = \mathbf{d}'$, either $WE(i, j) = \emptyset$ or $data(max(WE(i, j))) \neq d$, and $\exists e, f \in \mathcal{E} \setminus E$ such that $\mathcal{S}' = ((\mathcal{S} \lhd \{[e \leftarrow \mathsf{r}(i, j, d)], [f \leftarrow \mathsf{wfence}(i)]\}) \oplus (\{e' \rightsquigarrow e \ : \ e' \in max(E_{(i,j)})\} \cup \{e \rightsquigarrow f\}))$.
5. Atomic Read-write: $p \xrightarrow{\mathsf{arw}(i,j,d,d')}_i p'$, $\bigcup_{\ell=1}^{m} \widehat{E}_{(i,\ell)} = \emptyset$, $\mathbf{d}[j] = d$, $\mathbf{d}' = \mathbf{d}[j \leftarrow d']$, and $\mathcal{S} = \mathcal{S}'$.
6. Read fence: $p \xrightarrow{\mathsf{rfence}(i)}_i p'$, $\bigcup_{j=1}^{m} RE(i, j) = \emptyset$, $\mathbf{d} = \mathbf{d}'$, and $\mathcal{S} = \mathcal{S}'$.

7. Write fence: $p \xrightarrow{\mathsf{wfence}(i)}_i p'$, $\mathbf{d} = \mathbf{d}'$, and $\exists e \in \mathcal{E} \setminus E$ such that $\mathcal{S}' = ((\mathcal{S} \lhd [e \leftarrow \mathsf{wfence}(i)]) \oplus \{e' \rightsquigarrow e : \exists k.\ 1 \leq k \leq m \text{ and } e' \in max(\widehat{E}_{(i,k)})\})$.
8. Memory update: $\mathbf{p} = \mathbf{p}'$, and there is an event e such that e is a minimal of $\rightsquigarrow$, $\lambda(e) = \mathsf{w}(i, j, d)$ for some $d \in D$, $\mathbf{d}' = \mathbf{d}[j \leftarrow d]$, and $\mathcal{S}' = \mathcal{S} \rhd e$.
9. Read validation: $\mathbf{p} = \mathbf{p}'$, $\mathbf{d}' = \mathbf{d}$, and there is an event e such that e is a minimal of $\rightsquigarrow$, $\lambda(e) = \mathsf{r}(i, j, d)$, $\mathbf{d}[j] = d$, and $\mathcal{S}' = \mathcal{S} \rhd e$.
10. Write fence elimination: $\mathbf{p} = \mathbf{p}'$, $\mathbf{d}' = \mathbf{d}$, and there is an event e such that e is a minimal of $\rightsquigarrow$, $\lambda(e) = \mathsf{wfence}(i)$, and $\mathcal{S}' = \mathcal{S} \rhd e$.

Let us explain each case. A write operation $\mathsf{w}(i, j, d)$ is simply added to the structure by introducing a new event e labelled with this operation, which is inserted after all write_fences issued by $\mathcal{P}_i$ as well as all the write/read operations of $\mathcal{P}_i$ on x_j.

A read operation $\mathsf{r}(i, j, d)$ can be validated immediately (point 3) if $\mathcal{S}$ still contain a write of $\mathcal{P}_i$ on x_j (and there is no read of $\mathcal{P}_i$ on x_i after this write), and the last of such an operation writes precisely the value d on x_j. Otherwise, (in point 4) a read operation $\mathsf{r}(i, j, d)$ is simply added to the structure $\mathcal{S}$ after all reads/writes of $\mathcal{P}_i$ on x_j. Notice, that the event associated with this read operation is not ordered w.r.t. write_fences that are maximal in $\mathcal{S}$ (i.e., the read is allowed to overtake such write_fences). Moreover, a new write_fence is inserted after the read. This ensures that, as long as this read has not been validated, it cannot be overtaken by any write.

An atomic read-write operation, which acts as a fence on all operations of the process $\mathcal{P}_i$, can be executed only when all events before it have been executed. A read_fence issued by $\mathcal{P}_i$ is executed immediately (it is not stored in $\mathcal{S}$) if there is no reads in $\mathcal{S}$ issued by $\mathcal{P}_i$. A write_fence is inserted in $\mathcal{S}$ after all the events issued by $\mathcal{P}_i$.

Writes are removed from $\mathcal{S}$ and used to update the main memory when these operations correspond to minimal events of $\mathcal{S}$. Similarly, reads are validated w.r.t. the main memory and removed from $\mathcal{S}$ if they correspond to minimal events. Finally, a write_fence can simply be removed from $\mathcal{S}$ when it becomes minimal.

Let $\mathcal{S}_\emptyset$ denote the empty event structure. Then, we have:

Theorem 4. *For every states s and s', we have $Reach^{\mathsf{NSW}}_{\mathcal{N}}(s, s')$ iff $(s, \mathcal{S}_\emptyset) \Rightarrow^*_{\mathcal{N}} (s', \mathcal{S}_\emptyset)$.*

5 From Event Structures to FIFO Buffers

We provide in this section a model for NSW using FIFO buffers where reads and fences are never stored. We proceed in two steps. First, we show that it is possible to define an alternative operational model for NSW where reads can be immediately validated using informations about the sequence of states that the memory had in the past. The history of the memory states is stored in an additional FIFO buffer. Then, we show that it is also possible to get rid of write_fences by converting event structures into two-level structures of write buffers.

5.1 Eliminating reads from event structures

We present hereafter a new operational model where reads are validated using an additional buffer storing memory states, called *history buffer*. The idea is the following.

Consider a read operation $\mathsf{r}(i,j,d)$ issued by process $\mathcal{P}_i$ that can be validated during a computation by reading from a write operation $\mathsf{w}(k,j,d)$ issued by porcess $\mathcal{P}_k$. Then, if at the moment $\mathsf{r}(i,j,d)$ is issued $\mathsf{w}(k,j,d)$ has not yet been issued, it is actually possible for $\mathcal{P}_i$ to wait until $\mathcal{P}_k$ produces $\mathsf{w}(k,j,d)$. The reason is that issuing $\mathsf{w}(k,j,d)$ by $\mathcal{P}_k$ cannot depend from the actions of $\mathcal{P}_i$ after $\mathsf{r}(i,j,d)$, because otherwise, this would mean that there is a read by $\mathcal{P}_k$ before $\mathsf{w}(k,j,d)$ which needs (i.e., is causally dependent from) a write of $\mathcal{P}_i$ occurring after $\mathsf{r}(i,j,d)$. But this would imply the existence of a causality cycle, which contradicts the fact that such cycle do not exist in NSW computations due to the fact that writes cannot overtake reads (see Thm. 3).

Therefore, it is always possible to consider computations where reads are validated w.r.t. writes that have been issued in the past. However, since some actions must exit the event structure of the system configuration (due to fences), we need to maintain the history of all past memory states in a buffer.

Then, we use a buffer such that the last element represents actually the current state of the memory, and where the other elements represent the precedent states of the memory in the order they have been produced. Notice that a history buffer is never empty since it must contain at least one element representing the current state of the memory. For instance, consider the program on the right.

$x_1 = x_2 = 0$	
$\mathcal{P}_1$	$\mathcal{P}_2$
(1) $\mathsf{w}(x_1,1)$	(4) $\mathsf{r}(x_2,1)$
(2) wfence	(5) nop
(3) $\mathsf{w}(x_2,1)$	(6) $\mathsf{r}(x_1,0)$
$x_1 = x_2 = 1$	

Clearly the six operations of this program are executable under NSW because the operation $\mathsf{r}(x_1,0)$ can overtake the operation $\mathsf{r}(x_2,1)$. Now, let see how we can simulate this behavior without storing reads, starting from the memory state $(x_1 = 0, x_2 = 0)$. To validate the operation $\mathsf{r}(x_2,1)$, we need that $\mathcal{P}_1$ executes its third operation ($\mathsf{w}(x_2,1)$). Then, this process should perform its first write operation which is stored in the event structure, but the write_fence forces the execution of $\mathsf{w}(x_1,1)$, and the new memory state is $(x_1 = 1, x_2 = 0)$. After the execution of $\mathsf{w}(x_2,1)$ it is possible to validate $\mathsf{r}(x_2,1)$, but the validation of the operation $\mathsf{r}(x_1,0)$ of $\mathcal{P}_2$ needs the old memory state $(x_1 = 0, x_2 = 0)$. Putting this state in a history buffer allows to retrieve it in order to validate the last read and finish our simulation. Notice that in general the sequence of memory states that are needed to validate reads is not bounded. For instance, consider the case where $\mathsf{w}(x_1,1)$ of $\mathcal{P}_1$ is replaced by any longer sequence of different writes on x_1.

Now, since reads can be swapped, their validation can use writes that might be issued in a different order. However, reads by the same process on a same variable must be done in a coherent way, i.e., they should read from states occurring in the same order. To ensure that, we introduce pointers $\pi(i,j)$ on the history buffer defining for each process $\mathcal{P}_i$ and each variable x_j the oldest memory state that can be observed. Then, to validate a read on x_j by $\mathcal{P}_i$, we should find a memory state that occurs after $\pi(i,j)$ in the buffer where x_j has the right value. Actually, to simplify the construction, we allow that a pointer can move in a nondeterministic way toward the tail of the buffer (i.e., the most recent element). Then, to validate an operation $\mathsf{r}(i,j,d)$, we simply require that the value of x_j in the element pointed by $\pi(i,j)$ is precisely d. Also, when a write event $\mathsf{w}(i,j,d)$ exits the event structure and is used to update the memory, the pointer $\pi(i,j)$ is moved to the last element of the history buffer (i.e., the current state of the memory) since this is the only value of x_j that is visible to $\mathcal{P}_i$.

For instance, in the example above, the history buffer after the execution of $\mathsf{w}(x_2,1)$ is $(1,1)(1,0)(0,0)$, where the head (i.e., oldest element) is the right-most element, and the pointers are the following: $\pi(2,1)=(0,0)$ and $\pi(2,2)=(1,1)$. Then, it is possible to validate $\mathsf{r}(x_2,1)$ since the element pointed by $\pi(2,2)$ gives the right value, and similarly, pointer $\pi(2,1)$ allows to validate $\mathsf{r}(x_1,0)$. Actually, it is clear that at any moment, the relevant part of the history buffer is formed by the elements between the last element and the oldest element that is pointed by π. Beyond the first pointer, the elements can be considered as garbage and can be eliminated (but we do not need to). When the history buffer is reduced to one element (i.e., all the pointers point to the last element), this means that the buffer contains only the current state of the memory.

To give the formal description of our model, we need to introduce some definitions concerning buffers and their manipulation. An event structure $(E,\rightsquigarrow,\lambda)$ is *totally ordered* when $\rightsquigarrow$ is a total order. We use such structures to encode FIFO buffers. Given a buffer $\mathcal{B}=(E,\rightsquigarrow,\lambda)$ over an alphabet Σ, and a symbol $a\in\Sigma$, let $\mathit{add}(\mathcal{B},a)$ be the buffer $(E',\rightsquigarrow',\lambda')$ such that (1) $E'=E\cup\{e\}$ for some $e\in\mathcal{E}\setminus E$, (2) if $E=\emptyset$ then $\rightsquigarrow'=\{(e,e)\}$, otherwise $\rightsquigarrow'=(\rightsquigarrow\cup\{(\mathit{max}(E),e)\})^*$, and (3) $\lambda'=\lambda\cup[e\mapsto a]$. Then, if $\lambda(\mathit{min}(B))=a$, let $\mathit{remove}(\mathcal{B},a)$ be the buffer $(E',\rightsquigarrow',\lambda')$ such that (1) $E'=E\setminus\{\mathit{min}(E)\}$, (2) $\rightsquigarrow'=\rightsquigarrow|_{E'}$, and (3) $\lambda'=\lambda|_{E'}$. We also define the predicate *Empty* which is true when the buffer has an empty set of events. When the buffer $\mathcal{B}$ is not empty, we denote by $\mathit{tail}(\mathcal{B})$ (resp. $\mathit{head}(\mathcal{B})$) the element $\lambda(\mathit{max}(E))$ (resp. $\lambda(\mathit{min}(E))$).

Given a concurrent system $\mathcal{N}$, a *history buffer* of memory states is a tuple $\mathcal{H}=(E,\rightsquigarrow,\lambda,\pi)$ where $(E,\rightsquigarrow,\lambda)$ is a buffer over M (the set of all memory states) such that $E\neq\emptyset$, and $\pi:[n]\times[m]\to E$ is a mapping associating with each process and each variable an event in E. We say that a history buffer is *unitary* if $\mathcal{H}$ is reduced to a singleton (i.e., $\pi(i,j)=\mathit{max}(E)$ for all $i\in[n]$ and $j\in[m]$).

Then, we are ready to define the transition system of the new model. A configuration is a tuple $\langle\mathbf{p},\mathcal{S},\mathcal{H}\rangle$ where, as in the previous model $\mathbf{p}\in\mathbf{P}$ is a vector of control states of each of the processes and $\mathcal{S}$ is an event structure, and where $\mathcal{H}$ is a history buffer over M. The new transition relation $\Rrightarrow_{\mathcal{N}}$ is the smallest relation s.t. for every $\mathbf{p},\mathbf{p}'\in\mathbf{P}$, $\mathcal{S}=(E,\rightsquigarrow,\lambda),\mathcal{S}'=(E',\rightsquigarrow',\lambda')$ two event structures over $\mathcal{N}$, and $\mathcal{H}=(\mathcal{B},\pi)$ and $\mathcal{H}'=(\mathcal{B}',\pi')$ two history buffers over M, where $\mathcal{B}=(H,\rightsquigarrow_H,\lambda_H)$ and $\mathcal{B}'=(H',\rightsquigarrow_{H'},\lambda_{H'})$ are two buffers over M, we have $\langle\mathbf{p},\mathcal{S},\mathcal{H}\rangle\Rrightarrow_{\mathcal{N}}\langle\mathbf{p}',\mathcal{S}',\mathcal{H}'\rangle$ if there is an $i\in[n]$, and there are $p,p'\in P_i$, such that $\mathbf{p}[i]=p$, $\mathbf{p}'=\mathbf{p}[i\leftarrow p']$, and one of the following cases holds:

1. Nop: $p\xrightarrow{\mathsf{nop}}_i p'$, $\mathcal{S}=\mathcal{S}'$, and $\mathcal{H}=\mathcal{H}'$.
2. Write: $p\xrightarrow{\mathsf{w}(i,j,d)}_i p'$, $\mathcal{H}=\mathcal{H}'$, and $\exists e\in\mathcal{E}\setminus E$ such that $\mathcal{S}'=((\mathcal{S}\lhd[e\leftarrow\mathsf{w}(i,j,d)])\oplus\{e'\rightsquigarrow e\,:\,e'\in\mathit{max}(\widehat{E}_{(i,j)})\}$.
3. Write fence: $p\xrightarrow{\mathsf{wfence}(i)}_i p'$, $\mathcal{H}=\mathcal{H}'$, and $\exists e\in\mathcal{E}\setminus E$ such that $\mathcal{S}'=((\mathcal{S}\lhd[e\leftarrow\mathsf{wfence}(i)])\oplus\{e'\rightsquigarrow e\,:\,\exists k.\ 1\leq k\leq m\text{ and }e'\in\mathit{max}(\widehat{E}_{(i,k)})\})$.
4. RLWE: $p\xrightarrow{\mathsf{r}(i,j,d)}_i p'$, $\mathcal{S}=\mathcal{S}'$, $\mathcal{H}=\mathcal{H}'$, $\mathit{WE}(i,j)\neq\emptyset$, and $\mathit{data}(\mathit{max}(\mathit{WE}(i,j)))=d$.
5. Move pointer: $\mathbf{p}=\mathbf{p}'$, $\mathcal{S}=\mathcal{S}'$, $\mathcal{B}=\mathcal{B}'$, and $\exists j\in[m].\ \exists e\in H.\ \pi(i,j)\rightsquigarrow_H e$ and $\pi'=\pi[(i,j)\leftarrow e]$.
6. Read: $p\xrightarrow{\mathsf{r}(i,j,d)}_i p'$, $\mathcal{S}=\mathcal{S}'$, $\mathcal{H}=\mathcal{H}'$, $\mathit{WE}(i,j)=\emptyset$, and $\exists\mathbf{d}\in M$ such that $\lambda_H(\pi(i,j))=\mathbf{d}$ and $\mathbf{d}[j]=d$.

7. Read fence: $p \xrightarrow{\mathsf{rfence}(i)}_i p'$, $\mathcal{S}=\mathcal{S}'$, $\mathcal{H}=\mathcal{H}'$, and $\pi(i,j)=max(H)$ for every $j\in[m]$.
8. Atomic Read-write: $p \xrightarrow{\mathsf{arw}(i,j,d,d')}_i p'$, $\mathcal{S}=\mathcal{S}'$, $\bigcup_{\ell=1}^{m}\widehat{E}_{(i,\ell)}=\emptyset$, $\pi(i,\ell)=max(H)$ for every $\ell\in[m]$, there is a $\mathbf{d}=tail(\mathcal{B})$ such that $\mathbf{d}[j]=d$ and $\mathcal{B}'=add(\mathcal{B},\mathbf{d}[j\leftarrow d'])$, and $\pi'=\pi[(i,\ell)\leftarrow max(H')]_{\ell\in[m]}$.
9. Memory update: $\mathbf{p}=\mathbf{p}'$, $\exists e\in min(E)$ such that $\lambda(e)=\mathsf{w}(i,j,d)$ for some $j\in[m]$ and $d\in D$, $\mathcal{S}'=\mathcal{S}\rhd e$, $\mathcal{B}'=add(\mathcal{B},\mathbf{d})$ where $\mathbf{d}=tail(H)[j\leftarrow d]$, and $\pi'=\pi[(i,j)\leftarrow max(H')]$.
10. Write fence elimination: $\mathbf{p}=\mathbf{p}'$, $\mathcal{H}=\mathcal{H}'$, $\mathbf{d}'=\mathbf{d}$, and $\exists e\in min(E)$ such that $\lambda(e)=\mathsf{wfence}(i)$, and $\mathcal{S}'=\mathcal{S}\rhd e$.

Theorem 5. *Let $s=(\mathbf{p},\mathbf{d})$ and $s'=(\mathbf{p}',\mathbf{d}')$ be two states of $\mathcal{N}$, and let $\mathcal{H}$ and $\mathcal{H}'$ be two unitary history buffers over M such that $tail(\mathcal{H})=\mathbf{d}$ and $tail(\mathcal{H}')=\mathbf{d}'$. Then, $(s,\mathcal{S}_\emptyset)\Rightarrow^*_{\mathcal{N}}(s',\mathcal{S}_\emptyset)$ if and only if $\langle\mathbf{p},\mathcal{S}_\emptyset,\mathcal{H}\rangle\Rrightarrow^*_{\mathcal{N}}\langle\mathbf{p}',\mathcal{S}_\emptyset,\mathcal{H}'\rangle$.*

5.2 Eliminating write fences from event structures

$x=y=0$	
$\mathcal{P}_1$	$\mathcal{P}_2$
(1) $\mathsf{w}(x,1)$	(5) $\mathsf{w}(y,1)$
(2) wfence	(6) wfence
(3) rfence	(7) rfence
(4) $\mathsf{r}(y,0)$	(8) $\mathsf{r}(x,0)$
$x=y=1$	

We show in this section that it is possible to avoid storing write_fences and to convert event structures into write buffers. The idea is the following. We observe that the projection of the event structure on the events of a same process is, roughly speaking, a sequence of partial orders, each of these partial orders corresponding to the set of write events occurring between two successive write_fences. These partial order have also the property that they are unions of m total orders, each of them corresponding to the set of writes to a same variable. These total orders can naturally be manipulated using m FIFO buffers $WB_{(i,1)},\ldots,WB_{(i,m)}$. Then, to simulate the whole sequence of partial orders corresponding the the events of a process, we need to reuse the same buffers after each write_fence, while ensuring that all writes occurring before the write_fence are executed before all those occurring after it. The solution for that is to introduce for each process $\mathcal{P}_i$ an additional buffer $WB_{(i,m+1)}$ used to flush the buffers $WB_{(i,1)},\ldots,WB_{(i,m)}$ after each write_fence without imposing that their content is directly written in the main memory. To see the necessity of this, consider the example on the right which corresponds to the SB behavior that is also possible in TSO: The actions of this program are executable under NSW since reads and read_fences can overtake writes and write_fences. Then, if the execution of write_fences forces the commitment of the writes to the main memory, and since read_fences require that the next reads can only see the current memory state, then reads (4) and (8) in the program above cannot not be validated. However, if the writes (1) and (5) are flushed to intermediary buffers instead of being committed to the main memory, it is possible to validate the reads (4) and (8) since the main memory will remain unchanged.

To summarize, the architecture of our model is as follows. Each process $\mathcal{P}_i$ has two levels of buffers, a first level with m write buffers storing the writes for each variable, and a second level with one buffer used to serialize the writes before committing them to the main memory. Then, we have the history buffer, the last element of which represents the current state of the memory, and the rest of its elements represent the history of all past memory states. Pointers on this buffer allows to each process to know what is the oldest value it can read on each variable.

We give hereafter the formal definition of our model. A configuration in this model is a tuple of the form $\langle \mathbf{p}, (WB_{(i,j)})_{i\in[n]}^{j\in[m+1]}, \mathcal{H}\rangle$ where $\mathbf{p} \in \mathbf{P}$, for every $i \in [n]$ and every $j \in [m+1]$, $WB_{(i,j)}$ is a write buffer, and $\mathcal{H}$ is a history buffer over M. Then, we define the transition relation $\rightarrow_{\mathcal{N}}$ between configurations as the smallest relation such that for every $\mathbf{p},\mathbf{p}' \in \mathbf{P}$, for every two vectors of store buffers $(WB_{(i,j)})_{i\in[n]}^{j\in[m+1]}$ and $(WB'_{(i,j)})_{i\in[n]}^{j\in[m+1]}$, where $WB_{(i,j)} = (B_{(i,j)}, \leadsto_{(i,j)}, \lambda_{(i,j)})$ and $WB'_{(i,j)} = (B'_{(i,j)}, \leadsto'_{(i,j)}, \lambda'_{(i,j)})$ for all i and j, and for every two history buffers $\mathcal{H} = (\mathcal{B},\pi)$ and $\mathcal{H}' = (\mathcal{B}',\pi')$, where $\mathcal{B} = (H, \leadsto_H, \lambda_H)$ and $\mathcal{B}' = (H', \leadsto_{H'}, \lambda_{H'})$ are two buffers over M, we have $\langle \mathbf{p}, (WB_{(i,j)})_{i\in[n]}^{j\in[m+1]}, \mathcal{H}\rangle \rightarrow_{\mathcal{N}} \langle \mathbf{p}', (WB'_{(i,j)})_{i\in[n]}^{j\in[m+1]}, \mathcal{H}'\rangle$ if there are $i \in [n]$, and $p, p' \in P_i$, such that $\mathbf{p}[i] = p$, $\mathbf{p}' = \mathbf{p}[i \leftarrow p']$, $WB_{(k,j)} = WB_{(k,j)}$ for every $k \in [n]\setminus\{i\}$ and every $j \in [m+1]$, and one of the following cases holds:

1. Nop: $p \xrightarrow{\mathsf{nop}}_i p'$, $WB_{(i,j)} = WB'_{(i,j)}$ for every $j \in [m+1]$, and $\mathcal{H} = \mathcal{H}'$.
2. Write: $p \xrightarrow{\mathsf{w}(i,j,d)}_i p'$, $\mathcal{H} = \mathcal{H}'$, $WB_{(i,k)} = WB'_{(i,k)}$ for every $k \in ([m+1]\setminus\{j\}$, and $WB'_{(i,j)} = add(WB_{(i,j)}, \mathsf{w}(i,j,d))$.
3. Write fence: $p \xrightarrow{\mathsf{wfence}(i)}_i p'$, $Empty(WB_{(i,j)})$ for all $j \in [m]$, $WB_{(i,s)} = WB'_{(i,s)}$ for all $s \in [m+1]$, and $\mathcal{H} = \mathcal{H}'$.
4. Transfer write: $p = p'$, $\mathcal{H} = \mathcal{H}'$, $\exists j \in [m].\ WB_{(i,k)} = WB'_{(i,k)}$ for every $k \in ([m]\setminus\{j\})$, and $\exists\omega = head(WB_{(i,j)}).\ WB'_{(i,j)} = remove(WB_{(i,j)},\omega)$ and $WB'_{(i,m+1)} = add(WB_{(i,m+1)},\omega)$.
5. RLWE from $WB_{(i,j)}$, $j \in [m]$: $p \xrightarrow{\mathsf{r}(i,j,d)}_i p'$, $\mathcal{H} = \mathcal{H}'$, $WB_{(i,k)} = WB'_{(i,k)}$ for every $k \in [m+1]$, and $data(tail(WB_{(i,j)})) = d$.
6. RLWE from $WB_{(i,m+1)}$: $p \xrightarrow{\mathsf{r}(i,j,d)}_i p'$, $\mathcal{H} = \mathcal{H}'$, $WB_{(i,k)} = WB'_{(i,k)}$ for every $k \in [m+1]$, $Empty(WB_{(i,j)})$, the set $W_{(i,m+1)} = \{e \in B_{(i,m+1)} : \exists d' \in D.\ \lambda_{(i,m+1)}(e) = \mathsf{w}(i,j,d')\}$ is not empty, and $data(max(W_{(i,m+1)})) = d$.
7. Read: $p \xrightarrow{\mathsf{r}(i,j,d)}_i p'$, $\mathcal{H} = \mathcal{H}'$, $WB_{(i,k)} = WB'_{(i,k)}$ for every $k \in [m+1]$, $Empty(WB_{(i,j)})$, the set $W_{(i,m+1)}$ defined above is empty, and $\exists\mathbf{d} \in M$ such that $\lambda_H(\pi(i,j)) = \mathbf{d}$ and $\mathbf{d}[j] = d$.
8. Move pointer: $\mathbf{p} = \mathbf{p}'$, $\mathcal{B} = \mathcal{B}'$, $WB_{(i,k)} = WB'_{(i,k)}$ for every $k \in [m+1]$, and $\exists j \in [m].\ \exists e \in H.\ \pi(i,j) \leadsto_H e$ and $\pi' = \pi[(i,j) \leftarrow e]$.
9. Atomic Read-write: $p \xrightarrow{\mathsf{arw}(i,j,d,d')}_i p'$, $Empty(WB_{(i,j)})$ and $Empty(WB'_{(i,j)})$ for every $j \in [m+1]$, $\pi(i,\ell) = max(H)$ for every $\ell \in [m]$, there is a $\mathbf{d} = tail(\mathcal{B})$ such that $\mathbf{d}[j] = d$ and $\mathcal{B}' = add(\mathcal{B}, \mathbf{d}[j \leftarrow d'])$, and $\pi' = \pi[(i,\ell) \leftarrow max(H')]_{\ell\in[m]}$.
10. Read fence: $p \xrightarrow{\mathsf{rfence}(i)}_i p'$, $WB_{(i,k)} = WB'_{(i,k)}$ for every $k \in [m+1]$, $\mathcal{H} = \mathcal{H}'$, and $\pi(i,\ell) = max(H)$ for every $\ell \in [m]$.

11. Memory update: $\mathbf{p} = \mathbf{p}'$, $WB_{(i,k)} = WB'_{(i,k)}$ for every $k \in [m]$, $head(WB_{(i,m+1)}) = \mathsf{w}(i,j,d)$ for some $j \in [m]$ and $d \in D$, $WB'_{(i,m+1)} = remove(WB_{(i,m+1)}, \mathsf{w}(i,j,d))$, $\mathcal{B}' = add(\mathcal{B}, \mathbf{d})$ where $\mathbf{d} = tail(H)[j \leftarrow d]$, and $\pi' = \pi[(i,j) \leftarrow max(H')]$.

Theorem 6. *Let $s = (\mathbf{p}, \mathbf{d})$ and $s' = (\mathbf{p}', \mathbf{d}')$ be two states of $\mathcal{N}$, and let $\mathcal{H}$ and $\mathcal{H}'$ be two unitary history buffers over M such that $tail(\mathcal{H}) = \mathbf{d}$ and $tail(\mathcal{H}') = \mathbf{d}'$. Then, $(s, \mathcal{S}_\emptyset) \Rightarrow^*_{\mathcal{N}} (s', \mathcal{S}_\emptyset)$ if and only if $\langle \mathbf{p}, \overline{\mathcal{S}_\emptyset}, \mathcal{H} \rangle \rightarrow^*_{\mathcal{N}} \langle \mathbf{p}', \overline{\mathcal{S}_\emptyset}, \mathcal{H}' \rangle$, where $\overline{\mathcal{S}_\emptyset}$ denotes an $[n] \times [m+1]$-dim vector of empty write buffers.*

It is worth noting that for PSO, i.e., when read_fences are systematically inserted after reads, the operational model we define has always a history buffer of size 1 (i.e., reduced to the memory state). Notice that still we need two levels of write buffers for PSO due to the use of write_fences. For TSO, write buffers for each variable ($WB_{(i,j)}$ for $j \in [m]$) are not needed since writes are immediately followed by write_fences. This coincides with the operational model defined, e.g., in [3].

6 The state reachability problem of NSW

We show hereafter that the state reachability problem of NSW is decidable. For that, we use the framework defined in [1] which allows to establish that state reachability can be solved using backward reachability analysis in the following case: Given a well quasi-ordering (WQO) $\preceq$ on configurations[4], if the system is monotonic w.r.t. $\preceq$, i.e., larger configurations w.r.t. $\preceq$ can always simulate smaller configurations, then backward reachability in this system is guaranteed to terminate if it starts from $\preceq$-upward closed sets, i.e., sets that whenever they contain a configuration c, they also contain all $\preceq$-larger configurations than c.

To define such ordering, we observe that a value in the memory written by some process might be overwritten by other write operations by the same process before any other process has had time to read it. Therefore, the effect of a write operation sent by a process to its store buffer may never be used, and this would suggest that we should define $\preceq$ to reflect the subword relation between the buffer contents. However, this intuition cannot be exploited directly. As we will see below, NSW's are not monotonic in general w.r.t. such as subword-based relation. To circumvent this problem, we introduce another model called NSW$^+$ obtained from the NSW, where, roughly, serialization buffers $W_{(i,m+1)}$ contain memory states (corresponding to cumulated effects of write operations) instead of write operations and we associate one history buffer per process, and we show that (1) the state reachability problem in a given NSW is reducible to the one in its corresponding NSW$^+$, and (2) every NSW$^+$ is monotonic w.r.t. a subword-based relation on buffers. Notice that the translation from NSW to NSW$^+$ preserves reachability but the resulting model from this translation is not bisimilar to the original one (and therefore monotonicity can not be transferred).

[4] Recall that a well quasi-ordering $\preceq$ over a set E is an ordering such that for every infinite sequence $e_1, e_2, \ldots$ of elements of E, there exist two integers $i < j$ such that $e_i \preceq e_j$.

Informal introduction to NSW^+: We explain hereafter how a NSW^+ model is defined starting from a given NSW. Let us first see why NSW's are not monotonic w.r.t. the subword relation, i.e., considering that the buffers in NSW are *lossy* is not sound. More precisely, while it can be shown that it is possible to consider safely that the write buffers $WB_{(i,j)}$ for all $i \in [n]$ and $j \in [m]$ as well as the history buffer are lossy, the serialization buffers $WB_{(i,m+1)}$ for $i \in [n]$ cannot be simply turned to lossy buffers. Consider first a sequence of write operations $\mathsf{w}(i,j,d')\mathsf{w}(i,j,d)$ in the write buffer $WB_{(i,j)}$, for some $j \in [m]$, where $\mathsf{w}(i,j,d)$ is the oldest operation. Since both operations are on the same variable x_j, loosing the operation $\mathsf{w}(i,j,d)$, i.e., replacing this sequence by just $\mathsf{w}(i,j,d')$, yields a valid computation corresponding to compaction of the two operations. Indeed, it is possible to overwrite the value d by d' before that any process is able to read d. Therefore, it is possible to loose any operation in a write buffer corresponding to a variable, except the last operation. This is especially important for the read-local-write-early operation. Then, by considering the last symbol in each write buffer $WB_{(i,j)}$ as a strong symbol (can not be lost), and turning $WB_{(i,j)}$ to a lossy channel does not introduce computations that are not possible in the original program. Observe that the number of possible such strong symbols is finite (one per write buffer $WB_{(i,j)}$).

Consider now a sequence of memory states $\mathbf{d} \cdot \mathbf{d}'$ in the history buffer $\mathcal{H}$, where $\mathbf{d}'$ is the oldest state. Then, loosing the memory state $\mathbf{d}'$ in $\mathcal{M}_i$ is similar to considering that this state has not been observed by $\mathcal{P}_i$. This is perfectly valid since processes observe the states of the memory in an asynchronous way, and therefore they may miss some states. However, memory states in $\mathcal{H}$ that are pointed by some pointer $\pi(i,j)$ should not be lost, and they must be considered as strong symbol. Indeed, without these pointed states, reads cannot be validated. In addition, we also should not loose the tail of $\mathcal{H}$ (which corresponds to the current memory state) since it is used to compute the next memory state. Then, pointed elements as well as the last element of the history buffer must be considered as strong symbols (again the number of such symbols is finite).

It remains to consider the case of the serialization write buffer $WB_{(i,m+1)}$. Consider a sequence of operations $\mathsf{w}(i,j,d')\mathsf{w}(i,k,d)$ in $WB_{(i,m+1)}$. Since these two operations are on different variables, loosing $\mathsf{w}(i,k,d)$ does not correspond to the compaction of the two operations. To encode the compaction (or the summary) of such a sequence of operations, we need to use a vector of values defining the last written value to each variable by the operations in the sequence. Then, an idea is to replace the content of $WB_{(i,m+1)} = \omega_\ell \cdots \omega_1$ by the sequence of summaries $\sigma_\ell \cdots \sigma_1$ where σ_i is the summary of the sequence $\omega_i \cdots \omega_1$. For instance, in our example, the sequence of summaries is $(x_j = d', x_k = d)(x_k = d)$. Then, loosing $(x_k = d)$ does not correspond to loosing the effect of the operation $\mathsf{w}(i,k,d)$ since this effect is still visible in $(x_j = d', x_k = d)$. Assume now that $(x_k = d)$ has not been lost and has been updated to the main memory. This value of x_k in the main memory can be over-written by a write operation $(x_k = d'')$ $(d'' \neq d)$ of a different process from $\mathcal{P}_i$. Then, when the system decides to update $(x_j = d', x_k = d)$ to the main memory, we should not reset the value of x_k to d (since the write operation $(x_k = d)$ has already taken effect). This shows that $WB_{(i,m+1)}$ (under NSW^+) must contain a *valid* sequence of memory states (that will be used to update the memory in the future). Then, we can formulate a similar argument as in the case of the history buffer to allow some of the memory states in $WB_{(i,m+1)}$ to be lost.

However, in order to have a valid sequence of memory states, the serialization buffer $WB_{(i,m+1)}$ under NSW^+ should simulate the contributions of the other processes. Therefore, it has to insert in $WB_{(i,m+1)}$ the memory states resulting from writes performed by other processes. This implies that the system should guess in advance in which order the write operations will be updated to the main memory. This is performed under NSW^+ as follows: (1) a write is removed from some write buffer $WB_{(k,j)}$ (chosen nondeterministically), (2) a new memory state is then computed from the last state added to $WB_{(k,m+1)}$, and (3) this new state is added to *all* the serialization buffers. Observe that a memory state in $WB_{(i,m+1)}$ resulting from a write operation of a process $\mathcal{P}_k$ (with $k \neq j$) should not be detected by $\mathcal{P}_i$ (since it has not been yet committed to the main memory).

Observe that the execution of each thread is totally determined by the sequence of memory states and its local configuration (i.e., its control state, its store buffer contents, and its serialization buffer content). Therefore, under NSW^+, each process $\mathcal{P}_i$ has its own private copy of the history buffer $\mathcal{H}_i$ (without any need of synchronization with the other threads) since it has already the sequence of memory states in its serialization buffer. Now, if a memory state is at the head of the serialization buffer $WB_{(i,m+1)}$ of the process $\mathcal{P}_i$, then this state will be removed from all this buffer and one copy is transferred to its history buffer $\mathcal{H}_i$.

Formal definition of NSW^+: A configuration of NSW^+ is a tuple of the form $\langle \mathbf{p}, (WB_{(i,j)})_{i\in[n]}^{j\in[m+1]}, (\mathcal{H}_i)_{i\in[n]}\rangle$ where $\mathbf{p}$ and $(WB_{(i,j)})_{i\in[n]}^{j\in[m]}$ are defined as in the previous section, $(WB_{(i,m+1)})_{i\in[n]}$ are write buffers over $F = \{\mathsf{w}(i,j,\mathbf{d}) : j \in [m] \wedge \mathbf{d} \in M\}$, and $\mathcal{H}_i$ are history buffers over M. Then, we define the transition relation $\mapsto_{\mathcal{N}}$ as the smallest relation such that for every $\mathbf{p}, \mathbf{p}' \in \mathbf{P}$, for every two vectors of buffers $(WB_{(i,j)})_{i\in[n]}^{j\in[m+1]}$ and $(WB'_{(i,j)})_{i\in[n]}^{j\in[m+1]}$, where $WB_{(i,j)} = (B_{(i,j)}, \rightsquigarrow_{(i,j)}, \lambda_{(i,j)})$ and $WB'_{(i,j)} = (B'_{(i,j)}, \rightsquigarrow'_{(i,j)}, \lambda'_{(i,j)})$ for all $i \in [n]$ and $j \in [m+1]$, and for every two vectors of history buffers $\big(\mathcal{H}_i = (\mathcal{B}_i, \pi_i)\big)_{i\in[n]}$ and $\big(\mathcal{H}'_i = (\mathcal{B}'_i, \pi'_i)\big)_{i\in[n]}$, where $\mathcal{B}_i = (H_i, \rightsquigarrow_{H_i}, \lambda_{H_i})$ and $\mathcal{B}'_i = (H'_i, \rightsquigarrow_{H'_i}, \lambda_{H'_i})$ are two buffers over M for all $i \in [n]$, we have $\langle \mathbf{p}, (WB_{(i,j)})_{i\in[n]}^{j\in[m+1]}, (\mathcal{H}_i)_{i\in[n]}\rangle \rightarrow_{\mathcal{N}} \langle \mathbf{p}', (WB'_{(i,j)})_{i\in[n]}^{j\in[m+1]}, (\mathcal{H}'_i)_{i\in[n]}\rangle$ if there are $i \in [n]$, and $p, p' \in P_i$, such that $\mathbf{p}[i] = p$, $\mathbf{p}' = \mathbf{p}[i \leftarrow p']$, $\mathcal{H}_k = \mathcal{H}_k$ for all $k \in [n] \setminus \{i\}$, and one of the following cases holds:

1. Nop: $p \xrightarrow{\mathsf{nop}}_i p'$, $WB_{(k,j)} = WB'_{(k,j)}$ for all $k \in [n]$ and $j \in [m+1]$, and $\mathcal{H}_i = \mathcal{H}'_i$.
2. Write: $p \xrightarrow{\mathsf{w}(i,j,d)}_i p'$, $\mathcal{H}_i = \mathcal{H}'_i$, $WB_{(k,\ell)} = WB'_{(k,\ell)}$ for every $(k,\ell) \in ([n] \times [m+1]) \setminus \{(i,j)\}$, and $WB'_{(i,j)} = \mathit{add}(WB_{(i,j)}, \mathsf{w}(i,j,d))$.
3. Write fence: $p \xrightarrow{\mathsf{wfence}(i)}_i p'$, $\mathit{Empty}(WB_{(i,j)})$ for all $j \in [m]$, $WB_{(k,\ell)} = WB'_{(k,\ell)}$ for all $k \in [n]$ and $\ell \in [m+1]$, and $\mathcal{H}_i = \mathcal{H}'_i$.
4. Transfer write: $p = p'$, $\mathcal{H}_i = \mathcal{H}'_i$, $\exists j \in [m].\ WB_{(k,\ell)} = WB'_{(k,\ell)}$ for all $(k,\ell) \in ([n] \times [m] \setminus \{(i,j)\})$, and $\exists \omega = \mathit{head}(WB_{(i,j)})$. $WB'_{(i,j)} = \mathit{remove}(WB_{(i,j)}, \omega)$ and for every $k \in [n]$, $WB'_{(k,m+1)} = \mathit{add}(WB_{(k,m+1)}, \mathsf{w}(i,j,\mathbf{d}'))$ where $\mathbf{d}[\omega\rangle\mathbf{d}'$ and if

$Empty(WB_{(i,m+1)})$ then $\mathbf{d} = tail(\mathcal{B}_i)$ else $\mathsf{w}(t,\ell,\mathbf{d}) = tail(WB_{(i,m+1)})$ with $t \in [n]$ and $\ell \in [m]$.

5. RLWE from $WB_{(i,j)}$, $j \in [m]$: $p \xrightarrow{\mathsf{r}(i,j,d)}_i p'$, $\mathcal{H}_i = \mathcal{H}'_i$, $WB_{(k,\ell)} = WB'_{(k,\ell)}$ for all $k \in [n]$ and $\ell \in [m+1]$, and $data(tail(WB_{(i,j)})) = d$.
6. RLWE from $WB_{(i,m+1)}$: $p \xrightarrow{\mathsf{r}(i,j,d)}_i p'$, $\mathcal{H}_i = \mathcal{H}'_i$, $WB_{(k,\ell)} = WB'_{(k,\ell)}$ for all $(k,\ell) \in [n] \times [m+1]$, $Empty(WB_{(i,j)})$, the set $W_{(i,m+1)} = \{e \in B_{(i,m+1)} : \exists \mathbf{d}' \in M.\ \lambda_{(i,m+1)}(e) = \mathsf{w}(i,j,\mathbf{d}')\}$ is not empty, and $\lambda_{(i,m+1)}(max(W_{(i,m+1)})) = \mathsf{w}(i,j,\mathbf{d})$ such that $\mathbf{d}[j] = d$.
7. Read: $p \xrightarrow{\mathsf{r}(i,j,d)}_i p'$, $\mathcal{H}_i = \mathcal{H}'_i$, $WB_{(k,\ell)} = WB'_{(k,\ell)}$ for every $(k,\ell) \in [n] \times [m+1]$, $Empty(WB_{(i,j)})$, the set $W_{(i,m+1)}$ defined above is empty, and $\exists \mathbf{d} \in M$ such that $\lambda_{H_i}(\pi_i(i,j)) = \mathbf{d}$ and $\mathbf{d}[j] = d$.
8. Move pointer: $\mathbf{p} = \mathbf{p}'$, $\mathcal{B}_i = \mathcal{B}'_i$, $WB_{(k,\ell)} = WB'_{(k,\ell)}$ for every $(k,\ell) \in [n] \times [m+1]$, and $\exists j \in [m].\ \exists e \in H_i.\ \pi_i(i,j) \rightsquigarrow_{H_i} e$ and $\pi'_i = \pi_i[(k,j) \leftarrow e]_{k \in [n]}$.
9. Atomic Read-write: $p \xrightarrow{\mathsf{arw}(i,j,d,d')}_i p'$, $WB_{(k,\ell)} = WB'_{(k,\ell)}$ for all $(k,\ell) \in [n] \times [m]$, $Empty(WB_{(i,j)})$ and $Empty(WB'_{(i,j)})$ for every $j \in [m+1]$, $\pi_i(i,\ell) = max(H_i)$ for every $\ell \in [m]$, there is a $\mathbf{d} = tail(\mathcal{B}_i)$ such that $WB'_{(k',m+1)} = add(WB_{(k',m+1)}, \mathsf{w}(i,j,\mathbf{d}'))$ for all $k' \in ([n] \setminus \{i\})$, $\mathbf{d}[j] = d$, $\mathcal{B}'_i = add(\mathcal{B}_i, \mathbf{d}')$, and $\pi'_i = \pi_i[(k,\ell) \leftarrow max(H'_i)]_{k \in [n], \ell \in [m]}$ where $\mathbf{d}' = \mathbf{d}[j \leftarrow d']$.
10. Read fence: $p \xrightarrow{\mathsf{rfence}(i)}_i p'$, $WB_{(k,\ell)} = WB'_{(k,\ell)}$ for every $(k,\ell) \in [n] \times [m+1]$, $\mathcal{H}_i = \mathcal{H}'_i$, and $\pi_i(i,\ell) = max(H_i)$ for every $\ell \in [m]$.
11. Memory update: $\mathbf{p} = \mathbf{p}'$, $WB_{(k,\ell)} = WB'_{(k,\ell)}$ for every $(k,\ell) \in ([n] \times [m] \setminus \{(i,m+1)\})$, there exist $t \in [n]$, $j \in [m]$ and $\mathbf{d} \in M$ such that $head(WB_{(i,m+1)}) = \mathsf{w}(t,j,\mathbf{d})$, $WB'_{(i,m+1)} = remove(WB_{(i,m+1)}, \mathsf{w}(t,j,\mathbf{d}))$, $\mathcal{B}'_i = add(\mathcal{B}_i, \mathbf{d})$, and $\pi'_i = \pi_i[(k,j) \leftarrow max(H'_i)]_{k \in [n]}$ if $t = i$, otherwise $\pi'_i = \pi_i$.

Let us explain each case. A write operation $\mathsf{w}(i,j,d)$ is simply added to the write buffer $WB_{(i,j)}$ (as in NSW). A write fence operation $\mathsf{wfence}(i)$ can be executed only if all the store buffer $(WB_{(i,j)})_{i \in [n]}^{j \in [m]}$.

A transfer write operation is performed by the process $\mathcal{P}_i$ under NSW$^+$ as follows: (1) a write operation ω is removed from a write buffer $WB_{(i,j)}$ with $j \in [m]$, a new memory state $\mathbf{d}'$ is then computed from the last state $\mathbf{d}$ added to the history buffer $\mathcal{H}_i$ (i.e., $\mathbf{d} = tail(\mathcal{B}_i)$) if the serialization buffer $WB_{(i,m+1)}$ is empty (i.e. $Empty(WB_{(i,m+1)})$ holds) otherwise from the last event labelled by $\mathsf{w}(k,\ell,\mathbf{d})$ added to $WB_{(i,m+1)}$, and (3) a new event labelled by $\mathsf{w}(i,j,\mathbf{d}')$ is then added to all the serialization buffers $WB_{(k,m+1)}$ for all $k \in [n]$.

A read operation $\mathsf{r}(i,j,d)$ can be validated immediately (point 5) if the write buffer $WB_{(i,j)}$ is not empty and the last operation added to $WB_{(i,j)}$ precisely writes the value d on x_j (i.e., $data(tail(WB_{(i,j)})) = d$). Now, if the write buffer $WB_{(i,j)}$ is empty and the set of events in $WB_{(i,m+1)}$ associated with the process $\mathcal{P}_i$ and the variable x_j is not empty (i.e., the set $W_{i,m+1}$ defined above is not empty) (point 6), then the last event in $WB_{(i,j)}$ associated with $\mathcal{P}_i$ and x_j precisely writes the value d on x_j (i.e., $data(\lambda_{(i,m+1)}(WB_{(i,j)})) = \mathsf{w}(i,j,\mathbf{d})$ and $\mathbf{d}[j] = d$). Otherwise, (in point 7), we simply require that the value of x_j in the event pointed by $\pi_i(i,j)$ (in the history buffer $\mathcal{H}_i$ associated with $\mathcal{P}_i$) is d (i.e., $\lambda_{H_i}(\pi_i(i,j)) = \mathbf{d}$ and $\mathbf{d}[j] = d$).

A Move pointer operation of the process $\mathcal{P}_i$ can move, in nondeterministic way, the position of the pointer $\pi_i(i,j)$ associated to the variable x_j toward the tail of its history buffer $\mathcal{H}_i$ (i.e., most recent element in $\mathcal{H}_i$). Observe that we simultaneously move all the pointers $\pi_i(k,j)$ (for all $k \in [n]$) to the same element. This is only done for the purpose of having a history buffer $\mathcal{H}_i$ at the end of the computation reduced to one element. Observe also that the pointers $\pi_i(k,j)$ (for all $k \in [n] \setminus \{i\}$ and $j \in [m]$) do play any role under the NSW^+ since they will never be tested.

An atomic read-write operation $\mathsf{arw}(i,j,d,d')$, which acts as a fence on all operations of the process $\mathcal{P}_i$, can be executed only when: (1) all buffers of $\mathcal{P}_i$ are empty (i.e. $Empty(WB_{(i,\ell)})$ holds for all $\ell \in [m]$), and (2) the value of the variable x_j in the event pointed by $\pi_i(i,j)$ (which should correspond to the tail of the history buffer $\mathcal{H}_i$) is precisely d. If it is the case that a new event is added to the history buffer $\mathcal{H}_i$ labelled by the memory state $\mathbf{d}'$ obtained from the last memory state added to $\mathcal{H}_i$ by modifying the value of the variable x_j from d to d'. Moreover, all the pointers of the history buffer $\mathcal{H}_i$ are updated to this newly added event. Now, in order that the other processes take into account this new memory state, an event labelled by $\mathsf{w}(i,j,\mathbf{d}')$ is then added to all other serialization buffers $WB_{(k,)}$ (for all $k \neq i$).

A Read fence $\mathsf{rfence}(i)$ can be executed by the process $\mathcal{P}_i$ only when all the pointer of the history buffer $\mathcal{H}_i$ are pointing to the last event of $\mathcal{H}_i$.

A Memory update of the process $\mathcal{P}_i$ corresponds to remove the head element $\mathsf{w}(t,j,\mathbf{d})$ of $WB_{(i,m+1)}$, add a new event labelled by $\mathbf{d}$ to the history buffer $\mathcal{H}_i$, and update the position of the set of pointers $\pi_i(k,j)$ (with $k \in [n]$) to this newly added element if the element $\mathsf{w}(t,j,\mathbf{d})$ is performed by the process $\mathcal{P}_i$ (i.e., $t = i$).

In the following, we show that the state reachability problem for a concurrent system $\mathcal{N}$ under NSW can be reduced to its corresponding one for $\mathcal{N}$ under NSW^+.

Theorem 7. *Let $s = (\mathbf{p},\mathbf{d})$ and $s' = (\mathbf{p}',\mathbf{d}')$ be two states of $\mathcal{N}$, and let $\mathcal{H}$ and $\mathcal{H}'$ be two unitary history buffers over M such that $tail(\mathcal{H}) = \mathbf{d}$ and $tail(\mathcal{H}') = \mathbf{d}'$. Then, $\langle \mathbf{p}, \overline{\mathcal{S}_\emptyset}, \mathcal{H}\rangle \rightarrow^*_{\mathcal{N}} \langle \mathbf{p}', \overline{\mathcal{S}_\emptyset}, \mathcal{H}'\rangle$ iff $\langle \mathbf{p}, \overline{\mathcal{S}'_\emptyset}, \mathcal{H}, \ldots, \mathcal{H}\rangle \mapsto^*_{\mathcal{N}} \langle \mathbf{p}', \overline{\mathcal{S}'_\emptyset}, \mathcal{H}', \ldots, \mathcal{H}'\rangle$ where $\overline{\mathcal{S}_\emptyset}$ and $\overline{\mathcal{S}'_\emptyset}$ denotes an $[n] \times [m+1]$-dim vector of empty buffers.*

A proof of Theorem 7 can be found in Appendix C. The proof consists in showing that each computation in one of the models it is possible to associate a computation in the other model such that along these two computations we have: (1) the same sequence of memory states, and (2) the same sequence of operations performed by each of the processes, i.e., for each process, the two projections of these computations on the operations of that process are the same.

However, it is not obvious how to translate the ordering on NSW^+-configurations into one on NSW-configurations. In particular the standard proofs that show reductions between different semantics (models), where each configuration in one model is shown to be in (bi-)simulation with a configuration in the other model cannot be used here.

The state reachability problem for NSW^+: We show in the following that the state reachability problem is decidable for the NSW^+ model. As mentioned earlier, we establish this fact by proving that NSW^+'s are monotonic w.r.t. a particular WQO $\preceq$.

Let $\mathcal{N}$ be an NSW^+, and let us define the relation $\preceq$ on the configurations of $\mathcal{N}$. Consider two configurations $c = \langle \mathbf{p}, (WB_{(i,j)})_{i\in[n]}^{j\in[m+1]}, (\mathcal{H}_k)_{k\in[n]}\rangle$ and $c' = \langle \mathbf{p}', (WB'_{(i,j)})_{i\in[n]}^{j\in[m+1]}, (\mathcal{H}'_k)_{k\in[n]}\rangle$, where $WB_{(i,j)} = (B_{(i,j)}, \rightsquigarrow_{(i,j)}, \lambda_{(i,j)})$ and $WB'_{(i,j)} = (B'_{(i,j)}, \rightsquigarrow'_{(i,j)}, \lambda'_{(i,j)})$ for all i and j, and $\mathcal{H}_k = (\mathcal{B}_k, \pi_k)$ and $\mathcal{H}'_k = (\mathcal{B}'_k, \pi'_k)$ with $\mathcal{B}_k = (H_k, \rightsquigarrow_{H_k}, \lambda_{H_k})$ and $\mathcal{B}'_k = (H'_k, \rightsquigarrow_{H'_k}, \lambda_{H'_k})$ for all $k \in [n]$. Then, we consider that $c \preceq c'$ holds if

1. c and c' have the same vector of control states, i.e., $\mathbf{p} = \mathbf{p}'$,
2. the content of $WB_{(i,j)}$ is a subword of the content $WB'_{(i,j)}$, while the sequences of operations in $WB_{(i,j)}$ and $WB'_{(i,j)}$ corresponding the last operations performed every process on each of the variables are the same, i.e., for every $i \in [n]$ and $j \in [m+1]$, there is an injection $g_{(i,j)}$ from $B_{(i,j)}$ to $B'_{(i,j)}$ such that: (a) for every $e_1, e_2 \in B_{(i,j)}$, $\lambda'_{(i,j)}(g_{(i,j)}(e_1)) = \lambda_{(i,j)}(e_1)$ and $e_1 \rightsquigarrow_{(i,j)} e_2$ implies $g_{(i,j)}(e_1) \rightsquigarrow_{(i,j)} g_{(i,j)}(e_2)$, and (b) for every $k \in [n]$ and $\ell \in [m]$, if $E_{(k,\ell)} = \{e \in B_{(i,j)} \,:\, \lambda_{(i,j)}(e) \in \{\mathsf{w}(k,\ell,\mathbf{d}'), \mathsf{w}(k,\ell,d') \,|\, \mathbf{d}' \in M, d' \in D\}\}$ and $E'_{(k,\ell)} = \{e \in B'_{(i,j)} \,:\, \lambda'_{(i,j)}(e) \in \{\mathsf{w}(k,\ell,\mathbf{d}'), \mathsf{w}(k,\ell,d') \,|\, \mathbf{d}' \in M, d' \in D\}\}$, then $g_{(i,j)}(max(E_{(k,\ell)})) = max(E'_{(k,\ell)})$,
3. the content of $\mathcal{H}_k$ is a subword of the content $\mathcal{H}'_k$, while the last memory states added to $\mathcal{H}_k$ and $\mathcal{H}'_k$ are the same, and the memory states pointed by $\pi_k(i,j)$ and by $\pi'_k(i,j)$ are equal for every i and j, i.e., for every $k \in [n]$ there is an injection g_k from H_k to H'_k such that: (a) for every $e_1, e_2 \in H_k$, $\lambda_{H'_k}(g_k(e_1)) = \lambda_{H_k}(e_1)$ and $e_1 \rightsquigarrow_{H_k} e_2$ implies $g_k(e_1) \rightsquigarrow_{H'_k} g_k(e_2)$, (b) for every $i \in [n]$ and $j \in [m]$, $g_k(\pi_k(i,j)) = \pi'_k(i,j)$, and (c) $g_k(max(H_k)) = max(H'_k)$.

By Higman's lemma (the subword relation is a well quasi-ordering) and standard composition properties of well quasi-orderings, it is easy to prove the following fact.

Lemma 1 (WQO). *The relation $\preceq$ is a WQO on the set of NSW^+-configurations of $\mathcal{N}$.*

Given a set C of NSW^+-configurations, we define $C\!\uparrow = \{c' \,:\, c \in C \wedge c \preceq c'\}$, i.e. $C\!\uparrow$ is the set of configurations generated by those in C via $\preceq$. A set C is *upward closed* w.r.t. $\preceq$ if $C\!\uparrow = C$. Since $\preceq$ is a well-quasi ordering, we can show that every upward closed set of configurations has finite set of minimal elements. We can exploit this property to derive an algorithm for the state reachability problem for NSW^+ (by applying the methodology proposed in [1]. The first property we need to prove is that the transition relation NSW^+ is compatible with $\preceq$. Then, we can prove the following important fact:

Lemma 2 (Monotonicity). *For every configurations c_1, c_2, c'_1 of a $\mathcal{N}$ such that $c_1 \mapsto_{\mathcal{N}} c_2$ and $c_1 \preceq c'_1$, there exists a configuration c'_2 such that $c'_1 \mapsto^*_{\mathcal{N}} c'_2$ and $c_2 \preceq c'_2$.*

The proof of Lemma 2 can be found in Appendix D.

From [1] we know that monotonicity ensures that if a set of configurations C is $\preceq$-upward closed, then the set of its predecessors $pre_{\mathcal{N}}(C) = \{c : c' \in C \land c \mapsto_{\mathcal{N}} c'\}$ is also $\preceq$-upward closed, and since upward closed sets w.r.t. WQO are finitely defined by their minimals, this fact allows to deduce that the iterative computation of the set of all predecessors of C (i.e., $pre^{*}_{\mathcal{N}}(C)$) eventually terminates. We only need to show that:

Lemma 3 (Effectiveness). *Given a finite set M of $\preceq$-minimals of a $\preceq$-upward closed set C, the (finite) set of $\preceq$-minimals of $pre_{\mathcal{N}}(C)$ is effectively computable from M.*

Showing that we can effectively compute the $\preceq$-minimals of $pre(C)$ can be performed in a similar way as for lossy channel machines [1].

Then, from the three lemmas above and [1], we deduce the following fact:

Theorem 8. *The state reachability problem for* NSW^{+} *is decidable.*

As a corollary of Theorem 7 and Theorem 8, we obtain the main result of this paper:

Corollary 1. *The state reachability problem for* NSW *is decidable.*

7 Nonatomic Writes Cause Undecidability

So far, we have considered only models that do not contain the RRWE (read remote writes early) relaxation. In this section, we show that adding RRWE to NSW makes the reachability problem undecidable. The RRWE relaxation allows a processor to read other processors' writes even if they are not globally visible yet. This makes writes non-atomic and can be detected by the IRIW litmus test (Fig. 3). IRIW is not possible in NSW as defined earlier. However, if we change the model to allow a read operation of $\mathcal{P}_i$ on a variable x_j to be validated by the last write operation issued by $\mathcal{P}_k$ (with $k \neq i$) on x_j, although this last write operation has not been yet updated, it becomes possible.

$x = y = 0$			
$\mathcal{P}_1$	$\mathcal{P}_2$	$\mathcal{P}_3$	$\mathcal{P}_4$
(1) $\mathsf{r}(x,1)$	(4) $\mathsf{r}(y,1)$	(7) $\mathsf{w}(x,1)$	(8) $\mathsf{w}(y,1)$
(2) rfence	(5) rfence		
(3) $\mathsf{r}(y,0)$	(6) $\mathsf{r}(x,0)$		
$x = y = 1$			

Fig. 3. The IRIW (Independent Reads of Independent Writes) Litmus Test. $\mathcal{P}_3$ writes 1 to x and $\mathcal{P}_4$ writes 1 to y. In parallel, $\mathcal{P}_1$ observes that x has been modified before y, whereas $\mathcal{P}_2$ observes that y is modified before x.

An operational model An operational model for NSW with the RRWE relaxation can be defined as an extension of the one defined in Sec. 4. The idea is to add to the event structure $\mathcal{S} = (E, \rightsquigarrow, \lambda)$ a mapping $\sigma : [n] \times [m] \to E \cup \{\perp\}$, with $\perp \notin E$, that associates with each process and variable, either a pointer on some event of the structure, or $\perp$ when it is not defined. The pointer $\sigma(i, j)$ defines an event e such that every future read operation of $\mathcal{P}_i$ on the variable x_j should not take its value from a write event that is $\rightsquigarrow$-smaller than e. The intuition is that the validation of successive reads by the same process on a same variable should be done in a coherent way, i.e., the writes from which they read their values should occur in the same order. If $\sigma(i, j)$ points to some event e in the event structure, then e corresponds to the write event from which the last read performed by the process $\mathcal{P}_i$ on the variable x_j took its value. The fact that $\sigma(i, j) = \perp$ means that either $\mathcal{P}_i$ has never read a value from x_j, or the last write operation on x_j (issued by some other process) that has validated a read of $\mathcal{P}_i$ has already been updated.

Then, to validate a read operation of $\mathcal{P}_i$ on x_j using the RRWE, an event e must be found such that (1) e does not occur before the event $e' = \sigma(i, j)$ or any read/write event of P_i on x_j, and (2) e is the last write operation on x_j of $\mathcal{P}_k$ different from $\mathcal{P}_i$. If this is the case, then $\sigma(i, j)$ is updated to e and constraints are added to ensure that (i) e should be executed after the event e' and any read/write event of P_i on x_j, and (ii) e should be executed before all writes/reads by $\mathcal{P}_i$ on x_j coming after the validated read operation. When a write event is executed and exits the event structure $\mathcal{S}$, if this write event is pointed by $\sigma(i, j)$, then $\sigma(i, j)$ is set to $\perp$. $\mathcal{P}_i$ can perform a RLWE on x_j only if the event associated to the last write operation of P_i on x_j does not occur before $\sigma(i, j)$.

An atomic read-write operation $\mathsf{arw}(i, j, d, d')$ can be executed only when no pending reads on the same variable still exist in the structure $\mathcal{S}$, i.e., $\sigma(i, j) = \perp$. The reason is that operations on the same variable cannot be reordered. Finally, all the other operations are defined as in Sec. 4 while keeping the pointers unchanged.

As an example, consider the IRIW litmus test (Fig. 3). Starting from the memory state $(x = 0, y = 0)$ and an empty event structure $\mathcal{S}$, the execution of the writes (7) and (8) by $\mathcal{P}_3$ and $\mathcal{P}_4$ adds two events e_1 and e_2 to $\mathcal{S}$ labeled by $\mathsf{w}(3, x, 1)$ and $\mathsf{w}(4, y, 1)$, respectively. Then, $\mathcal{P}_1$ and $\mathcal{P}_2$ can execute their read operations (1) and (4) that are validated using the RRWE relaxation, and set the pointers $\sigma(1, x)$ and $\sigma(2, y)$ to e_1 and e_2. At this point, read_fences (2) and (5) can be executed, and then, the read operations (3) and (6) can be executed since they can be validated w.r.t. the content of the main memory. Finally, the write operations corresponding to the events e_1 and e_2 stored in $\mathcal{S}$ are committed to the main memory, and this yields the memory state $(x = 1, y = 1)$.

We can prove that the addition of the RRWE to NSW models leads to the undecidability of the state reachability problem. The proof is by a reduction of PCP.

Theorem 9. *The state reachability problem for* $\mathsf{NSW} \cup \{\mathsf{RRWE}\}$ *is undecidable.*

8 Conclusion and Future Work

We have sharpened the decidability boundary of the reachability problem for weak memory models by (1) introducing a model NSW which supports many important relaxations (delay writes, perform reads early, allow partial fences) yet has a decidable reachability problem, and (2) showing that the read-write relaxation and the non-atomic-stores-relaxation are problematic (cause non-decidability) if added to TSO or NSW, respectively.

Besides decidability, our work contributes in clarifying the effects and the power of common relaxations existing in weak memory models. It provides an insight on the formal models needed to reason about these relaxations, which can be useful for other formal algorithmic verification approaches, including approximate analyses. Notice that the models we introduce in Sections 4 and 5 can be also considered in the case of an infinite data domain, and the relationship between them still holds in the same manner. It is only when we address the decidability issue that we need to restrict ourselves to a finite data domain.

Future work may address the question of whether the boundary can be sharpened further by considering finer distinctions of the $\mathsf{r} \rightarrow \mathsf{w}$ relaxation, say by making it conditional on the absence of control- or data-dependencies. Moreover, we would like to explore the effect of non-atomic stores in more detail, such as whether it causes undecidability in weaker forms (e.g. if caused by static memory hierarchies) or if added to TSO rather than NSW.

References

1. Parosh Aziz Abdulla, Karlis Cerans, Bengt Jonsson, and Yih-Kuen Tsay. General decidability theorems for infinite-state systems. In *LICS*, pages 313–321, 1996.
2. S. Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. *Computer*, 29(12):66–76, 1996.
3. M. F. Atig, A. Bouajjani, S. Burckhardt, and M. Musuvathi. On the verification problem for weak memory models. In *POPL*, pages 7–18. ACM, 2010.
4. M. F. Atig, A. Bouajjani, and G. Parlato. Getting rid of store-buffers in TSO analysis. In *CAV*, 2011.
5. H. Boehm. WG21/N2176 memory model rationales. http://open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2176.html#dependencies, March 2007.
6. H. Boehm and S. Adve. Foundations of the C++ concurrency memory model. In *PLDI*, pages 68–78, 2008.
7. A. Bouajjani, R. Meyer, and E. Möhlmann. Deciding robustness against total store ordering. In *ICALP*, 2011.
8. S. Burckhardt, R. Alur, and M. Martin. CheckFence: Checking consistency of concurrent data types on relaxed memory models. In *PLDI*, pages 12–21, 2007.
9. S. Burckhardt and M. Musuvathi. Effective program verification for relaxed memory models. In *Computer-Aided Verification (CAV)*, pages 107–120, 2008. Extended Version as Tech Report MSR-TR-2008-12, Microsoft Research.
10. S. Burckhardt, M. Musuvathi, and V. Singh. Verifying local transformations on relaxed memory models. In *CC'10*, pages 104–123, 2010.
11. J. Burnim, K. Sen, and C. Stergiou. Testing concurrent programs on relaxed memory models. Technical Report UCB/EECS-2010-32, EECS Department, University of California, Berkeley, Mar 2010.
12. C. Chen, W. Chen, V. Sreedhar, R. Barik, V. Sarkar, and G. Gao. Establishing causality as a desideratum for memory models and transformations of parallel programs. Technical report, University of Delaware, 2010.

13. K. Gharachorloo, A. Gupta, and J. Hennessy. Performance evaluation of memory consistency models for shared-memory multiprocessors. In *ASPLOS'91*, pages 245–257, 1991.
14. M. Kuperstein, M. Vechev, and E. Yahav. Automatic inference of memory fences. In *FM-CAD*, pages 111–119, October 2010.
15. M. Kuperstein, M. Vechev, and E. Yahav. Partial-coherence abstractions for relaxed memory models. In *PLDI*, San Jose, CA, Jun 2011.
16. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comp.*, C-28(9):690–691, 1979.
17. A. Linden and P. Wolper. An automata-based symbolic approach for verifying programs on relaxed memory models. In *SPIN*, 2010.
18. A. Linden and P. Wolper. A verification-based approach to memory fence insertion in relaxed memory systems. In *SPIN*, 2011.
19. S. Mador-Haim, R. Alur, and M. Martin. Generating litmus tests for contrasting memory consistency models. In *Computer Aided Verification*, pages 273–287, 2010.
20. J. Manson, W. Pugh, and S.V. Adve. The java memory model. In *POPL*, pages pages = 378–391, 378–391, 2005.
21. S. Owens. Reasoning about the implementation of concurrency abstractions on x86-tso. In *ECOOP*, volume 6183 of *LNCS*, pages 478–503. Springer, 2010.
22. S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-TSO. In *TPHOL*, 2009.
23. E. L. Post. A variant of a recursively unsolvable problem. *Bull. of the American Mathematical Society*, 52:264–268, 1946.
24. S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. Understanding POWER multiprocessors. In *PLDI*, San Jose, CA, Jun 2011.
25. J. Sevcik. Safe optimisations for shared-memory concurrent programs. In *PLDI*, pages 306–316, 2011.
26. J. Sevcik, V. Vafeiadis, F. Z. Nardelli, S. Jagannathan, and P. Sewell. Relaxed-memory concurrency and verified compilation. In *POPL*, pages 43–54, 2011.
27. P. Sewell, S. Sarkar, S. Owens, F. Nardelli, and M. Myreen. x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53, 2010.
28. Y. Yang, G. Gopalakrishnan, and G. Lindstrom. UMM: an operational memory model specification framework with integrated model checking capability. *Concurrency and Computation: Practice and Experience*, 17(5-6):465–487, 2005.

A Proof of Theorem 2

The proof is done by a reduction of PCP (Post's Correspondence Problem), well-known to be undecidable [23], to our problem. It follows a similar schema as the one used in [3] for $\mathsf{TSO} \cup \{\mathsf{r} \to \mathsf{r/w}\}$, although the encoding is quite different.

We recall that PCP consists in, given two finite lists $\{u_1, \ldots, u_n\}$ and $\{v_1, \ldots, v_n\}$ of nonempty words over some alphabet Σ, checking whether there is a sequence of indices $i_1, \ldots, i_k \in [n]$ such that $u_{i_1} \cdots u_{i_k} = v_{i_1} \cdots v_{i_k}$.

Then, let $\{u_1, \ldots, u_n\}$ and $\{v_1, \ldots, v_n\}$ be an instance of PCP. We construct a system $\mathcal{N}$ with four processes $\mathcal{P}_1, \cdots, \mathcal{P}_4$ sharing a set of six variables $X = \{x_1, x_2, x_3, x_4, x_5, x_6\}$ such that, two specific states in $\mathcal{N}$ under $\mathsf{M} = \mathsf{TSO} \cup \{\mathsf{r} \to \mathsf{r/w}\}$ are related by the reachability relation $Reach^{\mathsf{M}}_{\mathcal{N}}$ if and only if PCP has a solution for the considered instance.

The idea of the reduction is as follows:

– Process $\mathcal{P}_1$ has a special state p_1. $\mathcal{P}_1$ guesses the solution of PCP as a sequence of indices $i_1, \ldots, i_k$ and performs iteratively a sequence of operations from the state p_1: It writes successively to x_1 the index i_j, and reads from x_2 for j ranging from 1 to k. Moreover, each write (resp. read) operation of any process to (resp. from) a variable is followed by a write (resp. read) operation of the marker $\sharp$. Hence, the process $\mathcal{P}_1$ has the following computation trace $\tau_1$from p_1 to p_1:

$$\mathsf{w}(1,1,i_1)\mathsf{w}(1,1,\sharp)\mathsf{r}(1,2,i_1)\mathsf{r}(1,2,\sharp)\mathsf{w}(1,1,i_2)\mathsf{w}(1,1,\sharp)$$
$$\mathsf{r}(1,2,i_2)\mathsf{r}(1,2,\sharp)\cdots\mathsf{w}(1,1,i_k)\mathsf{w}(1,1,\sharp)\ \mathsf{r}(1,2,i_k)\mathsf{r}(1,2,\sharp)$$

– Process $\mathcal{P}_2$ has a special state p_2. $\mathcal{P}_2$ guesses the solution of PCP as a sequence of indices $i_1, \ldots, i_k$ and performs iteratively a sequence of operations from the state p_2: It writes successively to x_3 the index i_j, and reads from x_4 for j ranging from 1 to k. Moreover, each write (resp. read) operation of any process to (resp. from) a variable is followed by a write (resp. read) operation of the marker $\sharp$. Hence, the process $\mathcal{P}_2$ has the following computation trace τ_2 from p_2 to p_2:

$$\mathsf{w}(2,3,i_1)\mathsf{w}(2,3,\sharp)\mathsf{r}(2,4,i_1)\mathsf{r}(2,4,\sharp)\mathsf{w}(2,3,i_2)\mathsf{w}(2,3,\sharp)$$
$$\mathsf{r}(2,4,i_2)\mathsf{r}(2,4,\sharp)\cdots\mathsf{w}(2,3,i_k)\mathsf{w}(2,3,\sharp)\ \mathsf{r}(2,4,i_k)\mathsf{r}(2,4,\sharp)$$

– Process $\mathcal{P}_3$ has a special state p_3. $\mathcal{P}_3$ guesses the solution of PCP as a sequence of indices $i_1, \ldots, i_k$ and performs iteratively a sequence of operations from p_3: It (1) writes successively to x_5 the sequence of symbols of u_{i_j}, (2) reads from x_6 the sequence of symbols of u_{i_j}, (3) writes to x_2 the index i_j, and (4) reads i_j from x_3, for j ranging from 1 to k. Moreover, each write (resp. read) operation of any process to (resp. from) a variable is followed by a write (resp. read) operation of the marker $\sharp$. Hence, the process $\mathcal{P}_3$ has the following computation trace τ_3 from p_3 to p_3:

$$\omega_1\mathsf{w}(3,2,i_1)\mathsf{w}(3,2,\sharp)\mathsf{r}(3,3,i_1)\mathsf{r}(3,3,\sharp)\omega_2\mathsf{w}(3,2,i_2)\mathsf{w}(3,2,\sharp)$$
$$\mathsf{r}(3,3,i_2)\mathsf{r}(3,3,\sharp)\cdots\omega_k\mathsf{w}(3,2,i_k)\mathsf{w}(3,2,\sharp)\mathsf{r}(3,3,i_k)\mathsf{r}(3,3,\sharp)$$

where for all $j \in [k]$, the sequence of operations ω_j is defined as follows:

$$\omega_j = \mathsf{w}(3,5,a_1^j)\mathsf{w}(3,5,\sharp)\mathsf{r}(3,6,a_1^j)\mathsf{r}(3,6,\sharp)\mathsf{w}(3,5,a_2^j)\mathsf{w}(3,5,\sharp)$$
$$\mathsf{r}(3,6,a_2^j)\mathsf{r}(3,6,\sharp)\cdots\mathsf{w}(3,5,a_{k_j}^j)\mathsf{w}(3,5,\sharp)\ \mathsf{r}(3,6,a_{k_j}^j)\mathsf{r}(3,6,\sharp)$$

with $u_{i_j} = a_1^j a_2^j \cdots a_{k_j}^j$.

– Process $\mathcal{P}_4$ has a special state p_4. $\mathcal{P}_4$ guesses the solution of PCP as a sequence of indices $i_1,\ldots,i_k$ and performs iteratively a sequence of operations from p_4: It (1) reads successively from x_5 the sequence of symbols of v_{i_j}, (2) writes to x_6 the sequence of symbols of v_{i_j}, (3) writes to x_4 the index i_j, and (4) reads i_j from x_1, for j ranging from 1 to k. Moreover, each write (resp. read) operation of any process to (resp. from) a variable is followed by a write (resp. read) operation of the marker $\sharp$. Hence, the process $\mathcal{P}_4$ has the following computation trace τ_4 from p_4 to p_4:

$$\omega_1'\mathsf{w}(4,4,i_1)\mathsf{w}(4,4,\sharp)\mathsf{r}(4,1,i_1)\mathsf{r}(4,1,\sharp)\omega_2\mathsf{w}(4,4,i_2)\mathsf{w}(4,4,\sharp)$$
$$\mathsf{r}(4,1,i_2)\mathsf{r}(4,1,\sharp)\cdots\nu_k\mathsf{w}(4,4,i_k)\mathsf{w}(4,4,\sharp)\mathsf{r}(4,1,i_k)\mathsf{r}(4,1,\sharp)$$

where for all $j \in [k]$, the sequence of operations ν_j is defined as follows:

$$\omega_j' = \mathsf{r}(4,5,b_1^j)\mathsf{r}(4,5,\sharp)\mathsf{w}(4,6,b_1^j)\mathsf{w}(4,6,\sharp)\mathsf{r}(4,5,b_2^j)\mathsf{r}(4,5,\sharp)$$
$$\mathsf{w}(4,6,b_2^j)\mathsf{w}(4,6,\sharp)\cdots\mathsf{r}(4,5,b_{\ell_j}^j)\mathsf{r}(4,5,\sharp)\ \mathsf{w}(4,6,b_{\ell_j}^j)\mathsf{w}(4,6,\sharp)$$

with $v_{i_j} = b_1^j b_2^j \cdots b_{\ell_j}^j$.

Observe that the insertion of the markers allows to ensure that a written value to a variable by one of the processes can be read at most once by the other processes. Observe that the concurrent system can be defined formally following a similar schema as the one used in Sec. 7.

Then, we prove that PCP has a solution if and only if it is possible $Reach_{\mathcal{N}}^{\mathsf{M}}(s,s)$ holds with $s = \langle \mathbf{p},\mathbf{d}\rangle$ such that $\mathbf{p}[i] = p_i$ for all $i \in [n]$ and $\mathbf{d}[j] = \sharp$ for all $j \in [m]$. In other words, there is a computation trace of $\mathcal{N}$ if and only if all the processes have guessed the same sequence of indices and that the sequences $u_{i_1}\cdots u_{i_k}$ and $v_{i_1}\cdots v_{i_k}$, guessed respectively by the processes $\mathcal{P}_3$ and $\mathcal{P}_4$, are the same.

The "only if direction" can be shown by constructing a trace $\tau_{3,4}$ from the shuffle of τ_3 and τ_4 such that: (1) any write of the process $\mathcal{P}_3$ to the variable x_5 should be immediately followed by its corresponding read operation of the process $\mathcal{P}_3$ to the same variable, and (2) any write of the process $\mathcal{P}_4$ to the variable x_6 should be immediately followed by its corresponding read operation of the process $\mathcal{P}_4$ to the variable x_6.

Let $\tau_{3,4}'$ be the projection of $\tau_{3,4}$ on the write operations of the process $\mathcal{P}_3$ to the variable x_2 and the read operations of the process $\mathcal{P}_4$ to the variable x_1. Then, we can construct from $\tau_{3,4}'$ the trace $\tau_1'$of the process $\mathcal{P}_1$ by: (1) replacing any write operation in $\tau_{3,4}'$ of the form $\mathsf{w}(3,2,d)$ by a read operation of the form $\mathsf{r}(1,2,d)$, and (2) replacing any read operation in $\tau_{3,4}'$ of the form $\mathsf{r}(4,1,d)$ by a write operation of the form $\mathsf{w}(1,1,d)$. It can be seen that τ_1' is in $[\{\tau_1\}]_R$ where R is the set of rewrite rules over traces defining the memory model M. In fact, this is based on the fact that write operations of process $\mathcal{P}_1$ can overtake its read operations and vice-versa. Thus, we can construct the computation trace $\tau_{1,3,4}$ from the shuffle of τ_1' and $\tau_{3,4}$ such that: (1) any write of the process $\mathcal{P}_3$ to the variable x_2 should be immediately followed by its corresponding read operation of the process $\mathcal{P}_1$ to the same variable, and (2) any write of the process $\mathcal{P}_1$ to the variable x_1 should be immediately followed by its corresponding read operation of the process $\mathcal{P}_4$ to the variable x_1.

Let $\tau'_{1,3,4}$ be the projection of $\tau_{1,3,4}$ on the write operations of the process $\mathcal{P}_4$ to the variable x_4 and the read operations of the process $\mathcal{P}_3$ to the variable x_3. Then, we can construct from $\tau'_{1,3,4}$ the trace $\tau'_2$of the process $\mathcal{P}_2$ by: (1) replacing any write operation in $\tau'_{1,3,4}$ of the form $\mathsf{w}(4,4,d)$ by a read operation of the form $\mathsf{r}(2,4,d)$, and (2) replacing any read operation in $\tau'_{1,3,4}$ of the form $\mathsf{r}(3,3,d)$ by a write operation of the form $\mathsf{w}(2,3,d)$. It can be seen that τ'_2 is in $[\{\tau_2\}]_R$. Thus, we can construct the computation trace τ from the shuffle of τ'_2 and $\tau_{1,3,4}$ such that: (1) any write of the process $\mathcal{P}_4$ to the variable x_4 should be immediately followed by its corresponding read operation of the process $\mathcal{P}_2$ to the same variable, and (2) any write of the process $\mathcal{P}_2$ to the variable x_3 should be immediately followed by its corresponding read operation of the process $\mathcal{P}_3$ to the variable x_3.

Finally, it is easy to see that $\mathbf{d}[\tau\rangle\mathbf{d}$ and hence that $Reach^{\mathsf{M}}_{\mathcal{N}}(s,s)$ holds with $s = \langle\mathbf{p},\mathbf{d}\rangle$ such that $\mathbf{p}[i] = p_i$ for all $i \in [n]$ and $\mathbf{d}[j] = \sharp$ for all $j \in [m]$.

The argument for the reverse direction is the following: If there is a computation trace τ such that $\mathbf{d}[\tau\rangle\mathbf{d}'$, then it can be seen that, due to the fact that a read can validate at most one write, the following facts hold:

- The sequence of read symbols by the process $\mathcal{P}_1$ from the variable x_2 is a subword of the sequence of written symbols by the process $\mathcal{P}_3$ the same variable.
- The sequence of read symbols by the process $\mathcal{P}_2$ from the variable x_4 is a subword of the sequence of written symbols by the process $\mathcal{P}_4$ the same variable.
- The sequence of read symbols by the process $\mathcal{P}_3$ from the variable x_3 is a subword of the sequence of written symbols by the process $\mathcal{P}_2$ the same variable.
- The sequence of read symbols by the process $\mathcal{P}_4$ from the variable x_1 is a subword of the sequence of written symbols by the process $\mathcal{P}_1$ the same variable.

From the above facts, we can conclude that all the processes have guessed that same sequence of indices $i_1,\ldots,i_k$ since the sequence of read symbols by each process from some variable is the same as the its sequence of written symbols to the same variable.

Furthermore, the sequence of read symbols by process $\mathcal{P}_3$ from the variable x_6 is a subword of the sequence of written symbols by $\mathcal{P}_4$ to the same variable, and the sequence of read symbols by process $\mathcal{P}_4$ from the variable x_5 is a subword of the sequence of written symbols by $\mathcal{P}_3$ to the same variable. Hence, we have $u_{i_1}\cdots u_{i_k} = v_{i_1}\cdots v_{i_k}$.

These facts imply that the processes have indeed guessed the same (right) solution to the given instance of PCP.

B Proof of Theorem 3

Assume that there is a computation having a causality relation containing a cycle C. Then, for every $i \in [n]$, let $E_i(C)$ be the projection of C on the events of process $\mathcal{P}_i$. By definition of the program order, for each process i, the set $E_i(C)$ is totally ordered w.r.t. the relation po. If $E_i(C)$ is nonempty, let μ_i denote its po-maximal element. It can be seen that μ_i corresponds necessarily to a write event. Indeed, by definition of the causality relation, a read event can only be the source of a dep-transition (which is subset of the po relation), and therefore if we assume that μ_i is a read, this would contradict its maximality in $E_i(C)$. Again by definition of the causality relation, a write event can only be the source of a rf-transition. Therefore, whenever $E_i(C)$ is nonempty, its maximal element μ_i has a rf-transition to some read event r of some process $\mathcal{P}_j$. Suppose that $j = i$. Since, μ_i is maximal in $E_i(C)$, we have necessarily $r \to_{\mathsf{po}} \mu_i$. But the fact that $\mu_i \to_{\mathsf{rf}} r$ (by definition of r), means that the read r has been executed *after* the write μ_i. This contradicts the fact that writes cannot overtake reads in NSW. Consequently, we must have $j \neq i$, which means that μ_i is an exit point from $E_i(C)$ by the cycle C. Moreover, a write event can only be the target of a dep-transition. Since μ_i is in a cycle, there is necessarily an entry point to $E_i(C)$, and this entry point must be a read event.

Similarly, it is possible to show that the po-minimal element of $E_i(C)$ is necessarily a read event since a write can only be the target of a dep-transition, and the existence of such a transition would contradict minimality in $E_i(C)$.

Therefore, the elements of each $E_i(C)$, for any i, can be partitioned into entry points (that are all read events), and exit points (that are all write events), with the property that after each entry point there is at least an exit point, and before each exit point there is at least an entry point. (We use "before" and "after" w.r.t. po.) Then, C can be seen as a sequence of entry and exit points to the different sets $E_i(C)$. Let $r_0 w_0 r_1 w_1 \cdots r_{\ell-1}, w_{\ell-1} r_0$ be this sequence, where the r_i's are (entry) read events, and the w_i's are (exit) write events. Notice that for every $i \in \{0,\ldots,\ell-1\}$, we have $r_i \to_{\mathsf{dep}} w_i$, and $w_i \to_{\mathsf{rf}} r_{(i+1)\,mod\,\ell}$.

Now, let us assume that the relaxation $\mathsf{r} \to \mathsf{w}$ has never been applied in the considered computation, and consider the event r_0. We know that its execution has been preceded by the execution of $w_{\ell-1}$ (due to the read-from relation). Then, since by our assumption writes have never overtaken reads, $r_{\ell-1}$ must have been executed before $w_{\ell-1}$, and therefore before r_0. By extending this reasoning to the whole sequence, we deduce that w_0 must have been executed before r_0, which contradicts the assumption that $\mathsf{r} \to \mathsf{w}$ was not used to relax program order.

C Proof of Theorem 7

The if direction: Let us introduce some notations that will be used below. Given a NSW^+ configuration c of the form $\langle \mathbf{p}, (WB_{(i,j)})_{i\in[n]}^{j\in[m+1]}, \mathcal{H}_1, \ldots, \mathcal{H}_n \rangle$, we define $state(c) = \mathbf{p}$, $buffer_{(i,j)}(c) = WB_{(i,j)}$ for all $i \in [n]$ and $j \in [m+1]$, and $history_k(c) = \mathcal{H}_k$ for all $k \in [n]$.

Given a history buffer $\mathcal{H} = (\mathcal{B}, \pi)$, we use $Event(\mathcal{H})$ to denote the event structure $\mathcal{B}$ and $Pointer(\mathcal{H})$ to denote the pointer π.

Given a content of the serialization buffer $WB_{(i,m+1)} = (B_{(i,m+1)}, \rightsquigarrow_{(i,m+1)}, \lambda_{(i,m+1)})$ under NSW^+ for some $i \in [n]$, we define the serialization buffer $Ser_i(WB_{(i,m+1)}) = (B'_{(i,m+1)}, \rightsquigarrow'_{(i,m+1)}, \lambda'_{(i,m+1)})$ under NSW as follows: (1) $B'_{(i,m+1)} = \{e \in B_{(i,m+1)} \,|\, \exists j \in [m], \mathbf{d} \in M.\ \lambda_{(i,m+1)}(e) = \mathsf{w}(i,j,\mathbf{d})\}$, (2) for every $e, e' \in B'_{(i,m+1)}$, we have $e \rightsquigarrow'_{(i,m+1)} e'$

if and only if $e \leadsto'_{(i,m+1)} e'$, and (3) for every $e \in B'_{(i,m+1)}$, if $\lambda_{(i,m+1)}(e) = \mathsf{w}(i, j, \mathbf{d})$ for some $j \in [m]$ and $\mathbf{d} \in M$, then $\lambda_{(i,m+1)}(e) = \mathsf{w}(i, j, \mathbf{d}[j])$. Intuitively, the serialization buffer $WB'_{(i,m+1)}$ is built from $WB_{(i,m+1)}$ by keeping only the events associated with the process $\mathcal{P}_i$ and labeling these events by their corresponding write operations.

Let $c_1, \ldots, c_n$ be an n-tuple of NSW^+-configurations, we say that $c_1, \ldots, c_n$ are synchronized over their history buffers (or $\mathit{Synchronized}(c_1, \ldots, c_n)$ holds) if and only if for every $i, k \in [n]$, we have $\mathit{Event}(\mathit{history}_i(c_i)) = \mathit{Event}(\mathit{history}_k(c_k))$ (i.e., the event structure associated to the history buffer of the process $\mathcal{P}_i$ in the configuration c_i is the same as the event structure associated to the history buffer of the process $\mathcal{P}_k$ in the configuration c_k).

We define also the mapping $f_{\mathsf{nsw+2nsw}}$ from n-tuple of NSW^+ configurations to NSW-configurations as follows: Given a n-tuple $c_1, \ldots, c_n$ of NSW-configurations such that $\mathit{Synchronized}(c_{i_1}, \ldots, c_{i_n})$ holds, we define the NSW-configuration $f_{\mathsf{nsw+2nsw}}(c_1, \ldots, c_n) = \langle \mathbf{p}, (WB_{(i,j)})_{i\in[n]}^{j\in[m+1]}, \mathcal{H} = (\mathcal{B}, \pi) \rangle$ as follows:

- For every $i \in [n]$, we have $\mathbf{p}[i] = \mathit{state}(c_i)[i]$. This means that the state of the process $\mathcal{P}_i$ in $f_{\mathsf{nsw+2nsw}}(c_1, \ldots, c_n)$ is the same as the state of $\mathcal{P}_i$ in the configuration c_i.
- For every $i \in [n]$ and $j \in [m]$, we have $WB_{(i,j)} = \mathit{buffer}_{(i,j)}(c_i)$. This means that the content of the write buffer $WB_{(i,j)}$ of the process $\mathcal{P}_i$ in the configuration $f_{\mathsf{nsw+2nsw}}(c_1, \ldots, c_n)$ is the same as the content of the write buffer $WB_{(i,j)}$ of $\mathcal{P}_i$ in the configuration c_i.
- For every $i \in [n]$, we have $WB_{(i,m+1)} = \mathit{Ser}_i(\mathit{buffer}_{(i,m+1)}(c_i))$. This means that the content of the write buffer $WB_{(i,m+1)}$ of the process $\mathcal{P}_i$ in the configuration $f_{\mathsf{nsw+2nsw}}(c_1, \ldots, c_n)$ is the same as the content of the write buffer $WB_{(i,m+1)}$ of $\mathcal{P}_i$ in the configuration c_i modulo the function Ser_i defined above.
- The event structure $\mathcal{B}$ is equal to $\mathit{Event}(\mathit{history}_i(c_i))$ for all $i \in [n]$ (which is well-defined since $c_1, \ldots, c_n$ are synchronized over their history buffers). This means that the event structure $\mathcal{B}_i$ is the same as the event structure of the history buffer associated to the process $\mathcal{P}_i$ in the configuration c_i.
- For every $i \in [n]$ and $j \in [m]$, we have $\pi(i, j) = \mathit{Pointer}(\mathit{history}_i(c_i))(i, j)$. This means that the process $\mathcal{P}_i$ is pointing to the sam events in the configurations $f_{\mathsf{nsw+2nsw}}(c_1, \ldots, c_n)$ and c_i.

Now, let us assume that there is a computation ρ of the form $\langle \mathbf{p}, \overline{\mathcal{S}'_\emptyset}, \mathcal{H}, \ldots, \mathcal{H} \rangle \mapsto^*_{\mathcal{N}} \langle \mathbf{p}', \overline{\mathcal{S}'_\emptyset}, \mathcal{H}', \ldots, \mathcal{H}' \rangle$. This means that there is a sequence $c_0, \ldots, c_k$ of NSW^+ such that $c_0 = \langle \mathbf{p}, \overline{\mathcal{S}'_\emptyset}, \mathcal{H}, \ldots, \mathcal{H} \rangle$, $c_k = \langle \mathbf{p}', \overline{\mathcal{S}'_\emptyset}, \mathcal{H}', \ldots, \mathcal{H}' \rangle$, and $c_{i-1} \mapsto_{\mathcal{N}} c_i$ for all $i \in [k]$. Let $L_{max} \in \mathbb{N}$ be the number of events in the history buffer $\mathcal{H}'$ and L_{min} be the number of events in the initial history buffer $\mathcal{H}$. For every $\ell \in \{L_{min}, \ldots, L_{max}\}$, we associate two n-tuple of NSW^+-configurations $\mathbf{v}_\ell^{min}$ and $\mathbf{v}_\ell^{max}$ such that:

- $\mathbf{v}_\ell^{min} = (c_{i_1}, c_{i_2}, \ldots, c_{i_n})$ such that for every $t \in [n]$, $i_t \in \{0, \ldots, k\}$ is the minimal index such that, in the configuration c_{i_t}, the event structure of the history buffer $\mathit{history}_t(c_{i_t})$ of the process $\mathcal{P}_t$ contains exactly ℓ events
- $\mathbf{v}_\ell^{max} = (c_{i'_1}, c_{i'_2}, \ldots, c_{i'_n})$ such that for every $t \in [n]$, $i'_t \in \{0, \ldots, k\}$ is the maximal index such that, in the configuration $c_{i'_t}$, the event structure of the history buffer $\mathit{history}_t(c_{i'_t})$ of the process $\mathcal{P}_t$ contains exactly ℓ events

Observe that, by definition, the n-tuples $\mathbf{v}_\ell^{min}$ and $\mathbf{v}_\ell^{max}$ are synchronized (i.e., *Synchronized*$(\mathbf{v}_\ell^{min})$ and *Synchronized*$(\mathbf{v}_\ell^{max})$ hold). Let us prove our first lemma:

Lemma 4. *For every $\ell \in \{L_{min}, \dots, L_{max}\}$, $f_{\mathsf{nsw+2nsw}}(\mathbf{v}_\ell^{min}) \to_{\mathcal{N}}^* f_{\mathsf{nsw+2nsw}}(\mathbf{v}_\ell^{max})$.*

Proof. Let us assume that $\mathbf{v}_\ell^{min} = (c_{i_1}, c_{i_2}, \dots, c_{i_n})$ and $\mathbf{v}_\ell^{max} = (c_{i'_1}, c_{i'_2}, \dots, c_{i'_n})$. From the definition of $\mathbf{v}_\ell^{min}$ and $\mathbf{v}_\ell^{max}$, we have $i_t \leq i'_t$ for all $t \in [n]$.

Observe that the event structures of the history buffers in $f_{\mathsf{nsw+2nsw}}(\mathbf{v}_\ell^{min})$ and $f_{\mathsf{nsw+2nsw}}(\mathbf{v}_\ell^{max})$ are the same. This means that along the computation $\sigma := f_{\mathsf{nsw+2nsw}}(\mathbf{v}_\ell^{min}) \to_{\mathcal{N}}^* f_{\mathsf{nsw+2nsw}}(\mathbf{v}_\ell^{max})$ no memory update operations have been performed. This implies that the order between the sequences of operations performed by each process along σ is not relevant. Hence, what we need to prove is that for every $t \in [n]$, we have a computation σ_t under NSW from the configuration $f_{\mathsf{nsw+2nsw}}(c_{i_1}, c_{i_2}, \dots, c_{i_t}, \dots, c_{i_n})$ to the configuration $f_{\mathsf{nsw+2nsw}}(c_{i_1}, c_{i_2}, \dots, c_{i'_t}, \dots, c_{i_n})$ where only the process $\mathcal{P}_t$ is active. Then, using the computations $\sigma_1, \dots, \sigma_n$, we can construct the computation σ as follows: First, we start by executing the sequence of operations of the process $\mathcal{P}_1$ performed along σ_1, then a sequence of of operations of the process $\mathcal{P}_2$ performed along σ_2, and so on ....

Now, for every $t \in [n]$, we know that there is a computation $\rho_t := c_{i_t} \leadsto_{\mathcal{N}}^* c_{i'_t}$ under NSW^+. Then, we can construct a computation σ_t of $\mathcal{N}$ under NSW from the configuration $f_{\mathsf{nsw+2nsw}}(c_{i_1}, c_{i_2}, \dots, c_{i_t}, \dots, c_{i_n})$ to the configuration $f_{\mathsf{nsw+2nsw}}(c_{i_1}, c_{i_2}, \dots, c_{i'_t}, \dots, c_{i_n})$ such that the sequence of operations performed by the process $\mathcal{P}_t$ along σ_t is the same as the sequence of operations performed by $\mathcal{P}_t$ along ρ_t. (Recall that the process $\mathcal{P}_t$ is the only active process along σ_t.) First, let us assume that $i_t = i'_t$, then we have $f_{\mathsf{nsw+2nsw}}(c_{i_1}, c_{i_2}, \dots, c_{i_t}, \dots, c_{i_n}) = f_{\mathsf{nsw+2nsw}}(c_{i_1}, c_{i_2}, \dots, c_{i'_t}, \dots, c_{i_n})$, and so $f_{\mathsf{nsw+2nsw}}(c_{i_1}, c_{i_2}, \dots, c_{i_t}, \dots, c_{i_n}) \to_{\mathcal{N}}^* f_{\mathsf{nsw+2nsw}}(c_{i_1}, c_{i_2}, \dots, c_{i'_t}, \dots, c_{i_n})$. Now, if $i_t < i'_t$, we can use the following Lemma 5 to prove that $f_{\mathsf{nsw+2nsw}}(c_{i_1}, c_{i_2}, \dots, c_{i_t}, \dots, c_{i_n}) \to_{\mathcal{N}}^* f_{\mathsf{nsw+2nsw}}(c_{i_1}, c_{i_2}, \dots, c_{i'_t}, \dots, c_{i_n})$.

Lemma 5. *For every $j \in \{i_t, \dots, i'_t - 1\}$, one of the following cases hold:*

– $f_{\mathsf{nsw+2nsw}}(c_{i_1}, c_{i_2}, \dots, c_j, \dots, c_{i_n}) \to_{\mathcal{N}} f_{\mathsf{nsw+2nsw}}(c_{i_1}, c_{i_2}, \dots, c_{j+1}, \dots, c_{i_n})$, *or*
– $f_{\mathsf{nsw+2nsw}}(c_{i_1}, c_{i_2}, \dots, c_j, \dots, c_{i_n}) = f_{\mathsf{nsw+2nsw}}(c_{i_1}, c_{i_2}, \dots, c_{j+1}, \dots, c_{i_n})$.

Proof. First observe that the NSW-configurations $\gamma = f_{\mathsf{nsw+2nsw}}(c_{i_1}, c_{i_2}, \dots, c_j, \dots, c_{i_n})$ and $\gamma' = f_{\mathsf{nsw+2nsw}}(c_{i_1}, c_{i_2}, \dots, c_{j+1}, \dots, c_{i_n})$ are well-defined. In fact, by definition, we have that the event structure of the history buffer associated with the process $\mathcal{P}_t$ in the configuration c_j and c_{j+1} is the same the event structure of the history buffer associated with the process $\mathcal{P}_t$ in the configurations c_{i_t} and $c_{i'_t}$.

Observe that the process $\mathcal{P}_t$ has in the configurations γ and c_j (resp. γ' and c_{j+1}): (1) the same control state, (2) the same content of the write buffers $(WB_{(i,j)})_{i\in[n]}^{j\in[m]}$, (3) the same pointed elements in the history buffer, and (4) the projection of $WB_{(i,m+1)}$ under NSW^+ over the write operations performed by $\mathcal{P}_t$ is exactly the same as the content of $WB_{(i,m+1)}$ under NSW. Hence, if $\mathcal{P}_t$ can perform an operation (other than an update operation) from c_j to c_{j+1}, then $\mathcal{P}_i$ can perform the same operation from γ to γ'.

Otherwise, the local configuration of the process $\mathcal{P}_t$ in c_j and c_{j+1} remains unchanged, and hence $\gamma = \gamma'$.

Other than the event structure of the history buffer of $\mathcal{P}_t$ remains the same along the computation from c_j to c_{j+1}, we exclude update operations of the process $\mathcal{P}_t$ under NSW^+ since this can corresponds to a write operation performed by other processes (recall that the serialization buffer $WB_{(t,m+1)}$ contain sequence of memory states due to write operations of all the processes). $\square$

Using the arguments above and Lemma 5, we conclude that for every $\ell \in \{L_{min}, \ldots, L_{min} - 1\}$, we have $f_{\mathsf{nsw+2nsw}}(\mathbf{v}_\ell^{min}) \rightarrow_{\mathcal{N}}^* f_{\mathsf{nsw+2nsw}}(\mathbf{v}_\ell^{max})$. $\square$

Our second lemma to prove is the following:

Lemma 6. *For every* $\ell \in \{L_{min}, \ldots, L_{max} - 1\}$*,* $f_{\mathsf{nsw+2nsw}}(\mathbf{v}_\ell^{max}) \rightarrow_{\mathcal{N}} f_{\mathsf{nsw+2nsw}}(\mathbf{v}_{\ell+1}^{min})$.

Proof. Let us assume that $\mathbf{v}_{\ell+1}^{min} = (c_{i_1}, c_{i_2}, \ldots, c_{i_n})$ and $\mathbf{v}_\ell^{max} = (c_{i'_1}, c_{i'_2}, \ldots, c_{i'_n})$. From the definition of $\mathbf{v}_{\ell+1}^{min}$ and $\mathbf{v}_\ell^{max}$, we have $i'_t < i_t$ for all $t \in [n]$ (since the size of the history buffer associated to $\mathcal{P}_t$ in the configuration c_{i_t} is stricly greater than the size of the history buffer associated to $\mathcal{P}_t$ in the configuration c_{i_t}).

For every $t \in [n]$, we can show (by contradiction) that we have the following computation $c_{i'_t} \rightsquigarrow_{\mathcal{N}} c_{i_t}$ which is due to an update (or an atomic read-write) operation performed by the process $\mathcal{P}_t$. This is an immediate consequence of the definition of $c_{i'_t}$ and c_{i_t}. Let us assume that $c_{i'_t} \rightsquigarrow_{\mathcal{N}} c_{i_t}$ is due to an update operation. The case of an atomic read-write operation is treated in a similar way.

Then, we have, for every $t \in [n]$, all these update operations correspond to a write operation ω issued by the same process (say $\mathcal{P}_{t'}$ with $t' \in [n]$) since all processes under NSW^+ have guessed the same order in which write operations will be update to the main memory (see the semantics under NSW^+ of a $\mathsf{Transfer\ write}$ operation).

Now, we can show that the write operation ω is in the head of the serialization buffer associated to the process $\mathcal{P}_{t'}$ in the configuration $f_{\mathsf{nsw+2nsw}}(\mathbf{v}_\ell^{max})$ and an update operation can be performed from this configuration under NSW to reach the configuration $f_{\mathsf{nsw+2nsw}}(\mathbf{v}_{\ell+1}^{min})$. Observe that for every $t \in [n] \setminus \{t'\}$, the process $\mathcal{P}_t$ has, in the configurations $f_{\mathsf{nsw+2nsw}}(\mathbf{v}_\ell^{max})$ and $f_{\mathsf{nsw+2nsw}}(\mathbf{v}_{\ell+1}^{min})$, the same control state, the same content of write buffers $(WB_{(t,j)})_{j\in[m+1]}$, and the same pointed events in the history buffer. $\square$

Now from Lemma 4 and Lemma 6, we obtain that there is a computation under NSW from the configuration $f_{\mathsf{nsw+2nsw}}(\mathbf{v}_{L_{min}}^{min})$ to the configuration $f_{\mathsf{nsw+2nsw}}(\mathbf{v}_{L_{max}}^{max})$. since $\mathbf{v}_{L_{min}}^{min} = (c_0, \ldots, c_0)$ and $\mathbf{v}_{L_{max}}^{max} = (c_k, \ldots, c_k)$, this implies that $f_{\mathsf{nsw+2nsw}}(\mathbf{v}_{L_{min}}^{min}) = (\mathbf{p}, \overline{\mathcal{S}_\emptyset}, \mathcal{H})$ and $f_{\mathsf{nsw+2nsw}}(\mathbf{v}_{L_{max}}^{max}) = (\mathbf{p}', \overline{\mathcal{S}_\emptyset}, \mathcal{H}')$. Hence $\langle \mathbf{p}, \overline{\mathcal{S}_\emptyset}, \mathcal{H} \rangle \rightarrow_{\mathcal{N}}^* \langle \mathbf{p}', \overline{\mathcal{S}_\emptyset}, \mathcal{H}' \rangle$.

The Only if direction: Assume that there is a computation ρ of $\mathcal{N}$ under NSW. This computation provides an order $\mathcal{O}$ in which the write operations are updated to the main memory (i.e., to the history buffer). This order will determine the moment at which the write operations under NSW^+ will be transferred from the write buffers $(WB_{(i,j)})_{i\in[n]}^{j\in[m]}$ to the serialization ones $(WB_{(i,m+1)})_{i\in[n]}$. Now, we can construct a computation ρ' of $\mathcal{N}$

under NSW^+ such that the following invariant is preserved: After each action performed by a process $\mathcal{P}_i$ under NSW and NSW$^+$, $\mathcal{P}_i$ will have: (1) the same control state, (2) the same content of $(WB_{(i,j)})_{i\in[n]}^{j\in[m]}$, (3) the same pointed elements in the history buffer, and (4) the projection of $WB_{(i,m+1)}$ under NSW^+ over the write operations performed by $\mathcal{P}_i$ is exactly the same as the content of $WB_{(i,m+1)}$ under NSW. Then, if $\mathcal{P}_i$ can perform an operation under NSW, then $\mathcal{P}_i$ can perform the same operation under NSW^+ while preserving the above invariant. The transfer of a write operation from a store buffer $WB_{(i,j)}$ to serialization buffers under NSW^+ can be only performed if it respects the order imposed by $\mathcal{O}$. Moving a pointer under NSW can be simulated by moving this pointer under NSW^+.

D Proof of Lemma 2

First, we can show that any operation $t \in \Delta_i$ performed by a process $\mathcal{P}_i$ from c_1 can be performed by $\mathcal{P}_i$ from c'_1 and we reach a configuration c'_2 larger or equal (wrt. to $\preceq$) than c_2. This is an immediate consequence of the definition of $\preceq$ since $\mathcal{P}_i$ has the same control state in c_1 and c'_1, the same last pending write operations per process (since they are encoded as strong symbols), the last event in the history buffers, and the pointed events in the history buffers.

Let us assume that the system $\mathcal{N}$ performs a *Transfer write* operation of a write operation $\mathsf{w}(i,j,d)$ (labeling an event element e) from c_1 and reaches c_2. Now, from the configuration c'_1, the system $\mathcal{N}$ can also perform several *Transfer write* operations (from the write buffer $WB_{(i,j)}$) until a write operation $\mathsf{w}(i,j,d)$ labeling an event element e' matching e (wrt. to one of the injection functions defining the ordering) is transferred. Then, we can easily prove that the reached configuration c'_2 after this sequence of *Transfer write* operations is larger or equal to c_2 (wrt. to $\preceq$).

Finally, the case of a *Memory update* or *Move pointer* operation is similar to *Transfer write* operation. A *Memory update* (resp. *Move pointer*) operation from c_1 can be simulated by a sequence of *Memory update* (resp. *Move pointer*) operations from c'_1.

E Proof of Theorem 9

The proof is by a reduction of PCP (Post's Correspondence Problem) to our problem. Let $\{u_1,\ldots,u_n\}$ and $\{v_1,\ldots,v_n\}$ be an instance of PCP. We construct a system $\mathcal{N}$ with two processes $\mathcal{P}_1$ and $\mathcal{P}_2$ sharing a set of four variables $X = \{x_1,x_2,x_3,x_4\}$ such that, two specific states in $\mathcal{N}$ are related by a run iff PCP has a solution for the considered instance. The idea of the reduction is as follows:

Process $\mathcal{P}_1$ guesses the solution of PCP as a sequence of indices $i_1,\ldots,i_k$ and performs iteratively a sequence of operations: It (1) writes successively to x_1 the symbols of u_{i_j}, (2) reads from x_3 the symbols of u_{i_j}, (3) writes to x_2 the index i_j, and (4) reads i_j from x_4, for j ranging backward from k to 1. Moreover, each write (resp. read) operation to (resp. from) a variable is followed by a write (resp. read) operation of the marker $\sharp$. The insertion of the markers allows to ensure that a written value to a variable by one of the processes can be read at most once by the other processes. In parallel, process $\mathcal{P}_2 also guesses the solution of PCP and performs the same operations as $\mathcal{P}_1$, except that it writes (resp. reads) symbols of the words v_{i_j} and the indices i_j to x_3 and x_4 (from x_1 and x_2), respectively.

Then, we prove that PCP has a solution if and only if it is possible to reach a state of the system $\mathcal{N}$ where the event structure is empty. In other words, a full computation of $\mathcal{N}$ checks that the two processes have guessed the same sequence of indices and that this sequence is indeed a solution for the considered PCP instance. The "only if direction" can be shown using the fact that the read operations of the indices i_j to x_2 and x_4 of processes $\mathcal{P}_1$ and $\mathcal{P}_2$ can be immediately validated using the RRWE. This means that when $\mathcal{P}_1$ writes u_{i_j} to x_1 followed by a read of u_{i_j} from x_3, $\mathcal{P}_2$ writes v_{i_j} to x_3 followed by a read of v_{i_j} from x_1. Then, when $\mathcal{P}_1$writes i_j to x_2 followed by a read of i_j from x_4, $\mathcal{P}_2$ writes i_j to x_4 followed by a read of v_{i_j} from x_2. Now, the read operations of the indices i_j to x_2 and x_4 of processes $\mathcal{P}_1$ and $\mathcal{P}_2$ can be immediately validated using the RRWE. Thus, the event structure of $\mathcal{N}$ will only contain only event associated to (1) write operations of the process $\mathcal{P}_1$ to the variables x_1 and x_2, and of the process $\mathcal{P}_2$ to the variables x_3 and x_4, and (2) read operations of the process $\mathcal{P}_1$ to the variable x_3, and of the process $\mathcal{P}_2$ to the variable x_1. (Notice that the write operations on the variables x_2 and x_4 do not play any role in the remaining part of the computation since they can be overtaken by any write/read operations on the variables x_1 and x_3.) Then, it is possible to construct a run of the $\mathcal{N}$ where the execution of each write done by one of the process $\mathcal{P}_1$ (resp. $\mathcal{P}_2$) on the variable x_1 (resp. x_3) is immediately followed by its corresponding read operation done by $\mathcal{P}_2$ (resp. $\mathcal{P}_1$) on x_2 (resp. x_1).

The argument for the reverse direction is the following: If there is a run which empties the event structure, then it can be seen that, due to the fact that a read can validate at most one write, the sequence of read symbols by process $\mathcal{P}_2$ is a subword of the sequence of written symbols by $\mathcal{P}_1$, and vice versa. The same holds also for the sequences of indices guessed by both processes. These facts imply that the processes have indeed guessed the same (right) solution to the given instance of PCP.

Let us define more formally the reduction. Let $D = \Sigma \cup \{\sharp, -\} \cup [n]$ be the set of data manipulated by processes $\mathcal{P}_1$ and $\mathcal{P}_2$.

To simplify the presentation, we need to introduce some notations. Let $i \in [2]$, $j \in [4]$, $s \in D^*$, $\mathsf{op} \in \{\mathsf{w}, \mathsf{r}\}$, $m = \mathit{length}(s)$ and such that $m \geq 2$. We use the macro transition $p \xrightarrow{\mathsf{op}(i,j,s)}_i p'$ to denote the sequence of consecutive transitions $p \xrightarrow{\mathsf{op}(i,j,s(1))}_i p_1$, $p_l \xrightarrow{\mathsf{op}(i,j,s(l+1))}_i p_{l+1}$ for all $l \in [m-2]$, and $p_{m-1} \xrightarrow{\mathsf{op}(i,j,s(m))}_i p'$ where $p_1, \ldots, p_m$ are extra intermediary control states of $\mathcal{P}_i$ that are not used anywhere else (and that we may omit from the set of control states of $\mathcal{P}_i$). We use also $op(i,j,s)$ to denote the fact that the event structure contains the following sequence of ordered operations $op(i,j,s(m)) \rightsquigarrow \cdots \rightsquigarrow op(i,j,s(1))$.

Let ν be a mapping from Σ^* to D^* such that for every word $u = a_1 \cdots a_m \in \Sigma^*$, $\nu(u) = \sharp \cdot a_1 \cdots \sharp \cdot a_m$.

Then, a computation of the process $\mathcal{P}_1$ (resp. $\mathcal{P}_2$) is a sequence of phases where each phase consists in the following operations:

1. Choose a number $l \in [n]$:
   $p \xrightarrow{\mathsf{nop}}_1 p_l$ (resp. $q \xrightarrow{\mathsf{nop}}_2 q_l$)

2. Write the sequence of data $\nu(u_l)$ (resp. $\nu(v_l)$) to x_1 (resp. x_3):
   $p_l \xrightarrow{\mathsf{w}(1,1,\nu(u_l))}_1 p_l^1$ (resp. $q_l \xrightarrow{\mathsf{w}(2,3,\nu(v_l))}_2 q_l^1$)
3. Read the sequence of data $\nu(u_l)$ (resp. $\nu(v_l)$) from x_3 (resp. x_1):
   $p_l^1 \xrightarrow{\mathsf{r}(1,3,\nu(u_l))}_1 p_l^2$ (resp. $q_l^1 \xrightarrow{\mathsf{r}(2,1,\nu(v_l))}_2 q_l^2$)
4. Write the sequence of data $\sharp \cdot l$ to x_2 (resp. x_4):
   $p_l^2 \xrightarrow{\mathsf{w}(1,2,\sharp \cdot l)}_1 p_l^3$ (resp. $q_l^2 \xrightarrow{\mathsf{w}(2,4,\sharp \cdot l)}_2 q_l^3$)
5. Read the sequence of data $\sharp \cdot l$ from x_4 (resp. x_2):
   $p_l^3 \xrightarrow{\mathsf{r}(1,4,\sharp \cdot l)}_1 p$ (resp. $q_l^3 \xrightarrow{\mathsf{r}(2,2,\sharp \cdot l)}_2 q$)

Next, we establish the link between the state reachability problem for the NSW $\cup$ $\{$RRWE$\}$ memory system $\mathcal{N}$ and the existence of a solution for the PCP.

Lemma 7. *There is $i_1,\ldots,i_k \in [n]$ such that $u_{i_1}\cdots u_{i_k} = v_{i_1}\cdots v_{i_k}$ if and only if the configuration $\langle (p,q),(\sharp,\sharp,\sharp,\sharp),\mathcal{S}_\emptyset\rangle$ is reachable in $\mathcal{N}$ from the initial configuration $\langle (p,q),(-,-,-,-),\mathcal{S}_\emptyset\rangle$.*

Proof. (The if direction:) Assume that $\langle (p,q),(\sharp,\sharp,\sharp,\sharp),\mathcal{S}_\emptyset\rangle$ is reachable in $\mathcal{N}$ from $\langle (p,q),(-,-,-,-),\mathcal{S}_\emptyset\rangle$. This means that all the read operations of $\mathcal{P}_1$ and $\mathcal{P}_2$ have been validated.

Then, assume that $i_k,\ldots,i_1$ is the sequence of indices chosen by $\mathcal{P}_1$ and that $j_h,\ldots,j_1$ is the sequence of indices chosen by $\mathcal{P}_2$. We use the facts that (1) write and read operations by a same process to a same variable cannot be reordered, and that (2) each write operation of $\mathcal{P}_1$ can only validate a unique read operation of $\mathcal{P}_2$ and vice-versa (but of course some written values can be missed since processes are asynchronous), to show that the following relations hold:

– $u_{i_1}u_{i_2}\cdots u_{i_k} \preceq v_{j_1}v_{j_2}\cdots v_{j_h}$.
– $v_{j_1}v_{j_2}\cdots v_{j_h} \preceq u_{i_1}u_{i_2}\cdots u_{i_k}$.
– $i_1 i_2\cdots i_k \preceq j_1 j_2\cdots j_h$.
– $j_1 j_2\cdots j_h \preceq i_1 i_2\cdots i_k$.

This implies that $u_{i_1}u_{i_2}\cdots u_{i_k} = v_{j_1}v_{j_2}\cdots v_{j_h}$ and $i_1 i_2\cdots i_k = j_1 j_2\cdots j_h$.

(The only-if direction:) Assume that there is a sequence of indices $i_1,\ldots,i_k \in [n]$ such that $u_{i_1}\cdots u_{i_k} = v_{i_1}\cdots v_{i_k}$. Then, we can construct the following run of $\mathcal{N}$ from the initial configuration $\langle (p,q),(-,-,-,-),\mathcal{S}_\emptyset\rangle$ to the configuration $\langle (p,q),(\sharp,\sharp,\sharp,\sharp),\mathcal{S}_\emptyset\rangle$:

For every l from k to 1, we have

1. First, $\mathcal{P}_1$ chooses the index i_l and stores in its event the sequence of operations $\mathsf{w}(1,2,\cdot i_l)\mathsf{r}(1,3,\nu(u_{i_l}))\mathsf{w}(1,1,\nu(u_{i_l}))$.
2. Then, $\mathcal{P}_2$ chooses the index i_l and stores in its event structure the sequence of operations $\mathsf{w}(2,4,\cdot i_l)\mathsf{r}(2,1,\nu(v_{i_l}))\mathsf{w}(2,3,\nu(v_{i_l}))$.
3. $\mathcal{P}_1$ can use the RRWE to validate the following read operation $\mathsf{r}(1,4,i_l)$ with the last write operation of the process $\mathcal{P}_2$ on x_4.
4. $\mathcal{P}_2$ can use the RRWE to validate the following read operation $\mathsf{r}(2,2,i_l)$ with the last write operation of the process $\mathcal{P}_1$ on x_2.
5. $\mathcal{P}_1$ stores in its event structure the write operation $\mathsf{w}(1,2,\sharp)$.

6. $\mathcal{P}_2$ stores in its event structure the write operation $\mathsf{w}(2,4,\sharp)$.
7. $\mathcal{P}_1$ can use the RRWE to validate the following read operation $\mathsf{r}(1,4,\sharp)$ with the last write operation of the process $\mathcal{P}_2$ on x_4.
8. $\mathcal{P}_2$ can use the RRWE to validate the following read operation $\mathsf{r}(2,2,\sharp)$ with the last write operation of the process $\mathcal{P}_1$ on x_2.

Finally, $\mathcal{N}$ adopts the following run where the execution of each write done by one of the process $\mathcal{P}_1$ (resp. $\mathcal{P}_2$) on the variable x_1 (resp. x_3) is immediately followed by its corresponding read operation done by the process $\mathcal{P}_2$ (resp. $\mathcal{P}_1$) on the variable x_2 (resp. x_1).