What's Decidable about Weak Memory Models?

Mohamed Faouzi Atig[1], Ahmed Bouajjani[2],
Sebastian Burckhardt[3], and Madanlal Musuvathi[3]

[1] Uppsala University, Sweden, mohamed_faouzi.atig@it.uu.se
[2] LIAFA, Paris Diderot Univ. & CNRS, France, 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 weak 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, 26, 4], testing tools [11, 18], analyses that check whether programs are exposed to specific relaxations [7, 9, 20], fence insertion tools [14, 15, 17], verified compilation [10, 24, 23], and formal models that closely approximate commercial multiprocessors [21, 22, 25].

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 verification problem is decidable? Understanding the answers to these questions is crucial for model checking safety properties of programs under a relaxed memory model and for checking if a program exhibits the same behavior under different 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.

$\mathsf{r} \rightarrow \mathsf{r/w}$ (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. 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.

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

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, 19]. On the other hand, a memory model that avoids this relaxation but otherwise remains 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 leads to undecidability. Such non-atomic writes can lead to counter-intuitive IRIW (independent reads of independent writes) effects [6].

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 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 3.6.
- 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 subtle 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 $length(w)$. (We assume that $length(\varepsilon) = 0$.) For every $i \in [length(w)]$, 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., $\exists i \in [length(w)]$ 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.

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 $d, d' \in D$. Intuitively, $\mathsf{r}(i, j, d)$ (resp. $\mathsf{w}(i, j, d)$) means that process i reads (resp. writes) the data d from (resp. to) the variable x_j. The semantics of atomic read-writes and of read/write_fences will be explained in section 3.2.

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 operations issued by its different processes. In the Sequential Consistency (SC) model, the 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 (i.e., barriers) can be used to impose the serialization of some operations at some execution points. 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 full fence, i.e., a fence such that any two operations by the same process occurring before and after (in program order) the full fence cannot be swapped. The operation $\mathsf{wfence}(i)$ (resp. $\mathsf{rfence}(i)$) is a fence for writes (resp. reads) 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 **op₁ to op₂** 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 $i \in [n]$, $j,k \in [m]$, $j \neq k$, 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 define in this framework the two well known models TSO and PSO as follows:

$$\begin{aligned} \mathsf{TSO} &= \mathsf{RWF} \cup \mathsf{RLWE} \cup \{\mathsf{w} \rightarrow \mathsf{r}\} \\ \mathsf{PSO} &= \mathsf{RWF} \cup \mathsf{RLWE} \cup \{\mathsf{w} \rightarrow \mathsf{r},\, \mathsf{w} \rightarrow \mathsf{w}\} \end{aligned}$$

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

Given a process $\mathcal{P}_i$ of $\mathcal{N}$, and two control states $p, p' \in P_i$, 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 are $p_0 \cdots p_\ell \in P_i^*$ 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 computation 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 vectors 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

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

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 $Reach_{\mathcal{N}}^{\mathsf{M}}$ 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 $Reach_{\mathcal{N}}^{\mathsf{M}}(s, s')$ 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 states s and s' of $\mathcal{N}$, checking whether $Reach_{\mathcal{N}}^{\mathsf{M}}(s, s')$ holds. We have:

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

We 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 in [3] is incomparable with TSO (since write_fences are necessary to simulate TSO under that model), and is strictly less expressive (w.r.t. the set of computation traces) than the PSO 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, using a reduction of Post's Correspondence Problem, the following stronger result:

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

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 Absence of Causality Cycles in NSW

Let po denote the *program order* relation corresponding to the order in which operations of each thread are issued by the program. Then, one can define a dependency relation between operations of a same process that reflects the data and control dependencies. We adopt here a conservative definition by considering that all operations occurring after a read operation, in the program order, are dependent from that 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 define a *read-from* relation, denoted rf, that associates with each read event of the computation a write event such that $\mathsf{w}(i,k,d) \to_{\mathsf{rf}} \mathsf{r}(j,k,d)$ 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 $\cup \{\mathsf{r} \to \mathsf{w}\}$, there are programs having computations with a cyclic causality relation. An example of such a program is given on the right. It is clear that under the SC model, the four operations of this program cannot belong to a same computation from $x = y = 0$ to $x = y = 1$. However, using the $\mathsf{r} \to \mathsf{w}$ relaxation, it is possible by permuting (1) and (2), to execute the four operations in the following order $(2), (3), (4), (1)$. This computation contains the causality cycle: $(2) \to_{\mathsf{rf}} (3) \to_{\mathsf{dep}} (4) \to_{\mathsf{rf}} (1) \to_{\mathsf{dep}} (2)$. We prove that by discarding the $\mathsf{r} \to \mathsf{w}$ relaxation, NSW avoids causal cycles.

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

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

Notice that since this theorem relies on the conservative definition of dependency given 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 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. ARW: $p \xrightarrow{\mathsf{arw}(i,j,d,d')}_i p'$, $\bigcup_{\ell=1}^{m} \widehat{E}_{(i,\ell)} = \emptyset$, $\mathbf{d}[j] = d$, $\mathbf{d}' = \mathbf{d}[j \leftarrow d']$, 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 $Reach_{\mathcal{N}}^{\mathsf{NSW}}(s,s')$ 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 provide 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 wfences 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 from a write operation $\mathsf{w}(k,j,d)$ issued by process $\mathcal{P}_k$. 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 $\mathcal{P}_k$ can't 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 state of the memory.

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 $\mathcal{P}_i$.

Notice that the relevant part of the history buffer at any moment is formed by the elements between the last element (current state of the memory) and the oldest element that is pointed by π.

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 $add(\mathcal{B}, a)$ 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 $\rightsquigarrow' = (\rightsquigarrow \cup \{(max(E), e)\})^*$, and (3) $\lambda' = \lambda \cup [e \mapsto a]$. Then, if $\lambda(min(B)) = a$, let $remove(\mathcal{B}, a)$ be the buffer $(E', \rightsquigarrow', \lambda')$ such that (1) $E' = E \setminus \{min(E)\}$, (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 $tail(\mathcal{B})$ (resp. $head(\mathcal{B})$) the element $\lambda(max(E))$ (resp. $\lambda(min(E))$).

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., $\pi(i,j) = max(E)$ 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 $\mathcal{S}' = ((\mathcal{S} \lhd [e \leftarrow \mathsf{w}(i,j,d)]) \oplus \{e' \rightsquigarrow e : e' \in max(\widehat{E}_{(i,j)})\}$.
3. Write fence: $p \xrightarrow{\mathsf{wfence}(i)}_i p'$, $\mathcal{H} = \mathcal{H}'$, and $\exists e \in \mathcal{E} \setminus E$ such that $\mathcal{S}' = ((\mathcal{S} \lhd [e \leftarrow \mathsf{wfence}(i)]) \oplus \{e' \rightsquigarrow e : \exists k.\ 1 \leq k \leq m \text{ and } e' \in max(\widehat{E}_{(i,k)})\})$.
4. RLWE: $p \xrightarrow{\mathsf{r}(i,j,d)}_i p'$, $\mathcal{S} = \mathcal{S}'$, $\mathcal{H} = \mathcal{H}'$, $WE(i,j) \neq \emptyset$, and $data(max(WE(i,j))) = d$.
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: $p \xrightarrow{\mathsf{r}(i,j,d)}_i p'$, $\mathcal{S} = \mathcal{S}'$, $\mathcal{H} = \mathcal{H}'$, $WE(i,j) = \emptyset$, 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. ARW: $p \xrightarrow{\mathsf{arw}(i,j,d,d')}_i p'$, $\mathcal{S} = \mathcal{S}'$, $\bigcup_{\ell=1}^{m} \widehat{E}_{(i,\ell)} = \emptyset$, $\pi(i,\ell) = max(H)$ 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, $(s, \mathcal{S}_0) \Rightarrow^*_{\mathcal{N}} (s', \mathcal{S}_0)$ if and only if $\langle \mathbf{p}, \mathcal{S}_0, \mathcal{H} \rangle \Rrightarrow^*_{\mathcal{N}} \langle \mathbf{p}', \mathcal{S}_0, \mathcal{H}' \rangle$.*

5.2 Eliminating write fences from event structures

We show in this section that we can 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 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 memory.

Then, 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 allow 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 $WB_{(i,j)} = (B_{(i,j)}, \rightsquigarrow_{(i,j)}, \lambda_{(i,j)})$ and $WB'_{(i,j)} = (B'_{(i,j)}, \rightsquigarrow'_{(i,j)}, \lambda'_{(i,j)})$ for all i and j, and for every two history buffers $\mathcal{H} = (\mathcal{B}, \pi)$ and $\mathcal{H}' = (\mathcal{B}', \pi')$, where $\mathcal{B} = (H, \rightsquigarrow_H, \lambda_H)$ and $\mathcal{B}' = (H', \rightsquigarrow_{H'}, \lambda_{H'})$ are two buffers over M, we have $\langle \mathbf{p}, (WB_{(i,j)})_{i\in[n]}^{j\in[m+1]}, \mathcal{H} \rangle \rightarrow_{\mathcal{N}} \langle \mathbf{p}', (WB'_{(i,j)})_{i\in[n]}^{j\in[m+1]}, \mathcal{H}' \rangle$ 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 $\exists j \in [m]$. $\exists e \in H$. $\pi(i,j) \rightsquigarrow_H e$ and $\pi' = \pi[(i,j) \leftarrow e]$.
9. ARW: $p \xrightarrow{\mathsf{arw}(i,j,d,d')}_i p'$, $Empty(WB_{(i,j)})$ 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 establishes 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

[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 $e_i \preceq e_j$.

configurations w.r.t. $\preceq$ can always simulate smaller ones, 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 one 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).

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, losing 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 lose 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, losing 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 lose 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, losing $\mathsf{w}(i,k,d)$ 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, losing $(x_k = d)$ does not correspond to losing 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 process 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 $\big(\mathcal{H}_i =$

$(\mathcal{B}_i, \pi_i))_{i\in[n]}$ and $(\mathcal{H}'_i = (\mathcal{B}'_i, \pi'_i))_{i\in[n]}$, 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 $WB'_{(i,j)} = add(WB_{(i,j)}, \mathsf{w}(i,j,d))$.
3. Write fence: $p \xrightarrow{\mathsf{wfence}(i)}_i p'$, $Empty(WB_{(i,j)})$ for all $j \in [m]$, $WB_{(k,\ell)} = WB'_{(k,\ell)}$ 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 $\exists \omega = head(WB_{(i,j)})$. $WB'_{(i,j)} = remove(WB_{(i,j)}, \omega)$ and for every $k \in [n]$, $WB'_{(k,m+1)} = add(WB_{(k,m+1)}, \mathsf{w}(i,j,\mathbf{d}'))$ 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. ARW: $p \xrightarrow{\mathsf{arw}(i,j,d,d')}_i p'$, $WB_{(k,\ell)} = WB'_{(k,\ell)}$ 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 $\pi'_i = \pi_i$.

We prove 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, $\langle \mathbf{p}, \overline{\mathcal{S}_0}, \mathcal{H} \rangle \rightarrow^*_{\mathcal{N}} \langle \mathbf{p}', \overline{\mathcal{S}_0}, \mathcal{H}' \rangle$ iff $\langle \mathbf{p}, \overline{\mathcal{S}'_0}, \mathcal{H}, \ldots, \mathcal{H} \rangle \mapsto^*_{\mathcal{N}} \langle \mathbf{p}', \overline{\mathcal{S}'_0}, \mathcal{H}', \ldots, \mathcal{H}' \rangle$ where $\overline{\mathcal{S}_0}$ and $\overline{\mathcal{S}'_0}$ denotes an $[n] \times [m+1]$-dim vector of empty buffers.*

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 $c \preceq c'$ 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 8 (WQO). *The relation $\preceq$ is a WQO on the set of NSW$^+$-configurations of $\mathcal{N}$.*

Then, we can prove the following important fact:

Lemma 9 (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 $c_2 \preceq c'_2$.*

From [1], we know that in order to show the decidability of the state reachability problem for NSW$^+$, we only need to show that:

Lemma 10 (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.*

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

Theorem 11. *The state reachability problem for* NSW$^+$ *is decidable.*

As a corollary of Theorem 7 and Theorem 11, we obtain our main result:

Corollary 12. *The state reachability problem for* NSW *is decidable.*

7 Nonatomic Writes Cause Undecidability

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

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.

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 $\mathcal{P}_i$ from a variable x_j to be validated by the last write issued by $\mathcal{P}_k$ (with $k \neq i$) on x_j, although this write has not been yet committed, it becomes possible.

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$(x = 0, y = 0)$ 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 reads (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, (2) and (5) can be executed, and then, the reads (3) and (6) can be validated w.r.t. the content of the main memory. Finally, the writes corresponding to e_1 and e_2 in $\mathcal{S}$ are committed to the main memory, and this yields the memory state $(x = 1, y = 1)$.

We can prove by a reduction of Post's Correspondance Problem the following fact:

Theorem 13. *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 further sharpening the boundary by considering finer distinctions of the $\mathsf{r} \to \mathsf{w}$ 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. Abdulla, P.A., Cerans, K., Jonsson, B., Tsay, Y.K.: General decidability theorems for infinite-state systems. In: LICS. pp. 313–321 (1996)
2. Adve, S., Gharachorloo, K.: Shared memory consistency models: a tutorial. Computer 29(12), 66–76 (1996)

3. Atig, M.F., Bouajjani, A., Burckhardt, S., Musuvathi, M.: On the verification problem for weak memory models. In: POPL. pp. 7–18. ACM (2010)
4. Atig, M.F., Bouajjani, A., Parlato, G.: Getting rid of store-buffers in TSO analysis. In: CAV (2011)
5. Boehm, H.: WG21/N2176 memory model rationales. http://open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2176.html#dependencies (March 2007)
6. Boehm, H., Adve, S.: Foundations of the C++ concurrency memory model. In: PLDI. pp. 68–78 (2008)
7. Bouajjani, A., Meyer, R., Möhlmann, E.: Deciding robustness against total store ordering. In: ICALP (2011)
8. Burckhardt, S., Alur, R., Martin, M.: CheckFence: Checking consistency of concurrent data types on relaxed memory models. In: PLDI. pp. 12–21 (2007)
9. Burckhardt, S., Musuvathi, M.: Effective program verification for relaxed memory models. In: Computer-Aided Verification (CAV). pp. 107–120 (2008 Extended Version as Tech Report MSR-TR-2008-12, Microsoft Research)
10. Burckhardt, S., Musuvathi, M., Singh, V.: Verifying local transformations on relaxed memory models. In: CC'10. pp. 104–123 (2010)
11. Burnim, J., Sen, K., Stergiou, C.: Testing concurrent programs on relaxed memory models. Tech. Rep. UCB/EECS-2010-32, EECS Department, University of California, Berkeley (Mar 2010), http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-32.html
12. Chen, C., Chen, W., Sreedhar, V., Barik, R., Sarkar, V., Gao, G.: Establishing causality as a desideratum for memory models and transformations of parallel programs. Tech. rep., University of Delaware (2010)
13. Gharachorloo, K., Gupta, A., Hennessy, J.: Performance evaluation of memory consistency models for shared-memory multiprocessors. In: ASPLOS'91. pp. 245–257 (1991)
14. Kuperstein, M., Vechev, M., Yahav, E.: Automatic inference of memory fences. In: FMCAD. pp. 111–119 (Oct 2010)
15. Kuperstein, M., Vechev, M., Yahav, E.: Partial-coherence abstractions for relaxed memory models. In: PLDI. San Jose, CA (Jun 2011)
16. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Comp. C-28(9), 690–691 (1979)
17. Linden, A., Wolper, P.: A verification-based approach to memory fence insertion in relaxed memory systems. In: SPIN (2011)
18. Mador-Haim, S., Alur, R., Martin, M.: Generating litmus tests for contrasting memory consistency models. In: Computer Aided Verification. pp. 273–287 (2010)
19. Manson, J., Pugh, W., Adve, S.: The java memory model. In: POPL. pp. pages = 378–391, 378–391 (2005)
20. Owens, S.: Reasoning about the implementation of concurrency abstractions on x86-tso. In: ECOOP, LNCS, vol. 6183, pp. 478–503. Springer (2010)
21. Owens, S., Sarkar, S., Sewell, P.: A better x86 memory model: x86-TSO. In: TPHOL (2009)
22. Sarkar, S., Sewell, P., Alglave, J., Maranget, L., Williams, D.: Understanding POWER multiprocessors. In: PLDI. San Jose, CA (Jun 2011)
23. Sevcik, J.: Safe optimisations for shared-memory concurrent programs. In: PLDI. pp. 306–316 (2011)
24. Sevcik, J., Vafeiadis, V., Nardelli, F.Z., Jagannathan, S., Sewell, P.: Relaxed-memory concurrency and verified compilation. In: POPL. pp. 43–54 (2011)
25. Sewell, P., Sarkar, S., Owens, S., Nardelli, F., Myreen, M.: x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. Commun. ACM 53 (2010)
26. Yang, Y., Gopalakrishnan, G., Lindstrom, G.: UMM: an operational memory model specification framework with integrated model checking capability. Concurrency and Computation: Practice and Experience 17(5-6), 465–487 (2005)