HAL
archives-ouvertes.fr

# Fences and Synchronisation Idioms in Weak Memory Models

Jade Alglave, Luc Maranget

## To cite this version:

Jade Alglave, Luc Maranget. Fences and Synchronisation Idioms in Weak Memory Models. [Research Report] RR-7152, INRIA. 2009. inria-00440863

**HAL Id: inria-00440863**

**https://hal.inria.fr/inria-00440863**

Submitted on 12 Dec 2009

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

# *Fences and Synchronisation in Weak Memory Models*

Jade Alglave — Luc Maranget

**N° 7152**

Décembre 2009

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--7152--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
PARIS - ROCQUENCOURT

# Fences and Synchronisation in Weak Memory Models

Jade Alglave , Luc Maranget

Thème :
Équipe-Projet Moscova

Rapport de recherche n° 7152 — Décembre 2009 — 21 pages

**Abstract:** We present an axiomatic framework, implemented in the Coq proof assistant, to define weak memory models in terms of several parameters: local reorderings of reads and writes, and visibility of inter and intra processor communications through memory, including full store atomicity relaxation. Thereby, we give a formal hierarchy of weak memory models, in which we provide a formal study of what should be the action and placement of fences to restore a given model such as *SC* from a weaker one. Finally, we provide formal requirements for abstract locks that guarantee *SC* semantics to data race free programs, and show that a particular implementation of locks matches these requirements.

**Key-words:** Weak Memory Models, Fences, Locks

Centre de recherche INRIA Paris – Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex
Téléphone : +33 1 39 63 55 11 — Télécopie : +33 1 39 63 53 30

# Fences and Synchronisation in Weak Memory Models

**Résumé :**

**Mots-clés :**

# 1 Introduction

Understanding the behaviour of a program running on a multiprocessor requires a precise definition of the underlying memory system and the behaviour of the processors involved—that is, the *memory model*. Previous studies [8] have discussed the need for rigorous definitions of weak memory models, which some of the public documentations [12, 17] lack. We provide here a generic framework, implemented in the Coq proof assistant [6], to precisely define a memory model in terms of several parameters expressing potential sources of weakness.

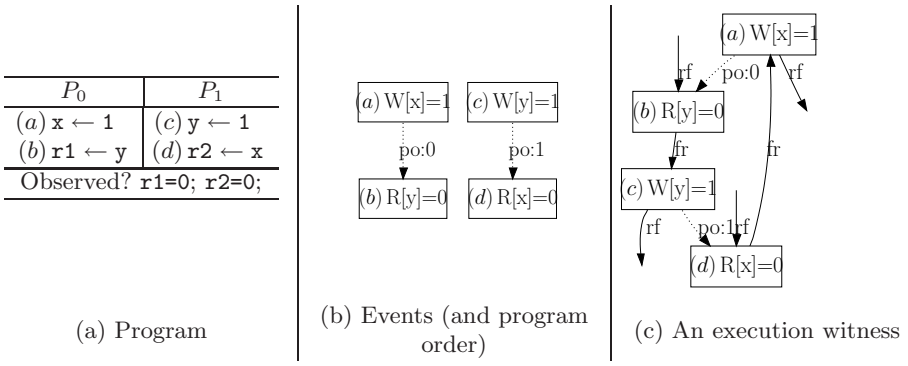
### Weak Memory Models

*Processor behaviour* A simple model of a processor's behaviour in a multiprocessor context could assume a sequential order, consistent with the program order, of all the read and write events issued by this processor, as a generalisation of the uniprocessor case. However, modern architectures [20, 17, 3] provide *relaxed memory models* that do not constrain the way reads and writes are ordered as much. These constraints, or their relaxation, are often referred to as *instruction reordering* [2, 5].

*Representation of memory* A simple model of a shared memory could assume a single memory on which several processors operate simultaneously, with all their writes being commited to memory as soon as they are issued. Thus, the connection between processors and memory could be considered as direct: as soon as a processor writes to memory, the value written overwrites the previous value and is immediately available to all processors. This property, called *store atomicity*, has been advocated as valuable [2, 5], as it provides the guarantee that actions on such a memory are serialisable, which leads to a rather understandable memory model. However, it is not guaranteed on certain architectures [13, 17], which relax the store atomicity constraint. This means a write is not available to all processors at once: a write could be *e.g.* at first initiated by a given processor, then commited to a cache, and finally to memory. This last step is called *globally performed* [10]. Even without assuming writes to be commited immediately, we suppose a total order on the globally performed writes to the same location, a property called *coherence* [17] that is widely assumed by modern architectures [3, 17, 20, 12].

### Contribution

*An axiomatic generic model* We define an *architecture* in terms of its ordering and store atomicity relaxations in Sec. 2.3, whose validity conditions are described in Sec. 2.4. We illustrate how to instantiate our model to produce Sequential Consistency (henceforth *SC*) [14] and Sparc *TSO* [20], and show equivalence with the native models, together with characterisation of executions that would be valid on these models in Sec. 2.6.




(a) Program (b) Events (and program order) (c) An execution witness

**Fig. 1.** A program and a candidate execution

*Study of barriers power* Most architectures provide mechanisms such as *barriers*—or *fences*—to restore *SC* from a weaker model. However, it is not clear how much power a barrier needs in order to do so, and where to place these constructions in the code. We examine these questions in Sec. 3.1 from a general point of view: we provide a sufficient condition on barriers to restore a stronger model from a weaker one. We refine this condition in some interesting cases in Sec. 3.2.

*Atomicity* Finally in Sec. 4.2, we ensure *SC* semantics to *data-race free* (henceforth *drf*) programs provided specified requirements on lock and unlock primitives, and show that a particular implementation of these primitives, involving *reservations* such as in Alpha [3] and Power [17], meets these requirements.

Our results and proofs are formalised in the Coq proof assistant; we omit the detail of proofs due to lack of space. However, the development and the sketches of the proofs are available at `http://moscova.inria.fr/~alglave/wmm/`.

## 2 Description of the model

Fig. 1(a) shows a program written in pseudo code and a potential outcome. We will write `x`, `y` for memory locations, and `r1`, `r2` for registers. If each location holds initially 0, this outcome may occur for example on an x86 machine since the write-read pair on each processor may be reordered [12, Sec. 2.3]. Such reorderings preclude the use of an interleaving semantics to reason on executions induced by weak memory models. Moreover, these reorderings affect the events generated by instructions, rather than the instructions themselves.

### 2.1 Basic objects

Thus, the model deals with various *events* occurring in an abstract execution of a multiprocessor program. We write $\mathbb{E}$ for the set of all events generated during a given execution.

A *memory event* $m$ represents a memory access, specified by its direction (write or read), its location $\mathrm{loc}(m)$, its value $\mathrm{val}(m)$, its processor $\mathrm{proc}(m)$, and a unique identifier. For example, consider the store to `x` with value 1 labelled $(a)$ in Fig. 1(a): it generates the homonymous event in Fig. 1(b). Henceforth, we write $r$ (resp. $w$) for an arbitrary read (resp. write) event. We write $\mathbb{M}$ (resp. $\mathbb{R}$, $\mathbb{W}$) for the set of memory events (resp. reads, writes).

A *barrier event* $b$ is generated by each barrier (or fence) instruction; we write $\mathbb{B}$ for the set of all such events in a given execution.

An execution is also characterised by the *program order* $\xrightarrow{\mathrm{po}}$, a total order amongst the events from the same processor[1] that never relates events from different processors. It reflects the sequential execution of instructions on a single processor: given two instruction instances $i_1$ and $i_2$ that generate events $e_1$ and $e_2$, $e_1 \xrightarrow{\mathrm{po}} e_2$ means that a sequential processor would execute $i_1$ before $i_2$.

### 2.2 Execution witnesses

Although it conveys important features of program execution, such as branch resolution, $\xrightarrow{\mathrm{po}}$ is not sufficient to characterise an execution. Indeed, we need to examine where a read value originates from, and express memory coherence. Therefore, we postulate two relations over events: $\xrightarrow{\mathrm{rf}}$ and $\xrightarrow{\mathrm{ws}}$.

*Read-from map (rf)* $w \xrightarrow{\mathrm{rf}} r$ means that $r$ loads the value stored by $w$; it implies that $w$ and $r$ share the same location and value. Moreover, given a read $r$ there exists a unique write $w$ such that $w \xrightarrow{\mathrm{rf}} r$ ($w$ can be an *init* store when $r$ loads from initial state). Formally, $\xrightarrow{\mathrm{rf}}$ must be well formed as follows, where $\mathbb{W}_{\ell,v}$ and $\mathbb{R}_{\ell,v}$ are the sets of writes and reads with location $\ell$ and value $v$:

$$\text{wf-rf}(\xrightarrow{\mathrm{rf}}) \triangleq \xrightarrow{\mathrm{rf}} \subseteq \bigcup_{\ell,v} (\mathbb{W}_{\ell,v} \times \mathbb{R}_{\ell,v}) \wedge \forall r,\ \exists! w.\ w \xrightarrow{\mathrm{rf}} r$$

*Write serialisation (ws)* In a coherent memory, all values written to a given location $\ell$ are serialised, following a *coherence order*. We define $\xrightarrow{\mathrm{ws}}$ as the union of the coherence orders for all memory locations, which must be well formed as follows, where $\mathbb{W}_\ell$ is the set of writes to location $\ell$:

$$\text{wf-ws}(\xrightarrow{\mathrm{ws}}) \triangleq \xrightarrow{\mathrm{ws}} \subseteq \bigcup_{\ell} (\mathbb{W}_\ell \times \mathbb{W}_\ell) \wedge \forall \ell.\ \text{total-order}\left(\xrightarrow{\mathrm{ws}}, (\mathbb{W}_\ell \times \mathbb{W}_\ell)\right)$$

*From-read map (fr)* We call $\xrightarrow{\mathrm{fr}}$ the following relation, gathers all pairs of reads $r$ and writes $w$ such that $r$ reads from a write that is before $w$ in $\xrightarrow{\mathrm{ws}}$:

$$r \xrightarrow{\mathrm{fr}} w \triangleq \exists\ w', w' \xrightarrow{\mathrm{rf}} r \wedge w' \xrightarrow{\mathrm{ws}} w$$

[1] When instructions may perform several memory accesses, $\xrightarrow{\mathrm{po}}$ becomes a partial order, thus should include some intra-instruction dependencies [18] to build a total order.

We gather the information that describes an execution into an *execution witness*, depicted by $X$:

$$X \triangleq (\mathbb{E}, \xrightarrow{\text{po}}, \xrightarrow{\text{rf}}, \xrightarrow{\text{ws}})$$

Fig. 1(c) shows an execution witness for the test of Fig. 1(a). The load $d$ reads the initial value of `x`, later overwritten by the store $a$. Thus, we have $d \xrightarrow{\text{fr}} a$, as a consequence of the serialisation of the init store to `x` (which comes first in $\xrightarrow{\text{ws}}$) and the write $a$. We define the well formedness predicate wf on execution witnesses, as the conjunction of the associated predicates for $\xrightarrow{\text{ws}}$ and $\xrightarrow{\text{rf}}$.

### 2.3 Execution order

We define the order in which memory events are globally performed—the *global happens-before* relation, depicted by $\xrightarrow{\text{ghb}}$—as a partial order over all such events of a candidate execution witness, following Alpha [3] or Sparc [20]. $m_1 \xrightarrow{\text{ghb}} m_2$ means that $m_1$ occurs before $m_2$ *w.r.t.* all processors. $\xrightarrow{\text{ghb}}$ is subject to various conditions—according to architectural parameters—which we now describe.

**Globality of relations** In our setting, writes are not necessarily atomic [2, 5] *i.e.* not necessarily available to all parts of the memory system—or globally performed [10]—at once. This determines which relations are to be considered global, *i.e.* included in $\xrightarrow{\text{ghb}}$.

First, $\xrightarrow{\text{rf}}$ is not necessarily included in $\xrightarrow{\text{ghb}}$. We distinguish the constraints induced by internal $\xrightarrow{\text{rf}}$ ($\xrightarrow{\text{rf}}$ relation on a same processor), which we note $\xrightarrow{\text{rfi}}$, and external ($\xrightarrow{\text{rf}}$ from one processor to another), which we note $\xrightarrow{\text{rfe}}$:

$$w \xrightarrow{\text{rfi}} r \triangleq w \xrightarrow{\text{rf}} r \wedge \text{proc}(w) = \text{proc}(r)$$

$$w \xrightarrow{\text{rfe}} r \triangleq w \xrightarrow{\text{rf}} r \wedge \text{proc}(w) \neq \text{proc}(r)$$

A memory model could allow *store forwarding*—or *read own's writes early* [2]—meaning the processor that issued a given write can read its value before any other participant has access to it: in that case, $\xrightarrow{\text{rfi}}$ is not included in $\xrightarrow{\text{ghb}}$. A model could also allow two particular processors that share a cache to read a write issued by their neighbour—*w.r.t.* the cache hierarchy—before any other participant that does not share the same cache—a particular case of *read others' writes early* [2]. In that case, $\xrightarrow{\text{rfe}}$ is not considered global.

Second, $\xrightarrow{\text{ws}}$ and $\xrightarrow{\text{fr}}$ are always global. Indeed, the coherence order for a given location is the order in which writes to this location are globally performed. Moreover, as $r \xrightarrow{\text{fr}} w$ expresses that the write $w'$ from which $r$ reads is globally performed before $w$, it forces the read $r$ to be globally performed before all participants agree on the value stored by $w$; otherwise $r$ could read its value from $w'$ in a given processor view, and from $w$ in another processor view.

**Preserved program order** Some models ensure that certain pairs of events are to be maintained in program order *w.r.t.* all participants: for example, *TSO* [20] ensures this property for all write-write pairs. To represent these constraints, we postulate a global relation $\xrightarrow{\text{ppo}}$ which gathers all pairs of events that are not to be reordered with respect to the program order $\xrightarrow{\text{po}}$.

Consider for instance the execution witness depicted in Fig. 1(c): it is valid only if the writes and reads to different locations appearing on each processor have been reordered. Indeed, if these pairs are maintained in program order—*i.e.* $a \xrightarrow{\text{ppo}} b$ and $c \xrightarrow{\text{ppo}} d$—we have a cycle in $\xrightarrow{\text{ghb}}$, since $\xrightarrow{\text{ppo}}$ and $\xrightarrow{\text{fr}}$ are included in $\xrightarrow{\text{ghb}}$: $a \xrightarrow{\text{ppo}} b \xrightarrow{\text{fr}} c \xrightarrow{\text{ppo}} d \xrightarrow{\text{fr}} a$. Therefore, an architecture that authorises the specified outcome cannot include write-read pairs to different locations in its $\xrightarrow{\text{ppo}}$.

**Barriers constraints** Architectures provide particular instructions—*barriers* or *fences* such as Power `sync`—to enforce a certain order between pairs of events, according to a particular semantics, such as the ones we define in Sec. 3. We postulate a global relation $\xrightarrow{\text{ab}}$ that gathers all such pairs.

**Architecture** An architecture, depicted by $A$, collects this information—where *int* and *ext* are booleans that determine whether $\xrightarrow{\text{rfi}}$ and $\xrightarrow{\text{rfe}}$ are included in $\xrightarrow{\text{ghb}}$, and *ppo* and *ab* are functions that, given an execution witness, output the eponymous relations:

$$A \triangleq (ppo, int, ext, ab)$$

We define $\xrightarrow{\text{ghb}}$ as the union of the global relations, with $\xrightarrow{\text{?rf}} \triangleq \xrightarrow{\text{?rfi}} \cup \xrightarrow{\text{?rfe}}$, where $\xrightarrow{\text{?rfe}}$ (resp. $\xrightarrow{\text{?rfi}}$) is $\xrightarrow{\text{rfe}}$ (resp. $\xrightarrow{\text{rfi}}$) if *ext* (resp *int*) is *true*, the empty relation otherwise:

$$\xrightarrow{\text{ghb}} \triangleq \xrightarrow{\text{ppo}} \cup \xrightarrow{\text{ws}} \cup \xrightarrow{\text{fr}} \cup \xrightarrow{\text{?rf}} \cup \xrightarrow{\text{ab}}$$

We write $A.\text{ghb}$ for the function that, given an execution witness, outputs the appropriate $\xrightarrow{\text{ghb}}$ relation, *w.r.t.* $A$. We write $A^{\epsilon}$ for an architecture $A$ where we consider *ab* to be the function that always outputs the empty relation.

### 2.4 Validity of an execution with respect to an architecture

We define the validity of an execution as the conjunction of several criterions on the various relations we defined.

**Uniprocessor behaviour** As stated by [17, p.29], one expects a sole processor to respect the *sequential execution model*, which is:

> *[...] the model of program execution in which the processor appears to execute one instruction at a time, completing each instruction before beginning to execute the next instruction.*

| $P_0$ | $P_1$ |
|---|---|
| $(a)$ `x` $\leftarrow$ `1` | $(b)$ `r1` $\leftarrow$ `x`<br>$(c)$ `x` $\leftarrow$ `2` |
| Forbidden: `x=1`; `r1=1`; | |

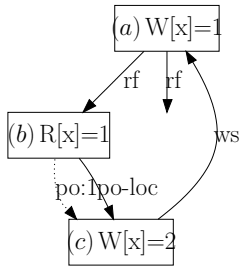



**Fig. 2.** Invalid execution by uniproc

We understand this as a constraint on a sole processor in a multiprocessor context: a sole processor cannot ignore memory coherence. For instance, a processor that reads a location $\ell$ twice cannot load values that would contradict write serialisation; or if a processor writes $v$ to $\ell$ and then reads $v'$ from $\ell$, one expects $v'$ not to precede $v$ in coherence order. We view the sequential execution model as preventing local reordering of memory accesses to the same location and enforcing memory coherence for each processor. We first define the relation $\overset{\text{po-loc}}{\rightarrow}$ which gathers all pairs of accesses to the same location in the program order:

$$m_1 \overset{\text{po-loc}}{\rightarrow} m_2 \triangleq m_1 \overset{\text{po}}{\rightarrow} m_2 \wedge \text{loc}(m_1) = \text{loc}(m_2)$$

To guarantee our executions respect the sequential execution model, we could enforce $\overset{\text{po-loc}}{\rightarrow} \subseteq \overset{\text{ghb}}{\rightarrow}$; but this would enforce $\overset{\text{rfi}}{\rightarrow} \subseteq \overset{\text{ghb}}{\rightarrow}$ (as $\overset{\text{rfi}}{\rightarrow} \subseteq \overset{\text{po-loc}}{\rightarrow}$) whereas we wish to remain free to consider $\overset{\text{rfi}}{\rightarrow}$ non global. Instead, we require $\overset{\text{po-loc}}{\rightarrow}$, $\overset{\text{rf}}{\rightarrow}$, $\overset{\text{ws}}{\rightarrow}$ and $\overset{\text{fr}}{\rightarrow}$ to be compatible; we define ($\overset{\text{hb-seq}}{\rightarrow}$ being the union of $\overset{\text{rf}}{\rightarrow}$, $\overset{\text{ws}}{\rightarrow}$ and $\overset{\text{fr}}{\rightarrow}$):

$$\text{uniproc}(X) \triangleq \text{acyclic}\ (\overset{\text{hb-seq}}{\rightarrow} \cup \overset{\text{po-loc}}{\rightarrow})$$

Fig. 2 illustrates this condition. The outcome reveals the complete execution witness: we have $c \overset{\text{ws}}{\rightarrow} a$ (by `x` final value) and $a \overset{\text{rf}}{\rightarrow} c$ (by `r1` final value). The cycle $a \overset{\text{rf}}{\rightarrow} b \overset{\text{po-loc}}{\rightarrow} c \overset{\text{ws}}{\rightarrow} a$ invalidates this execution: the read $b$ cannot read from the write $a$ as it is a future value of `x` in $\overset{\text{ws}}{\rightarrow}$.

We define the validity of an execution with respect to an architecture $A$ as the conjunction of these conditions:

$$A.\,\text{valid}(X) \triangleq \text{wf}(X) \wedge \text{uniproc}(X) \wedge \text{acyclic}(A.\text{ghb}(X))$$

### 2.5 Properties of validity

From this definition arises a very simple notion of comparison defined by a predicate among architectures: $A_1 \leq A_2$ means that $A_1$ is *weaker* than $A_2$. Below, $f_1 \subseteq f_2$ stands for: $\forall X, f_1(X) \subseteq f_2(X)$, if $f_1$ and $f_2$ are functions that

output relations over events when given an execution witness, and $\Rightarrow$ is the implication over booleans:

$$A_1 \leq A_2 \triangleq ppo_1 \subseteq ppo_2 \wedge int_1 \Rightarrow int_2 \wedge ext_1 \Rightarrow ext_2$$

**Validity is decreasing** We prove validity of an execution to be decreasing *w.r.t.* the weaker predicate; thus, a weaker architecture may exhibit at least all the behaviours authorised by a stronger one:

**Theorem 1 (Validity is decreasing).**

$$\forall A_1 A_2,\ (A_1 \leq A_2) \Rightarrow (\forall X, A_2^{\epsilon} \,.\, \mathrm{valid}(X) \Rightarrow A_1^{\epsilon} \,.\, \mathrm{valid}(X))$$

**Monotonicity of validity** Some programs running on an architecture $A_1$ could exhibit particular executions that would be valid on a stronger architecture $A_2$; we characterise all such by the following criterion:

$$A_1 . \mathrm{check}_{\mathrm{A}_2}(X) \triangleq \mathrm{acyclic}(\overset{?\mathrm{rf}_2}{\rightarrow} \cup \overset{\mathrm{ws}}{\rightarrow} \cup \overset{\mathrm{fr}}{\rightarrow} \cup \overset{\mathrm{ppo}_2}{\rightarrow})$$

**Theorem 2 (Characterisation).**

$$\forall A_1 A_2,\ (A_1 \leq A_2) \Rightarrow (\forall X, (A_1^{\epsilon} \,.\, \mathrm{valid}(X) \wedge A_1 \,.\, \mathrm{check}_{\mathrm{A}_2}(X)) \Leftrightarrow A_2^{\epsilon} \,.\, \mathrm{valid}(X))$$

### 2.6 Examples

We examine here two classical models, *SC* [14] and Sparc *TSO* [20]. In both cases we propose alternative formulations in our framework, which we proved equivalent to the original definitions. We omit the detail of formalism related to these equivalences due to lack of space. We first define a few convenient notations to extract pairs of memory events from the program order:

$$MM = (\mathbb{M} \times \mathbb{M}) \cap \overset{\mathrm{po}}{\rightarrow} \quad RM \triangleq (\mathbb{R} \times \mathbb{M}) \cap \overset{\mathrm{po}}{\rightarrow} \quad WW \triangleq (\mathbb{W} \times \mathbb{W}) \cap \overset{\mathrm{po}}{\rightarrow}$$

***SC*** is a model in which no reordering of events is allowed ($\overset{\mathrm{ppo}}{\rightarrow}$ equals $\overset{\mathrm{po}}{\rightarrow}$ on memory events) and writes are available to all processors as soon as they are issued ($\overset{\mathrm{rf}}{\rightarrow}$ are global). Thus, there is no need for barriers:

$$Sc \,.\, Arch \triangleq (\lambda X.MM, true, true, \lambda X.\emptyset)$$

Note that any architecture definable in our framework is weaker than *Sc*. Thm. 2 shows the following criterion characterises, on any architecture $A$, valid weak executions that are *Sc*:

$$A \,.\, \mathrm{check}_{Sc}(X) \triangleq \mathrm{acyclic}(\overset{\text{hb-seq}}{\rightarrow} \cup \overset{\mathrm{po}}{\rightarrow})$$

Thus, the outcome of Fig. 1 will never be the result of an *Sc* execution, as it exhibits the cycle: $a \overset{\mathrm{po}}{\rightarrow} b \overset{\mathrm{fr}}{\rightarrow} c \overset{\mathrm{po}}{\rightarrow} d \overset{\mathrm{fr}}{\rightarrow} a$.

***TSO*** is described in [2] as allowing two relaxations: *write to read program order* and *read own's write early*. We interpret the first relaxation as a $\stackrel{\text{ppo}}{\rightarrow}$ relation that orders load-load, load-store and store-store pairs: $\stackrel{\text{ppo-tso}}{\rightarrow} \triangleq RM \cup WW$. We interpret the second relaxation as internal $\stackrel{\text{rf}}{\rightarrow}$ not being global. And indeed the execution model of Sparc architectures is provided by the *Value* axiom of [20], which states that a read ($L_a$ for Sparc) reads from the most recent write ($S_a$) that is before it in the global ordering relation ($\leq$) or in the program order (;):

$$Val(L_a) = Val(\max_{\leq}\{S_a \mid S_a \leq L_a \vee S_a; L_a\})$$

Thus, we propose an alternative definition of Sparc $TSO$[2]:

$$Tso^{\epsilon} \,.\, Arch \triangleq (\lambda X. \stackrel{\text{ppo-tso}}{\rightarrow}, \mathit{false}, \mathit{true}, \lambda X.\emptyset)$$

Thm. 2 shows the following criterion, where $\stackrel{\text{hb-tso}}{\rightarrow}$ is $\stackrel{\text{ws}}{\rightarrow} \cup \stackrel{\text{fr}}{\rightarrow} \cup \stackrel{\text{rfe}}{\rightarrow}$, characterises as valid (*w.r.t.* any $A \leq Tso$) executions that would be valid on $Tso^{\epsilon}$:

$$A \,.\, \text{check}_{Tso}(X) \triangleq \text{acyclic}(\stackrel{\text{hb-tso}}{\rightarrow} \cup \stackrel{\text{ppo-tso}}{\rightarrow})$$

Thus, we can conclude that the outcome of Fig. 1 may show up on a *Tso* machine, as it does not exhibit any cycle in $\stackrel{\text{hb-tso}}{\rightarrow} \cup \stackrel{\text{ppo-tso}}{\rightarrow}$.

## 3 Semantics of barriers

A program may exhibit on a weaker architecture executions that would not be valid on a stronger one. However, one may want to ensure validity on any memory model—typically *SC*—of a given program. Architectures provide special instructions, namely *barriers*, which enforce some ordering constraints in a program when present in the program order between two instructions. We consider here the question of restoring a stronger model from a weaker one by using barriers: we examine both their placement and the power they should have to do so.

### 3.1 Barriers guarantee

Consider two architectures $A_1 \leq A_2$. We define the predicate fb— for *fully barriered*—on $A_1$ as follows, where $\stackrel{\text{r}_2\setminus 1}{\rightarrow} \triangleq \stackrel{\text{r}_2}{\rightarrow} \setminus \stackrel{\text{r}_1}{\rightarrow}$:

$$A_1 \,.\, \text{fb}_{A_2}(X) \triangleq \left((\stackrel{\text{ppo}_2\setminus 1}{\rightarrow}) \cup (\stackrel{?\text{rf}_2\setminus 1}{\rightarrow}; \stackrel{\text{ppo}_2}{\rightarrow})\right) \subseteq \stackrel{\text{ab}_1}{\rightarrow}$$

We prove that the above condition on $\stackrel{\text{ab}_1}{\rightarrow}$ suffices to restore $A_2^{\epsilon}$ from $A_1$:

**Theorem 3 (Barriers guarantee).**

$$\forall A_1 A_2, (A_1 \leq A_2) \Rightarrow (\forall X, A_1 \,.\, \text{valid}(X) \wedge A_1 \,.\, \text{fb}_{A_2}(X) \Rightarrow A_2^{\epsilon} \,.\, \text{valid}(X))$$

[2] We elide barrier semantics, which we study in detail in Sec. 3.

This theorem provides an insight on the power the barriers provided by an architecture $A_1$ should have to restore a stronger one $A_2$. They should:

1. restore the pairs that are preserved in the program order on $A_2$ and not on $A_1$, which is a static property;
2. compensate for the fact that some writes may not be globally performed at once on $A_1$ while they are on $A_2$, which we model by (some subrelation of) $\xrightarrow{\text{rf}}$ not being global on $A_1$ while it is on $A_2$; this is a dynamic property.

**Static property of barriers** is expressed by the condition $\xrightarrow{\text{ppo}_{2\setminus 1}} \subseteq \xrightarrow{\text{ab}_1}$: a barrier provided by $A_1$ should enforce events generated by a same processor to be globally performed in program order if they are on $A_2$. In this case, it suffices to insert a barrier between the instructions that generate these events.

**Dynamic property of barriers** is expressed by the condition $\xrightarrow{?\text{rf}_{2\setminus 1}};\xrightarrow{\text{ppo}_2}$: a barrier provided by $A_1$ should force store atomicity for the write events that have this property on $A_2$. This is how we interpret the *cumulativity* of barriers as stated by Power [17]. We interpret furthermore the *A-cumulativity* (resp. *B-cumulativity*) property as applying to barriers that enforce ordering of pairs in $\xrightarrow{\text{rf}};\xrightarrow{\text{po}}$ (resp. $\xrightarrow{\text{po}};\xrightarrow{\text{rf}}$). We consider a barrier that only preserves pairs in $\xrightarrow{\text{po}}$ to be *non cumulative*. Thm. 3 states it suffices to insert an A-cumulative barrier between each pair of instructions such that the first one in the program order reads from a write which is to be globally performed on $A_2$ but is not on $A_1$.

Note that it would suffice to have $w \xrightarrow{\text{ab}_1} r$ whenever $w \xrightarrow{?\text{rf}_{2\setminus 1}} r$ holds to restore store atomicity. However, such an achievement would probably be extremely costly: it supposes that when the barrier event occurs (after $r$), it waits until $w$ is globally performed, and then reads again to ensure that $r$ is globally performed after $w$. Our condition does not enforce such a costly policy: given the situation $w \xrightarrow{\text{rf}} r \xrightarrow{\text{po}} m$, where $r$ is not necessarily globally performed after $w$ is, inserting a barrier between $r$ and $m$—more precisely inserting a barrier instruction between the instructions generating $r$ and $m$—only forces the processor that generates $r$ and $m$ to delay $m$ until $w$ is globally performed.

**Restoring *Sc*** We model an A-cumulative barrier as the following function on execution witnesses that returns an ordering relation when given a placement of barriers in the code[3]:

$$m_1 \xrightarrow{\text{fenced}} m_2 \triangleq \quad \exists b, m_1 \xrightarrow{\text{po}} b \xrightarrow{\text{po}} m_2$$
$$\text{A-cumul}(X, \xrightarrow{\text{fenced}}) \triangleq \quad \xrightarrow{\text{fenced}} \cup \xrightarrow{\text{rf}};\xrightarrow{\text{fenced}}$$

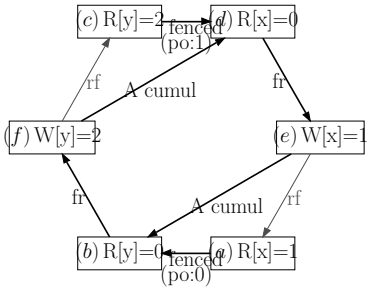
[3] One may need to distinguish between $\xrightarrow{\text{fenced-base}}$ and $\xrightarrow{\text{fenced-cumul}}$ to express a barrier that would cumulate on some pairs, while preserving the local order of some others.

**iriw**

| $P_0$ | $P_1$ | $P_2$ | $P_3$ |
|---|---|---|---|
| $(a)$ `r1 ← x` | $(c)$ `r2 ← y` | $(e)$ `x ← 1` | $(f)$ `y ← 2` |
| `fence` | `fence` | | |
| $(b)$ `r2 ← y` | $(d)$ `r1 ← x` | | |

Observed? `0:r1=1; 0:r2=0; 1:r2=2; 1:r1=0;`

(a) Program




(b) Placement of A-cumulative barriers

**Fig. 3.** Study of **iriw**

Assuming such a barrier—*i.e.* the A-cumul relation included in $\xrightarrow{\text{ab}}$— fencing all $\xrightarrow{\text{po}}$ pairs suffices to restore *Sc* by Thm. 3:

**Corollary 1 (Barriers restoring *Sc*).**

$$\forall A\ X, (A\,.\,\text{valid}(X) \wedge \text{A-cumul}(X, MM) \subseteq \xrightarrow{ab}) \Rightarrow Sc\,.\,\text{valid}(X)$$

Consider the example given in Fig. 3(a): the specified outcome may arise in the absence of barriers on a weak architecture, as the result of a non-*Sc* execution. The barrier whose semantics we designed above forbids this outcome, as shown in Fig. 3(b): when placed in between each pair of reads on the first two processors, it not only prevents their reordering, but also forces the writes from the last two processors to be globally performed before the second component of each pair of reads.

Thus, one ensures an *Sc* behaviour for all programs by fencing *all* pairs in $\xrightarrow{\text{po}}$. However, this would much impair performance; to restore *Sc* from *A*, it indeed suffices to invalidate non *Sc* executions, by fencing only the $\xrightarrow{\text{po}}$ pairs appearing in the $\xrightarrow{\text{hb-seq}} \cup \xrightarrow{\text{po}}$ cycles of these executions. The static analysis of [19] may be of considerable help, as it is based on compile-time (safe) approximation of cycles in $\xrightarrow{\text{hb-seq}} \cup \xrightarrow{\text{po}}$ that may arise at runtime. We believe this technique would apply to architectures with store atomicity relaxation, provided their barriers offer A-cumulativity.

**Restoring *Tso*$^\epsilon$** We design a semantics for a barrier that would restore *Tso*$^\epsilon$ from any weaker architecture *A*:

$$\text{ext-A-cumul}(X, \xrightarrow{\text{fenced}}) \triangleq \xrightarrow{\text{fenced}} \cup \xrightarrow{\text{rfe}}; \xrightarrow{\text{fenced}}$$

As internal $\xrightarrow{\text{rf}}$ are not considered global in *Tso*$^\epsilon$, there is no need to compensate them: the cumulativity power of this barrier applies only to external $\xrightarrow{\text{rf}}$. Furthermore, all pairs preserved by the program order in *Tso*$^\epsilon$ are to be preserved,

as depicted by the placement condition $\overset{\text{fenced}}{\rightarrow} = \overset{\text{ppo-tso}}{\rightarrow}$ in the following corollary of Thm. 3:

**Corollary 2 (Barriers restoring $Tso^{\epsilon}$).**

$$\forall A,\ A \leq Tso \Rightarrow \forall X,\ (A\,.\,\text{valid}(X) \wedge \text{ext-A-cumul}(X, \overset{ppo\text{-}tso}{\rightarrow}) \subseteq \overset{ab}{\rightarrow}) \Rightarrow Tso^{\epsilon}\,.\,\text{valid}(X)$$

### 3.2 Considering a weaker guarantee

Consider the particular case of two architectures $A_2$ and $A_1$ with the same policy *w.r.t.* the store atomicity and store buffer relaxations, which we model by $ext_1 = ext_2$ and $int_1 = int_2$. In this case, there is no need for a barrier as powerful as above to restore $A_2$ from $A_1$: a barrier that only orders the events that surround it statically—that is, a non cumulative barrier, which action we model by non-cumul$(X, \overset{\text{fenced}}{\rightarrow}) \triangleq \overset{\text{fenced}}{\rightarrow}$—is enough. Consider the wfb predicate:

$$A_1\,.\,\text{wfb}_{A_2}(X) \triangleq \overset{\text{ppo}_2\setminus 1}{\rightarrow} \subseteq \overset{\text{ab}_1}{\rightarrow}$$

This states that the barriers provided by $A_1$ maintain the pairs that are preserved in the program order on $A_2$ but not on $A_1$. Moreover, this guarantee applies if $A_2$ hinders the store buffer relaxation by its preserved program order, *i.e.* when $\overset{\text{rfi}}{\rightarrow} \subseteq \overset{\text{ppo}_2}{\rightarrow}$—which is particular to $Sc$—as stated in the following:

**Theorem 4 (Non cumulative barriers guarantee).**

$$\forall A_1 A_2, ((A_1 \leq A_2) \wedge (ext_1 = ext_2) \wedge ((int_1 = int_2) \vee (\overset{rfi}{\rightarrow} \subseteq \overset{ppo_2}{\rightarrow}))) \Rightarrow$$
$$(\forall X, A_1\,.\,\text{valid}(X) \wedge A_1\,.\,\text{wfb}_{A_2}(X) \Rightarrow A_2^{\epsilon}\,.\,\text{valid}(X))$$

**From *Tso* to *Sc*** As $\overset{\text{rfe}}{\rightarrow}$ are considered global in both $Tso$ and $Sc$, and $Sc$ hinders the store buffering relaxation by its $\overset{\text{ppo}}{\rightarrow}$ definition, Thm. 4 applies. Hence to restore $Sc$ from $Tso$, it suffices to fence all pairs in $\overset{\text{ppo-sc}}{\rightarrow} \setminus \overset{\text{ppo-tso}}{\rightarrow} = WR$, where $WR \triangleq (\mathbb{W} \times \mathbb{R}) \cap \overset{\text{po}}{\rightarrow}$, as expressed by the following corollary of Thm. 4:

**Corollary 3 (Barriers restoring $Sc$ from $Tso$).**

$$\forall X, (Tso\,.\,\text{valid}(X) \wedge \text{non-cumul}(X, WR) \subseteq \overset{ab_{Tso}}{\rightarrow}) \Rightarrow Sc\,.\,\text{valid}(X))$$

**From *Pso* to *Tso*$^{\epsilon}$** We comment here on the two definitions of $PSO$ given in Sparc documentations [20]. We first adapt the definition of [20, V8] to our framework:

$$Pso^{\epsilon}\,.\,Arch \triangleq (\lambda X.RM, false, true, \lambda X.\emptyset)$$

As for $Tso$, we deduce from the Value axiom that external $\overset{\text{rf}}{\rightarrow}$ is global, whereas internal is not. Thus, $Tso$ and $Pso$ agree on both the store atomicity and the store buffering relaxations, which allows us to apply Thm. 4: $Tso^{\epsilon}$ is restored from $Pso$ by inserting non cumulative barriers between all $\overset{\text{ppo-tso}}{\rightarrow} \setminus \overset{\text{ppo-pso}}{\rightarrow} = WW$ pairs. And indeed, [20, V9] specifies that $TSO$ is obtained from $PSO$ by adding *StoreStore* barriers after each write.

```
     loop:
(a1) lwarx r1,0,r5
     [...]
(a2) stwcx. r2,0,r5
 (b) bne loop
```

**Fig. 4.** A generic Read-Modify-Write in PowerPC assembly

## 4 Synchronisation Idioms

Cumulativity of barriers may be challenging to implement, or too dear, as it slows considerably the performance of a program. We examine here programming idioms that use less costly barriers, though the trade-off may not be worth it.

### 4.1 Atomicity

Architectures provide special instructions to ensure *atomicity* to a given operation, such as `lwarx` and `stwcx.` in Power [17], or `ldstub` in Sparc [20]. We assume special read and write events to represent the accesses performed by such instructions, which we write $r^*$ and $w^*$. We write $\mathbb{R}^*$ (resp. $\mathbb{W}^*$) for the set of special reads (resp. writes) included in $\mathbb{R}$ (resp. $\mathbb{W}$), and define the effect of an atomic pair as follows:

$$\begin{aligned}\mathrm{atom}(r, w, \ell) \triangleq\ & r \in \mathbb{R}^* \wedge w \in \mathbb{W}^* \wedge \mathrm{loc}(r) = \mathrm{loc}(w) = \ell \wedge \\ & r = \max_{\xrightarrow{\mathrm{po}}}(\{m \mid m \in (\mathbb{R}^* \cup \mathbb{W}^*) \wedge m \xrightarrow{\mathrm{po}} w\}) \wedge \\ & \neg(\exists w', \mathrm{proc}(w') \neq \mathrm{proc}(r) \wedge \mathrm{loc}(w') = \ell \wedge r \xrightarrow{\mathrm{fr}} w' \wedge w' \xrightarrow{\mathrm{ws}} w)\end{aligned}$$

Thus, we consider that two events $r$ and $w$ form an atomic pair *w.r.t.* a location $\ell$—*i.e.* ensure atomicity to the accesses between them in $\xrightarrow{\mathrm{po}}$—if:

- they are both special and to $\ell$,
- $r$ is the maximal special read in $\xrightarrow{\mathrm{po}}$ before $w$, and
- no other processor wrote to $\ell$ between $r$ and $w$

We examine here a particular construct, *Read-Modify-Write*, on which crucial pieces of code such as *Test-And-Set* and *Compare-And-Swap* build. We give a generic PowerPC implementation of this construct, which would be similar in Alpha, in Fig. 4: in between the `lwarx` and `stwcx.`, the code is left to the choice of the programmer, avoiding the dynamic occurrence of other `lwarx` and `stwcx.` in the execution path from $(a_1)$ to $(a_2)$.

The $(a_1)/(a_2)$ pair of Fig. 4 is *successful* when `stwcx.` performs its store, in which case it sets the condition register appropriately, so that the code exits the loop.

### 4.2 Locks

An important use of Read-Modify-Write is made in the implementation of *lock* and *unlock* primitive that may be found in the litterature [17, Appendix B], as depicted in Fig. 5. These primitives paired together define a *critical section*, locked by a lock variable $\ell$. We write $\overset{\text{lock}_\ell}{\rightarrow}$ for the relation between two events in two distinct critical sections holding the same lock $\ell$ and $\text{Cs}_\ell$ for the set of critical sections with lock $\ell$; we write $\text{lock}_\ell$ for the function that outputs the $\overset{\text{lock}_\ell}{\rightarrow}$ relation when given an execution witness. We consider that $m_1 \overset{\text{lock}}{\rightarrow} m_2$ whenever there is a location $\ell$ such that $m_1 \overset{\text{lock}_\ell}{\rightarrow} m_2$.

```
     loop:
(a1) lwarx r6,0,r3
 (b) cmpw r4,r6
 (c) bne loop
(a2) stwcx. r5,0,r3
 (d) bne loop
     isync
     [...]
```

(a) Lock

```
     [...]
     lwsync
 (e) stw r4,0,r3
```

(b) Unlock

**Fig. 5.** Lock and unlock in PowerPC , where initially `r3 = ℓ`, `r4 = 0` and `r5 = 1`

**Drf programs have *Sc* semantics** We demonstrate, under several conditions on the $\overset{\text{lock}}{\rightarrow}$ relation, that drf programs have *Sc* semantics. By drf, we mean that each pair $(e_1, e_2)$ of *competing* accesses—that is, accesses to the same memory location, on different processors, at least one of which is a write—is such that each of its components is in its own critical section with the same lock variable $\ell$, a property we write $\text{locked}(e_1, e_2, \ell)$:

$$\text{compete}(m_1, m_2) \triangleq \text{loc}(m_1) = \text{loc}(m_2) \wedge \text{proc}(m_1) \neq \text{proc}(m_2) \wedge (m_1 \in \mathbb{W} \vee m_2 \in \mathbb{W})$$

$$\text{locked}(m_1, m_2, \ell) \triangleq \exists(\text{cs}_1, \text{cs}_2) \in (\text{Cs}_\ell \times \text{Cs}_\ell), \text{cs}_1 \neq \text{cs}_2 \wedge m_1 \in \text{cs}_1 \wedge m_2 \in \text{cs}_2$$

$$A.\ \text{drf}(X) \triangleq \forall m_1 m_2, \text{compete}(m_1, m_2) \Rightarrow (\exists \ell, \text{locked}(m_1, m_2, \ell))$$

As stated by the following theorem, we show that if the $\overset{\text{lock}}{\rightarrow}$ relation is:

- total on pairs of events in two distinct critical sections holding the same lock,
- compatible with $\overset{\text{ghb}}{\rightarrow}$ as well as with $\overset{\text{rf}}{\rightarrow}$ and $\overset{\text{po}}{\rightarrow}$,

*i.e.* satisfies what we call the drf requirements, drf programs have *Sc* semantics:

$$A.\ \text{wf-lock}(\overset{\text{lock}}{\rightarrow}) \triangleq (\forall m_1 m_2 \ell, (\text{locked}(m_1, m_2, \ell) \Rightarrow (m_1 \overset{\text{lock}_\ell}{\rightarrow} m_2 \vee m_2 \overset{\text{lock}_\ell}{\rightarrow} m_1) \wedge$$
$$(\forall \ell, \text{acyclic}(\overset{\text{lock}_\ell}{\rightarrow} \cup \overset{\text{ghb}}{\rightarrow})) \wedge (\forall \ell, \text{acyclic}(\overset{\text{lock}_\ell}{\rightarrow} \cup \overset{\text{rf}}{\rightarrow})) \wedge \text{acyclic}(\overset{\text{lock}}{\rightarrow} \cup \overset{\text{po}}{\rightarrow})$$

And indeed, if lock is compatible with $\xrightarrow{\text{ghb}}$, $\xrightarrow{\text{rf}}$ and $\xrightarrow{\text{po}}$, we know that $\xrightarrow{\text{hb-seq}} \cup \xrightarrow{\text{po}}$ is acyclic, thus:

**Theorem 5 (**Drf **requirements ensure** *Sc* **semantics to** drf **programs).**

$$\forall AX, A\,.\,\text{valid}(X) \wedge A\,.\,\text{drf}(X) \wedge (\forall \ell, A\,.\,\text{wf-lock}(\text{lock}(X))) \Rightarrow A\,.\,\text{check}_{Sc}(X)$$

**An example implementation** We define predicates taken and free using the atom predicate:

$$\text{taken}(\ell, r) \triangleq \exists w, \text{atom}(r, w, \ell) \wedge \text{val}(r) = 0 \wedge \text{val}(w) = 1$$
$$\text{free}(\ell, r, w) \triangleq r \xrightarrow{\text{po}} w \wedge \text{taken}(\ell, r) \wedge \text{loc}(w) = \ell \wedge \text{val}(w) = 0$$

A read $r$ takes a lock $\ell$ if it reads 0 from $\ell$ (indicating the lock was free) and it forms an atomic pair with a write $w$ writing 1 to $\ell$. Freeing the lock is the action of the next write event, if any, in $\xrightarrow{\text{po}}$ after a taken operation: it sets the lock variable to 0, and is in $\xrightarrow{\text{po}}$ (therefore in $\xrightarrow{\text{fr}}$ by uniproc) after the read that read the lock as free.

With these two constructs, we define semantics for lock and unlock primitives, such as depicted in Fig. 5:

$$\text{Lock}(\ell, (r, c)) \triangleq \text{taken}(\ell, r) \wedge c \in \mathbb{B} \wedge r \xrightarrow{\text{po}} c$$
$$\text{Unlock}(\ell, r, (b, w)) \triangleq \text{free}(\ell, r, w) \wedge b \in \mathbb{B} \wedge b \xrightarrow{\text{po}} w$$

A lock acquisition consists of a taken operation followed by an *import barrier* [17], which properties we will study in Sec. 4.2 An unlock consists of an *export barrier* [17] followed by a write event that frees the lock variable.

We define a critical section as a triple consisting of a lock and an unlock primitives with the same lock variable $\ell$, and the set of events that are in $\xrightarrow{\text{po}}$ between the barrier of the lock and the one of the unlock:

$$\text{cs}(\ell, (r, c), (b, w)) \triangleq (\text{Lock}(\ell, (r, c)), \{e \mid c \xrightarrow{\text{po}} e \xrightarrow{\text{po}} b\}, \text{Unlock}(\ell, r, (b, w)))$$

We consider two distinct critical sections $\text{cs}_1$ and $\text{cs}_2$ with the same variable $\ell$ to be serialised if $\text{cs}_2$ Lock's read reads from $\text{cs}_1$ Unlock's write:

$$\text{cs}_1 \xrightarrow{\text{css}_\ell} \text{cs}_2 \triangleq (\text{cs}_1, \text{cs}_2) \in (\text{Cs}_\ell \times \text{Cs}_\ell) \wedge \text{cs}_1 \neq \text{cs}_2 \wedge \exists r_1 c_1 b_1 w_1, \exists r_2 c_2 b_2 w_2, w_1 \xrightarrow{\text{rf}} r_2 \wedge$$
$$\text{cs}_1 = \text{cs}(\ell, (r_1, c_1), (b_1, w_1)) \wedge \text{cs}_2 = \text{cs}(\ell, (r_2, c_2), (b_2, w_2))$$

Finally, we define $\xrightarrow{\text{lock}_\ell}$ as the relation over events induced by $\xrightarrow{\text{css}_\ell}$, *i.e.* we consider two events to be ordered by $\xrightarrow{\text{lock}_\ell}$ if they are in two distinct critical sections with same lock $\ell$, the second reading from the first one, where $m \in \text{cs}$ expresses the fact that $m$ is between cs import and export barriers in $\xrightarrow{\text{po}}$:

$$m_1 \xrightarrow{\text{lock}_\ell} m_2 \triangleq (\exists \ell\, \text{cs}_1\, \text{cs}_2, m_1 \in \text{cs}_1 \wedge m_2 \in \text{cs}_2 \wedge \text{cs}_1 \xrightarrow{\text{css}_\ell} \text{cs}_2) \qquad \text{(base)}$$
$$\vee\, (\exists m, m_1 \xrightarrow{\text{lock}_\ell} m \xrightarrow{\text{lock}_\ell} m_2) \qquad \text{(transitivity)}$$

**Meeting the drf requirements**

*Totality of* $\overset{lock_\ell}{\rightarrow}$ We assume the existence of an init store to $\ell$, which is in $\overset{\text{lock}_\ell}{\rightarrow}$ with all critical sections to $\ell$, as it is the first store in $\overset{\text{ws}}{\rightarrow}$, and suppose that two critical sections to $\ell$ cannot be nested. Moreover, when two critical sections have their Lock's read reading from the same Unlock's write, one of their Lock's write takes the lock, which invalidates the reservation of the other one: therefore, its Lock waits until the lock is free, which means it appears later in the critical sections serialisation. Thus, if two critical sections have the same antecedent, they are ordered *w.r.t.* one another. This is the only part of our results whose mechanisation is not (yet) complete at the time of the submission.

*Import and export barriers* Let us assume two events $m_1$ and $m_2$ such that $m_1 \overset{\text{lock}_\ell}{\rightarrow} m_2$ in the base case. Writing $w_1$ for the unlock's write of the critical section $\text{cs}_1$ that protects $m_1$ and $r_2^*$ for the lock's read of the critical section $\text{cs}_2$ that protects $m_2$, we know $w_1 \overset{\text{rf}}{\rightarrow} r_2^*$.

Thus, if $\text{cs}_1$ export barrier $b_1$ orders write-write and read-write pairs locally, we have $m_1 \overset{\text{ab}}{\rightarrow} w_1$. Moreover, if $b_1$ is B-cumulative, we have $m_1 \overset{\text{ab}}{\rightarrow} r_2^*$. Formally, we define an export barrier as satisfying the following predicate:

$$\text{export}(b) \triangleq \forall m_1 m_2, (m_1 \overset{\text{po}}{\rightarrow} b \overset{\text{po}}{\rightarrow} m_2 \wedge \neg(m_1 \in \mathbb{W} \wedge m_2 \in \mathbb{R})) \Rightarrow (m_1 \overset{\text{ab}}{\rightarrow} m_2) \quad \text{(export base)}$$

$$\wedge\, \forall mwr, (m \overset{\text{po}}{\rightarrow} b \overset{\text{po}}{\rightarrow} w \overset{\text{rf}}{\rightarrow} r) \Rightarrow (m \overset{\text{ab}}{\rightarrow} r) \quad \text{(export B-cumulativity)}$$

Furthermore, if $\text{cs}_2$ import barrier $c_2$ orders read-write and read-read pairs locally, we have $r_2^* \overset{\text{ab}}{\rightarrow} m_2$.

$$\text{import}(b) \triangleq \forall m_1 m_2, (m_1 \overset{\text{po}}{\rightarrow} b \overset{\text{po}}{\rightarrow} m_2 \wedge m_1 \in \mathbb{R}) \Rightarrow (m_1 \overset{\text{ab}}{\rightarrow} m_2) \quad \text{(import base)}$$

This is already enough to guarantee compatibility of $\overset{\text{lock}_\ell}{\rightarrow}$ and $\overset{\text{ghb}}{\rightarrow}$, as $m_1 \overset{\text{lock}_\ell}{\rightarrow} m_2$ implies $m_1 \left(\overset{\text{ab}}{\rightarrow}\right)^+ m_2$. It also guarantees compatibility of $\overset{\text{lock}}{\rightarrow}$ and $\overset{\text{po}}{\rightarrow}$: as both relations are transitive, a cycle in their union leads to a cycle in $\overset{\text{lock}}{\rightarrow};\overset{\text{po}}{\rightarrow}$. Therefore, consider three events $m_1 \overset{\text{lock}}{\rightarrow} m_2 \overset{\text{po}}{\rightarrow} m_3$. In this case, $m_3$ is in $\overset{\text{po}}{\rightarrow}$ after $\text{cs}_2$ import barrier, thus we have $r_2^* \overset{\text{ab}}{\rightarrow} m_3$. By B-cumulativity of $\text{cs}_1$ export barrier, we also have $m_1 \overset{\text{ab}}{\rightarrow} r_2^*$. Therefore $m_1$ and $m_3$ are in this order in $\left(\overset{\text{ab}}{\rightarrow}\right)^+$, in which there can be no cycle.

$\overset{lock_\ell}{\rightarrow}$ *is compatible with* $\overset{rf}{\rightarrow}$ Since $\overset{\text{rf}}{\rightarrow}$ is trivially transitive, we examine as above a cycle in $\overset{\text{lock}_\ell}{\rightarrow};\overset{\text{rf}}{\rightarrow}$. Consider three events $m_1 \overset{\text{lock}_\ell}{\rightarrow} m_2 \overset{\text{rf}}{\rightarrow} m_3$ in the base case. If $m_3$ reads $m_2$ from the same processor, that is $m_2 \overset{\text{rfi}}{\rightarrow} m_3$, we know $m_1$ and $m_3$ are globally ordered: the previous case applies as $\overset{\text{rfi}}{\rightarrow} \subseteq \overset{\text{po}}{\rightarrow}$. Otherwise, we need

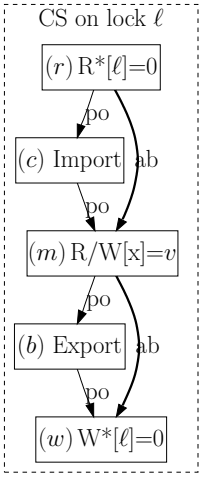



**Fig. 6.** Figures on locks

to extend the power of $\mathrm{cs}_2$ import barrier to B-cumulativity:

$$\mathrm{import}(b) \triangleq \quad \ldots \qquad \text{(import base)}$$
$$\wedge\, \forall m_1 w m_2, (m_1 \xrightarrow{\mathrm{po}} b \xrightarrow{\mathrm{po}} w \xrightarrow{\mathrm{rf}} m_2) \Rightarrow (m_1 \xrightarrow{\mathrm{ab}} m_2) \qquad \text{(import B-cumulativity)}$$

Thus, we have $r_2^* \xrightarrow{\mathrm{ab}} m_3$ by $\mathrm{cs}_2$ import barrier B-cumulativity, and $m_1 \xrightarrow{\mathrm{ab}} r_2^*$ by $\mathrm{cs}_1$ export barrier B-cumulativity.

Finally, as $\xrightarrow{\mathrm{lock}}$ satisfies the drf requirements, we show that critical sections implemented as in Sec. 4.2 guarantee *Sc* semantics to drf programs by Thm. 5:

**Theorem 6** (Drf **programs have** *Sc* **semantics).**

$$\forall AX, A.\ \mathrm{valid}(X) \wedge A.\ \mathrm{drf}(X) \wedge (\mathrm{lock}(X) = \xrightarrow{lock}) \Rightarrow A.\ \mathrm{check}_{Sc}(X)$$

We provide an insight on the power of import and export barriers, namely, in our model, an import barrier should order *RM* pairs statically; an export barrier should order *MW* pairs statically and both should be B-cumulative.

In Power [17], the export barrier may be either a `sync` or a `lwsync`: both of them are, to the best of our knowledge, B-cumulative, and order read-write and write-write pairs, which is compatible with our export predicate. The import barrier may be either a `lwsync`—which is compatible with our import predicate as it is B-cumulative and orders read-read and read-write pairs—or a sequence `bne;isync`, which we believe do not provide any cumulativity, though it orders

the appropriate pairs statically: our model would not guarantee *Sc* semantics to programs with locks implemented with this sequence.

## 5 Conclusion

*Minimality of the model* We advocate here the fact that, though several constructions such as barriers are specific to a given architecture, most of the reasoning on memory models can be done in the same generic terms. Thus we provide a unifying framework for reasoning about and formally comparing memory models. Moreover, we highlight crucial concepts that should be precisely defined to provide formal models, namely globality of read-from maps and preserved program order. This would certainly allow us to examine the trade-offs between two models, or code portability from one architecture to another. In this respect, we tried to minimise the number of our axioms, and also their scope, to embrace a large range of memory models. For example, we chose not to enforce that values do not come out of thin air [16], though it could be easily added to the model.

Moreover, the principal validity condition of our model is fairly simple: once $\xrightarrow{\text{ws}}$ and $\xrightarrow{\text{rf}}$ are available, checking the validity of a given execution resides in the acyclicity check of $\xrightarrow{\text{ghb}}$. This approach has been known for a long time for *SC* [15], and some recent verification tools use it for architectures with store buffer relaxation [11, 7] (*i.e.* $\xrightarrow{\text{rfi}}$ non global) such as *TSO*. We believe the present work provides alternative, machine checked, semantical foundations for these tools. More significantly, our formalism allows generalisation to even weaker memory models, that relax store atomicity (*i.e.* $\xrightarrow{\text{rfe}}$ non global).

*Modeling store-atomicity relaxation* We model indeed the potential non atomicity of writes by reasoning on globally performed events and considering $\xrightarrow{\text{rf}}$ not to be global in the general case. Other models handle non atomic writes by defining, for a sole store instruction, several write events [13], or one view order per processor [9, 1]. The complexity of these models is in sharp contrast with the models of architectures that do not relax store atomicity, such as Sparc [20]. We believe we have shown the simplicity and elegance of Sparc style models can very well be extended to models that relax store atomicity.

*Modeling Power* Our style of reasoning over globally performed events may lead to a model which is too coarse-grained—by this we mean too liberal— to study very weak architectures such as Power [17] and ARM [4], whose semi-formal definitions adopt the one order per processor approach. Nevertheless, we believe that our work already yields significant insights for such architectures. We indeed provide formal definitions and insights on the level of sophistication of barriers required to guarantee sequential consistency and to correctly implement locks: we model cumulativity, whereas [1] only models what was the state of the art at that time, *i.e.* non cumulative barriers.

Relaxing further some particularities of our model, namely the atomicity of reads and the existence of a coherence order per location, though widely assumed by modern architectures [20, 17, 3] may be discussed as we do in the following.

We interpret coherence [17] as the existence of a total order $\xrightarrow{\text{ws}}$ of stores to the same given memory location. We advocate actually the fact that a memory is a shared memory if and only if there exists some sort of coherence on it: thus, it is possible to consider a point in time when all processors agree—possibly *via* a cache protocol —on the value present in a memory location. We believe a model that would have no global coherence—*e.g.* a per-processor coherence order—would represent a distributed memory rather than a shared one: each processor would have the knowledge of its copy of a location, without considering the accesses performed to it by its neighbour, which contradicts the sharing of this location.

However, it may be possible to relax our coherence. We indeed force each processor to respect this order by considering $\xrightarrow{\text{ws}}$ as being global and by requiring the uniproc condition. However, older version of Power architectures—pre Power 4—do not respect our uniproc condition, as they do not include read-read pairs in $\xrightarrow{\text{po-loc}}$. This supposes refined cache protocoles to provide the illusion of a coherent memory [2]: relaxing coherence may be at that cost.

*Atomicity of reads* We believe a model with non atomic reads would also contradict the existence of a coherence order: in our model, this would lead to consider $\xrightarrow{\text{fr}}$ non global. Thus, a read may read a given value in its processor view, and another one in another processor view, which would reveal the existence of distinct copies of a given location.

## Acknowledgments

We thank Damien Doligez, Xavier Leroy, Susmit Sarkar and Peter Sewell for invaluable discussions, Assia Mahboubi and Vincent Siles for advices on the Coq development, Jules Villard and Boris Yakobowski for comments on a draft.

## References

1. A. Adir, H. Attiya, and G. Shurek. Information-Flow Models for Shared Memory with an Application to the PowerPC Architecture. *IEEE Transactions on Parallel and Distributed Systems*, May 2003.
2. S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29:66–76, 1995.
3. Alpha Architecture Reference Manual, Fourth Edition, 2002.
4. ARM. ARM architecture reference manual (armv7-a and armv7-r edition), April 2008. Available from ARM.
5. Arvind and J.-W. Maessen. Memory model = instruction reordering + store atomicity. In *Proc. ISCA*, June 2006.

6. Y. Bertot and P. Casteran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Springer Verlag, EATCS Texts in Theoretical Computer Science.
7. S. Burckhardt and M. Musuvathi. Effective program verification for relaxed memory models. In *CAV 2008*, 2008.
8. S. Burckhardt and M. Musuvathi. Memory model safety of programs. In *ECA-2*, 2008.
9. W. W. Collier. *Reasoning About Parallel Architectures*. Prentice-Hall, 1992.
10. M. Dubois and C. Scheurich. Memory access dependencies in shared-memory multiprocessors. *IEEE Transactions on Software Engineering*, 16(6), June 1990.
11. S. Hangal, D. Vahia, C. Manovit, J.-Y. J. Lu, and S. Narayanan. TSOTool: A Program for Verifying Memory Systems Using the Memory Consistency Model. In *Proc. ISCA 2004*, June 2004.
12. Intel 64 Architecture Memory Ordering White Paper, August 2007.
13. A formal specification of Intel Itanium processor family memory ordering. October 2002. Document Number 251429-001.
14. L. Lamport. How to make a correct multiprocess program execute correctly on a multiprocessor. *IEEE Trans. Comput.*, 46(7):779–782, 1979.
15. A. Landin, E. Hagersten, and S. Haridi. Race-free interconnection networks and multiprocessor consistency. *SIGARCH Comput. Archit. News*, 19(3):106–115, 1991.
16. J. Manson, W. Pugh, and S. V. Adve. The Java Memory Model. In *Proc. POPL 05*.
17. *Power ISA Version 2.06*. 2009.
18. S. Sarkar, P. Sewell, F. Zappa Nardelli, S. Owens, T. Ridge, T. Braibant, M. Myreen, and J. Alglave. The semantics of x86-CC multiprocessor machine code. In *Proc. POPL*, January 2009.
19. D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst.*, 10(2):282–312, 1988.
20. The Sparc Architecture Manual Versions 8 and 9, 1992 and 1994.

Centre de recherche INRIA Paris – Rocquencourt
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399