Stability in Weak Memory Models With Proofs

Jade Alglave[1,2] and Luc Maranget[2]

[1] University of Oxford [2] INRIA

Abstract. Concurrent programs running on weak memory models exhibit relaxed behaviours, making them hard to understand and to debug. We examine how to constrain the behaviour of such programs via synchronisation to ensure what we call their stability, i.e. that they behave as if they were running on a stronger model than the actual one, for example Sequential Consistency (SC). First, we define sufficient conditions ensuring stability to a program, and show that Power's locks and read-modify-write primitives meet them. Second, we minimise the amount of required synchronisation by characterising which parts of a given execution should be synchronised. Third, we characterise the programs stable from a weak architecture to SC. Finally, we present the offence tool implementing our approach, by placing either lock-based or lock-free synchronisation in a x86 or Power program to ensure its stability.

Concurrent programs running on modern multiprocessors exhibit subtle behaviours, making them hard to understand and to debug: modern architectures (*e.g.* x86 or Power) provide *weak memory models*, allowing optimisations such as *instruction reordering*, *store buffering* or *write atomicity relaxation* [2]. Thus an execution of a program may not be an interleaving of its instructions, as it would be on a Sequentially Consistent (SC) architecture [17]. Hence standard analyses for concurrent programs might be unsound, as noted by M. Rinard in [21]. There exist a few memory model aware verification tools [11, 15, 20, 26], but they often focus on one model at a time, or cannot handle the write atomicity relaxation exhibited for example by Power: generality remains a challenge.

Fortunately, we can force a program running on a weak architecture to behave as if it were running on a stronger one (*e.g.* SC) by using *synchronisation primitives*. Hence, as observed by S. Burckhart and M. Musuvathi in [12], *"we can sensibly verify the relaxed executions [...] by solving the following two verification problems separately: 1. Use standard verification methodology for concurrent programs to show that the [SC] executions [...] are correct. 2. Use specialized methodology for* memory model safety *verification [...]"*. Here, *memory model safety* means checking that the executions of a program, although running on a weak architecture, are actually SC. To apply standard verification techniques to concurrent programs running on weak memory models, we thus first need to ensure that our programs have a SC behaviour. S. Burckhart and M. Musuvathi focus in [12] on memory model safety for TSO [24]. We generalise their idea to a wider class of models (the one defined in [5], and recalled in Sec. 1): we examine how to force a program running on a weak architecture A_1 to behave as if running on a stronger one A_2, a property that we call *stability from A_1 to A_2*.

To ensure stability to a program, we examine the problem of placing *lock-based* or *lock-free* synchronisation primitives in a program. We call *synchronisation mapping* an insertion of synchronisation primitives (either *barriers* (or *fences*), *read-modify-writes*, or *locks*) in a program. We study whether a given synchronisation mapping ensures stability to a program running on a weak memory model, *e.g.* that we placed enough primitives in the code to ensure that it only has SC executions. D. Shasha and M. Snir proposed in [23] the *delay set analysis* to insert barriers in a program, but their work does not provide any semantics for weak memory models. Hence questions remain $w.r.t.$ the adequacy of their method in the context of such models. On the contrary, locks allow the programmer to ignore the details of the memory model, thanks to the *data race free guarantee* (DRF guarantee) proposed in [3] by S. Adve and M. Hill.

Yet, from a compilation point of view, locks are costly. As noted by S. Adve and H.-J. Boehm in [4], *"[o]n hardware that relaxes write atomicity [..., e.g. Power], it is often unclear that more efficient mappings (than the use of locks) are possible; even the fully fenced implementation may not be sequentially consistent."* Hence not only do we need to examine the *soundness* of our synchronisation mappings (*i.e.* that they actually ensure stability to a given program), but also their cost. We present here several new contributions:

1. We define in Sec. 2 sufficient conditions on synchronisation to ensure stability to a program. As an illustration, we provide in Sec. 3 semantics to the locks and read-modify-writes (rmw) of the Power architecture [1] (*i.e.* to the `lwarx` and `stwcx.` instructions) and show in Coq that they meet these conditions.
2. We propose along the way several synchronisation mappings, which we prove in Coq to enforce a SC behaviour to an x86 or Power program.
3. We optimise these mappings by generalising in Sec. 4 the approach of [23] to weak memory models and both lock-based and lock-free synchronisation, and characterise in Coq the executions stable from a weak architecture to SC.
4. We describe in Sec. 5 our new offence tool, which places either lock-based or lock-free synchronisation in a x86 or Power assembly program to ensure its stability, following the aforementionned characterisation. We detail how we used offence to test and measure the cost of our synchronisation mappings.

We formalised all our results in Coq. The Coq development[2], the documentation and sources of offence and all the experimental details can be found online[1].

1 Context

We give here the technical background on which we build our results and proofs. This section summarises the generic model [5] on which we build, which embraces SC [17], Sun TSO, PSO and RMO [24], Alpha [7] and a fragment of Power [1].

Executions An *event* e is a read or a write, composed of a direction R (read) or W (write), a location $\mathrm{loc}(e)$, the instruction from which it comes $\mathrm{ins}(e)$, a value

[1] `http://diy.inria.fr/offence` [2] `http://moscova.inria.fr/~alglave/stability`

iriw

P_0	P_1	P_2	P_3
(a) `r1` $\leftarrow$ `x`	(c) `r3` $\leftarrow$ `y`	(e) `x` $\leftarrow$ `1`	(f) `y` $\leftarrow$ `2`
(b) `r2` $\leftarrow$ `y`	(d) `r4` $\leftarrow$ `x`		

Observed? `r1=1`; `r2=0`; `r3=2`; `r4=0`;

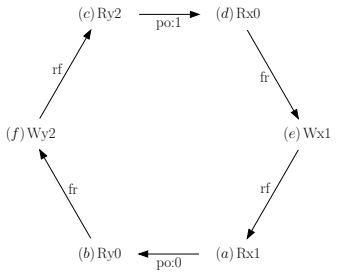


Fig. 1. The **iriw** test and a non-SC execution

val(e), an originating processor proc(e), and a unique identifier. We represent each instruction by the events it issues. In Fig. 1, we associate the store (e) $x \leftarrow$ `1` on P_2 with the event $(e)\mathrm{W}x1$. We write $\mathbb{E}$ for the set of events, and $\mathbb{W}$ (resp. $\mathbb{R}$) for the subset of write (resp. read) events. We write w (resp. r) for a write (resp. read), and m or e when the direction is irrelevant.

We associate a program with an *event structure* $E \triangleq (\mathbb{E}, \overset{\mathrm{po}}{\rightarrow})$, composed of its events $\mathbb{E}$ and the *program order* $\overset{\mathrm{po}}{\rightarrow}$, a per-processor total order over $\mathbb{E}$. In Fig. 1, the read (a) from x on P_0 is in program order with the read (b) from y on P_0, *i.e.* $(a)\mathrm{R}x1 \overset{\mathrm{po}}{\rightarrow} (b)\mathrm{R}y0$. The $\overset{\mathrm{dp}}{\rightarrow}$ relation (included in $\overset{\mathrm{po}}{\rightarrow}$, the source being a read) models the dependencies between instructions, *e.g.* when we compute the address of a load or store from the value of a preceding load.

Given an event structure E, we represent an execution $X \triangleq (\overset{\mathrm{ws}}{\rightarrow}, \overset{\mathrm{rf}}{\rightarrow})$ of the corresponding program by two relations over $\mathbb{E}$. The *write serialisation* $\overset{\mathrm{ws}}{\rightarrow}$ is a per-location total order on writes modeling the *memory coherence* assumed by modern architectures [13], linking a write w to any write w' to the same location hitting the memory after w. The *read-from map* $\overset{\mathrm{rf}}{\rightarrow}$ links a write w to a read r from the same location that reads from w. We derive the *from-read map* $\overset{\mathrm{fr}}{\rightarrow}$ from $\overset{\mathrm{ws}}{\rightarrow}$ and $\overset{\mathrm{rf}}{\rightarrow}$. A read r is in $\overset{\mathrm{fr}}{\rightarrow}$ with a write w when the write w' from which r reads hit the memory before w did: $r \overset{\mathrm{fr}}{\rightarrow} w \triangleq \exists w', w' \overset{\mathrm{rf}}{\rightarrow} r \wedge w' \overset{\mathrm{ws}}{\rightarrow} w$. We write $\overset{\mathrm{com}}{\rightarrow} \triangleq \overset{\mathrm{ws}}{\rightarrow} \cup \overset{\mathrm{fr}}{\rightarrow} \cup \overset{\mathrm{rf}}{\rightarrow}$ for the union of our *communication relations*.

In Fig. 1, the specified outcome corresponds to the execution on the right, if each location and register initially holds 0. If `r1=1` in the end, the read (a) read its value from the write (e) on P_2, hence $(e) \overset{\mathrm{rf}}{\rightarrow} (a)$. If `r2=0`, the read (b) read its value from the initial state, thus before the write (f) on P_3, hence $(b) \overset{\mathrm{fr}}{\rightarrow} (f)$. Similarly, we have $(f) \overset{\mathrm{rf}}{\rightarrow} (c)$ from `r3=2`, and $(d) \overset{\mathrm{fr}}{\rightarrow} (e)$ from `r4=0`.

Architectures In a shared-memory multiprocessor, a write may be committed first into a store buffer, then into a cache, and finally into memory. Hence, while a write transits in store buffers and caches, a processor may read a past value.

We model this by some subrelation of $\overset{\text{rf}}{\rightarrow}$ being *non-global*: they can be ignored by some processors. We write $\overset{\text{rfi}}{\rightarrow}$ (resp. $\overset{\text{rfe}}{\rightarrow}$) for the *internal* (resp. *external*) read-from map, *i.e.* a read-from map between two events from the same (resp. distinct) processor(s). Hence we model a read r by a processor P_0 reading from a write w in P_0's store buffer by $w \overset{\text{rfi}}{\rightarrow} r$ being non-global. When r reads from a write w by a distinct processor P_1 into a cache shared by P_0 and P_1 only (a case of *write atomicity* relaxation [2]), $w \overset{\text{rfe}}{\rightarrow} r$ is non-global, and w is said to be *non-atomic*. TSO authorises *e.g.* store buffering (*i.e.* $\overset{\text{rfi}}{\rightarrow}$ is non-global) but considers stores to be atomic (*i.e.* $\overset{\text{rfe}}{\rightarrow}$ is global). We write $\overset{\text{grf}}{\rightarrow}$ for the global subrelation of $\overset{\text{rf}}{\rightarrow}$. We consider $\overset{\text{ws}}{\rightarrow}$ and $\overset{\text{fr}}{\rightarrow}$ global, since $\overset{\text{ws}}{\rightarrow}$ is the order in which the writes to a certain location hit the memory.

Moreover, some pairs of events in the program order may be reordered. Thus only a subset of the pairs of events in $\overset{\text{po}}{\rightarrow}$, gathered in a subrelation $\overset{\text{ppo}}{\rightarrow}$ (*preserved program order*), is guaranteed to occur in this order. TSO for example authorises write-read pairs to be reordered, but nothing else: $\overset{\text{ppo}}{\rightarrow} = \overset{\text{po}}{\rightarrow} \setminus (\overset{\text{po}}{\rightarrow} \cap (\mathbb{W} \times \mathbb{R}))$.

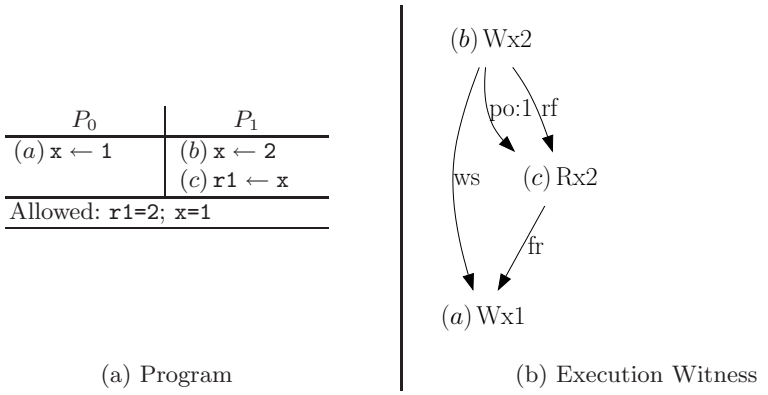
Finally, architectures provide barrier instructions to order certain pairs of events. We gather the orderings induced by barriers in the global relation $\overset{\text{ab}}{\rightarrow}$. Following [5], the relation $\overset{\text{fence}}{\rightarrow} \subseteq \overset{\text{po}}{\rightarrow}$ induced by a barrier `fence` is *non-cumulative* when it orders certain pairs of events surrounding the barrier. For example, the x86 `mfence` barrier is a non-cumulative barrier ordering write-read pairs only: we have $(w \overset{\text{mfence}}{\rightarrow} r) \Rightarrow (w \overset{\text{ab}}{\rightarrow} r)$. Power `bne; isync` sequence is a non-cumulative barrier for read-read and read-write pairs: we have $(r \overset{\text{bne;isync}}{\rightarrow} m) \Rightarrow (r \overset{\text{ab}}{\rightarrow} m)$.

The relation $\overset{\text{fence}}{\rightarrow}$ is *A-cumulative* (resp. *B-cumulative*) when it makes the writes atomic (*e.g.* by flushing the store buffers and caches). For example, Power `sync` barrier is non-, A- and B-cumulative for all pairs: we have $(m_1 \overset{\text{sync}}{\rightarrow} m_2)$ (resp. $(m_1 \overset{\text{rf}}{\rightarrow} w \overset{\text{sync}}{\rightarrow} m_2)$, $(m_1 \overset{\text{sync}}{\rightarrow} w \overset{\text{rf}}{\rightarrow} m_2)$) implies $(m_1 \overset{\text{ab}}{\rightarrow} m_2)$. Power `lwsync` is non-, A- and B-cumulative for all pairs except write-read ones; we have $(m_1 \overset{\text{lwsync}}{\rightarrow} m_2)$ (resp. $(m_1 \overset{\text{rf}}{\rightarrow} w \overset{\text{lwsync}}{\rightarrow} m_2)$, $(m_1 \overset{\text{lwsync}}{\rightarrow} w \overset{\text{rf}}{\rightarrow} m_2)$) implies $(m_1 \overset{\text{ab}}{\rightarrow} m_2)$ except when $(m_1, m_2) \in (\mathbb{W} \times \mathbb{R})$.

An *architecture* $A \triangleq (\text{ppo}, \text{grf}, \text{ab})$ specifies the function ppo (resp. grf, ab) returning the relation $\overset{\text{ppo}}{\rightarrow}$ (resp. $\overset{\text{grf}}{\rightarrow}$, $\overset{\text{ab}}{\rightarrow}$) when given an execution.

Validity The $\text{uniproc}(E, X) \triangleq \text{acyclic}(\overset{\text{com}}{\rightarrow} \cup \overset{\text{po-loc}}{\rightarrow})$ condition (where $\overset{\text{po-loc}}{\rightarrow}$ is the program order restricted to events with the same location) forces a processor in a multiprocessor context to respect the memory *coherence* [13]. The $\text{thin}(E, X) \triangleq \text{acyclic}(\overset{\text{rf}}{\rightarrow} \cup \overset{\text{dp}}{\rightarrow})$ condition prevents executions where values seem to come *out of thin air* [19]. We define the *global happens-before* relation $A.\text{ghb}(E, X)$ of an execution (E, X) on an architecture A as the union of the relations global on A:

$$A.\text{ghb}(E, X) \triangleq \overset{\text{ws}}{\rightarrow} \cup \overset{\text{fr}}{\rightarrow} \cup \overset{\text{ppo}}{\rightarrow} \cup \overset{\text{grf}}{\rightarrow} \cup \overset{\text{ab}}{\rightarrow}$$



(a) Program (b) Execution Witness

Fig. 2. A program and an execution witness

An execution (E, X) is *valid* on an architecture A, written $A\,.\,\text{valid}(E, X)$, when the relation $A\,.\,\text{ghb}(E, X)$ is acyclic (together with the two above checks):

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

A hierarchy of architectures We consider an architecture A_1 to be *weaker* than an architecture A_2, written $A_1 \leq A_2$, when A_1 authorises at least all the executions valid on A_2 (writing $\xrightarrow{\mathrm{r}_i}$ for the relation $\xrightarrow{\mathrm{r}}$ *w.r.t.* A_i):

$$A_1 \leq A_2 \;\triangleq\; \xrightarrow{\mathrm{ppo}_1} \subseteq \xrightarrow{\mathrm{ppo}_2} \wedge \xrightarrow{\mathrm{grf}_1} \subseteq \xrightarrow{\mathrm{grf}_2}$$

For example, TSO authorises store buffering, which means the internal $\xrightarrow{\mathrm{rf}}$ is not global. Hence for TSO, $\xrightarrow{\mathrm{grf}}$ is restricted to the external one, *i.e.* $\xrightarrow{\mathrm{grf}} = \xrightarrow{\mathrm{rfe}}$. Moreover, TSO authorises the reordering of write-read pairs, hence its $\xrightarrow{\mathrm{ppo}}$ is strictly included in $\xrightarrow{\mathrm{po}}$. Therefore, TSO is weaker than SC.

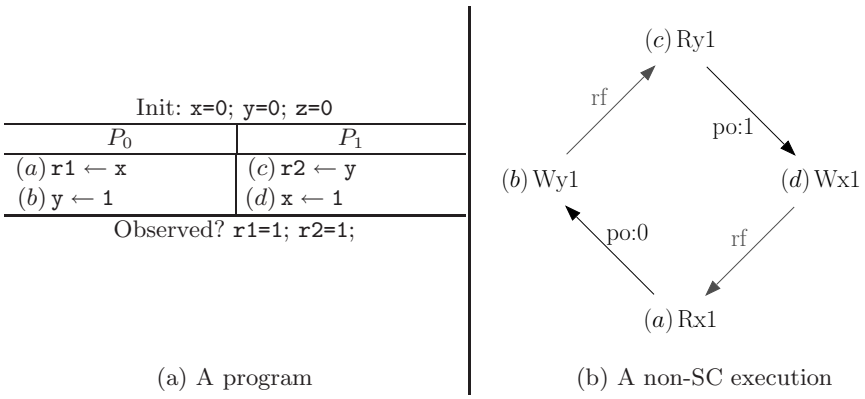
As shown in [5], a weak execution (*i.e.* valid on a weak architecture A_1) is valid on a stronger one A_2 if and only if:

Lem. 1 (Characterisation).

$$\forall A_1 \leq A_2, \forall EX, (A_1\,.\,\text{valid}(E, X) \wedge \text{acyclic}(A_2\,.\,\text{ghb}(E, X))) \Leftrightarrow A_2\,.\,\text{valid}(E, X)$$

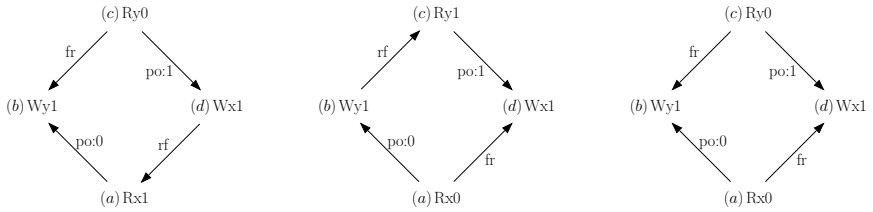
For example, the execution (E, X) of Fig. 2(b) is valid on TSO and $SC\,.\,\text{ghb}(E, X)$ is acyclic, hence (E, X) is also valid on SC.

Building a valid execution from an order We consider two relations to be *compatible* when their union is acyclic. Consider an architecture A without barriers, *i.e.* $\xrightarrow{\mathrm{ab}_A} = \emptyset$. The characterisation above means in particular that, for any event structure E, one can build an execution, associated with E and valid on A, from a total order $\xrightarrow{\mathrm{o}}$ on $\text{evts}(E)$ compatible with $A\,.\,\text{ppo}(E)$.



(a) A program (b) A non-SC execution

Fig. 3. A program and a non-SC execution



(a) (b) (c)

Fig. 4. SC executions for the test of Fig. 3(a)

Consider *e.g.* the event structure $(\{(a),(b),(c),(d)\}, \{(a) \overset{\text{po}}{\to} (b), (c) \overset{\text{po}}{\to} (d)\})$ associated with the program of Fig. 3. On SC we have $(a) \overset{\text{ppo}}{\to} (b)$ and $(c) \overset{\text{ppo}}{\to} (d)$. Hence we can build a valid SC execution from the order $(a) \overset{\text{o}}{\to} (b) \overset{\text{o}}{\to} (c) \overset{\text{o}}{\to} (d)$, which is the one we give in Fig. 4(b). The first write in the order $\overset{\text{o}}{\to}$ is (b), a write to y, which is immediately followed by the read (c) to y with the same value, hence we have $(b) \overset{\text{rf}}{\to} (c)$. There is no write preceding the read (a) from x, hence (a) reads from the initial state. Moreover, this initial write to x precedes the write (d) in $\overset{\text{ws}}{\to}$, hence $(a) \overset{\text{fr}}{\to} (d)$.

We write $\text{rf}(\overset{\text{o}}{\to})$ (resp. $\text{ws}(\overset{\text{o}}{\to})$) for the $\overset{\text{rf}}{\to}$ (resp. $\overset{\text{ws}}{\to}$) extracted from $\overset{\text{o}}{\to}$. We have $(x,y) \in \text{rf}(\overset{\text{o}}{\to})$ when x is a write and y a read, both to the same location ℓ, with the same value, and x is the maximal previous write to ℓ before y in $\overset{\text{o}}{\to}$, *i.e.* $\neg(\exists z \in \mathbb{W}_\ell, x \overset{\text{o}}{\to} z \overset{\text{o}}{\to} y)$. We have $(x,y) \in \text{ws}(\overset{\text{o}}{\to})$ when x and y are writes to the same location and $x \overset{\text{o}}{\to} y$. Formally, we have:

Lem. 2 (Extraction of a valid execution from an order).

$$\forall EX \overset{\text{o}}{\to}, \text{total-order}(\overset{\text{o}}{\to}, \text{evts}(E)) \wedge \text{acyclic}(\overset{\text{o}}{\to} \cup A.\text{ppo}(E)) \wedge \\ X = (\text{ws}(\overset{\text{o}}{\to}), \text{rf}(\overset{\text{o}}{\to})) \Rightarrow A.\text{valid}(E,X)$$

Proof Since $\overset{\text{o}}{\to}$ is a total order, $\text{ws}(\overset{\text{o}}{\to})$ is by definition a per-location total order on writes. Moreover, $\text{rf}(\overset{\text{o}}{\to})$ trivially satisfies the definition of read-from map.

We define the extracted from-read map $\text{fr}(\overset{\text{o}}{\to})$ as:

$$(r,w) \in \text{fr}(\overset{\text{o}}{\to}) \triangleq \exists w_r, (w_r, r) \in \text{rf}(\overset{\text{o}}{\to}) \wedge (w_r, w) \in \text{ws}(\overset{\text{o}}{\to})$$

We define the extracted communication $\text{com}(\overset{\text{o}}{\to})$ as:

$$\text{com}(\overset{\text{o}}{\to}) \triangleq \text{ws}(\overset{\text{o}}{\to}) \cup \text{rf}(\overset{\text{o}}{\to}) \cup \text{fr}(\overset{\text{o}}{\to})$$

Since $\overset{\text{ab}_A}{\to} = \emptyset$, we know that $A.\text{ghb}(E,X)$ is included in $\text{com}(\overset{\text{o}}{\to}) \cup A.\text{ppo}(E)$.

By definition, $\text{ws}(\overset{\text{o}}{\to})$ and $\text{rf}(\overset{\text{o}}{\to})$ are included in $\overset{\text{o}}{\to}$. Let us show that $\text{fr}(\overset{\text{o}}{\to})$ is included in $\overset{\text{o}}{\to}$: consider $(r,w) \in \text{fr}(\overset{\text{o}}{\to})$; since $\overset{\text{o}}{\to}$ is total, we have either $r \overset{\text{o}}{\to} w$ or $w \overset{\text{o}}{\to} r$. Suppose $w \overset{\text{o}}{\to} r$; since $(r,w) \in \text{fr}(\overset{\text{o}}{\to})$, there exists w_r such that $(w_r, r) \in \text{rf}(\overset{\text{o}}{\to})$, hence $w_r \overset{\text{o}}{\to} r$, and $(w_r, w) \in \text{ws}(\overset{\text{o}}{\to})$, hence $w_r \overset{\text{o}}{\to} w$. Thus we have $w_r \overset{\text{o}}{\to} w \overset{\text{o}}{\to} r$. By definition of $\text{rf}(\overset{\text{o}}{\to})$, w_r is the maximal previous write to $\text{loc}(r)$ before r in $\overset{\text{o}}{\to}$, hence a contradiction.

Hence $A.\text{ghb}(E,X)$ is included in $\overset{\text{o}}{\to} \cup A.\text{ppo}(E)$. Moreover, $\overset{\text{o}}{\to}$ is compatible with $A.\text{ppo}(E)$ by hypothesis (*i.e.* their union is acyclic). Hence there cannot be any cycle in $A.\text{ghb}(E,X)$. □

In the following, we consider A_2 to be without barriers, *i.e.* $\overset{\text{ab}_2}{\to} = \emptyset$. We write $\overset{\text{ghb}_2}{\to}$ for $A_2.\text{ghb}(E,X)$.

2 Covering relations

We examine now how to force the executions of a program running on a weak architecture A_1 to be valid on a stronger one A_2, which we call *stability from* A_1 *to* A_2, *i.e.* we examine when the following property holds for all (E, X):

$$\text{stable}_{\text{A}_1,\text{A}_2}(E, X) \triangleq A_1\,.\,\text{valid}(E, X) \Rightarrow A_2\,.\,\text{valid}(E, X)$$

The execution of **iriw** in Fig. 1 is not stable from Power to SC, for it is valid on Power yet not on SC. We can stabilise an execution by using *synchronisation idioms*, *e.g.* barriers or locks. Synchronisation idioms *arbitrate conflicts* between accesses, *i.e.* ensure that one out of two conflicting accesses occurs before the other. We formalise this with an irreflexive *conflict* relation $\overset{\text{c}}{\rightarrow}$ over events, such that $\forall xy, x \overset{\text{c}}{\rightarrow} y \Rightarrow \neg(y \overset{\text{po}}{\rightarrow} x)$ and a *synchronisation* relation $\overset{\text{s}}{\rightarrow}$ over events. An execution (E, X) is *covered* when $\overset{\text{c}}{\rightarrow}$ is *arbitrated* by $\overset{\text{s}}{\rightarrow}$:

$$\text{covered}_{c,s}(E, X) \;\triangleq\; \forall xy, x \overset{\text{c}}{\rightarrow} y \Rightarrow x \overset{\text{s}}{\rightarrow} y \vee y \overset{\text{s}}{\rightarrow} x$$

We consider a relation $\overset{\text{s}}{\rightarrow}$ to be *covering* when ordering by $\overset{\text{s}}{\rightarrow}$ the conflicting accesses of an execution (E, X) valid on A_1 guarantees its validity on A_2, *i.e.* the synchronisation $\overset{\text{s}}{\rightarrow}$ arbitrates enough conflicts to enforce a strong behaviour:

$$\text{covering}(\overset{\text{c}}{\rightarrow}, \overset{\text{s}}{\rightarrow}) \triangleq \forall EX, (A_1\,.\,\text{valid}(E, X) \wedge \text{covered}_{c,s}(E, X)) \Rightarrow A_2\,.\,\text{valid}(E, X)$$

Lock-based synchronisation For example, the DRF guarantee [3] ensures that if the *competing accesses*, defined below, of an execution are ordered by locks, then this execution is SC. Hence locks are covering *w.r.t.* the competing accesses. Following [3], two events are *competing* if they are from two distinct processors, relative to the same location, and one of them at least is a write (*e.g.* in Fig. 1, the read (a) from x on P_0 and the write (e) to x on P_2):

$$m_1 \overset{\text{cmp}}{\leftrightarrow} m_2 \triangleq \text{proc}(m_1) \neq \text{proc}(m_2) \wedge \text{loc}(m_1) = \text{loc}(m_2) \wedge (m_1 \in \mathbb{W} \vee m_2 \in \mathbb{W})$$

We describe the ordering induced by locks by a relation $\overset{\text{lock}}{\rightarrow}$ (instantiated in Sec. 3.1) over $\mathbb{E}$ compatible with $\overset{\text{com}}{\rightarrow}$ (*i.e.* $\text{acyclic}(\overset{\text{lock}}{\rightarrow} \cup \overset{\text{com}}{\rightarrow})$), corresponding in Fig. 1 to placing locks to a variable ℓ_1 on the accesses (a), (d) and (e) relative to x, and locks to a different variable ℓ_2 on the accesses (b), (c) and (f) relative to y. Thus we have a cycle in $\overset{\text{lock}}{\rightarrow} \cup \overset{\text{po}}{\rightarrow}$: $(a) \overset{\text{po}}{\rightarrow} (b) \overset{\text{lock}}{\rightarrow} (f) \overset{\text{lock}}{\rightarrow} (c) \overset{\text{po}}{\rightarrow} (d) \overset{\text{lock}}{\rightarrow} (e) \overset{\text{lock}}{\rightarrow} (a)$. If $\overset{\text{lock}}{\rightarrow} \cup \overset{\text{po}}{\rightarrow}$ is acyclic, then the execution of Fig. 1 is forbidden. Formally, we have:

Lem. 3. $\text{acyclic}(\overset{\text{lock}}{\rightarrow} \cup \overset{\text{po}}{\rightarrow}) \Rightarrow \text{covering}(\overset{\text{cmp}}{\leftrightarrow}, (\overset{\text{lock}}{\rightarrow} \cup \overset{\text{po}}{\rightarrow})^+)$

Proof Let (E, X) be valid on A_1 and covered. Suppose by contradiction that there is a cycle in $\overset{\text{ghb}_2}{\rightarrow}$, which is by definition a cycle in $\overset{\text{ws}}{\rightarrow} \cup \overset{\text{fr}}{\rightarrow} \cup \overset{\text{grf}_2}{\rightarrow} \cup \overset{\text{ppo}_2}{\rightarrow}$. The events in $\overset{\text{ws}}{\rightarrow}$, $\overset{\text{fr}}{\rightarrow}$ or $\overset{\text{rf}}{\rightarrow}$ and from distinct processors are competing. Since (E, X) is covered, $\overset{\text{sync}}{\rightarrow}$ orders the competing accesses according to $\overset{\text{com}}{\rightarrow}$. The remaining events in $\overset{\text{com}}{\rightarrow}$ belong to the same processor, hence are in $\overset{\text{po}}{\rightarrow}$ by uniproc. Moreover, we know $\overset{\text{ppo}_2}{\rightarrow} \subseteq \overset{\text{po}}{\rightarrow}$. Hence a cycle in $\overset{\text{ws}}{\rightarrow} \cup \overset{\text{fr}}{\rightarrow} \cup \overset{\text{grf}_2}{\rightarrow} \cup \overset{\text{ppo}_2}{\rightarrow}$ is a cycle in $\overset{\text{sync}}{\rightarrow} \cup \overset{\text{po}}{\rightarrow}$, which contradicts the irreflexivity of $\overset{\text{s}_{\text{drf}}}{\rightarrow}$. □

This lemma leads to a mapping which we call L (for locks), which simply places a lock by the same lock variable on each side of a given conflict edge; following Lem. 3, it gives a SC behaviour to a program.

Lock-free synchronisation We give here an example of a covering lock-free synchronisation relation. A program can distinguish between two architectures $A_1 \leq A_2$ for one of two reasons. First, if the program involves a pair (x, y) maintained in program order on A_2 (*i.e.* $x \overset{\text{ppo}_2}{\rightarrow} y$) but not on A_1 (*i.e.* $\neg(x \overset{\text{ppo}_1}{\rightarrow} y)$). In Fig. 1, we have $(a) \overset{\text{po}}{\rightarrow} (b)$. Hence on a strong architecture A_2 such as SC where $\overset{\text{ppo}_2}{\rightarrow} = \overset{\text{po}}{\rightarrow}$, we have $(a) \overset{\text{ppo}_2}{\rightarrow} (b)$. On a weak architecture A_1 such as Power, where the read-read pairs in program order are not maintained, we have $\neg((a) \overset{\text{ppo}_1}{\rightarrow} (b))$.

Second, if the program reads from a write atomic on A_2 but not on A_1. In Fig. 1, we have $(e) \overset{\text{rfe}}{\rightarrow} (a)$. On a strong architecture A_2 such as SC where the writes are atomic, *i.e.* $\overset{\text{grf}}{\rightarrow} = \overset{\text{rf}}{\rightarrow}$, we have $(e) \overset{\text{grf}}{\rightarrow} (a)$. On a weak architecture A_1 such as Power, which relaxes write atomicity, we have $\neg((e) \overset{\text{grf}}{\rightarrow} (a))$. We call such reads *fragile reads* and define them as ($\overset{\text{r}_2 \setminus 1}{\rightarrow} \triangleq \overset{\text{r}_2}{\rightarrow} \setminus \overset{\text{r}_1}{\rightarrow}$ being the set difference):

$$\text{fragile}(r) \triangleq \exists w, w \overset{\text{grf}_2 \setminus 1}{\rightarrow} r$$

We consider such differences between architectures as conflicts, and formalise this notion as follows. We consider that two events form a *fragile pair* (written $\overset{\text{frag}}{\rightarrow}$) if they are maintained in the program order on A_2, and either they are not maintained in the program order on A_1, or the first event is a fragile read:

$$m_1 \overset{\text{frag}}{\rightarrow} m_2 \triangleq m_1 \overset{\text{ppo}_2}{\rightarrow} m_2 \wedge \left(\neg(m_1 \overset{\text{ppo}_1}{\rightarrow} m_2) \vee \text{fragile}(m_1)\right)$$

An execution is covered if the barrier relation $\overset{\text{ab}_1}{\rightarrow}$ arbitrates the fragile pairs. In Fig. 1, this corresponds to placing a barrier between (c) and (d) on P_1, *i.e.* $(c) \overset{\text{ab}_1}{\rightarrow} (d)$, and another barrier between (a) and (b) on P_0, *i.e.* $(a) \overset{\text{ab}_1}{\rightarrow} (b)$. Hence we have a cycle in $\overset{\text{ab}_1}{\rightarrow} \cup \overset{\text{rf}}{\rightarrow}$: $(d) \overset{\text{rfe}}{\rightarrow} (a) \overset{\text{ab}_1}{\rightarrow} (b) \overset{\text{rfe}}{\rightarrow} (c) \overset{\text{ab}_1}{\rightarrow} (d)$. If $\overset{\text{ab}_1}{\rightarrow}$ is *A-cumulative* $w.r.t.$ $\overset{\text{grf}_2 \setminus 1}{\rightarrow}$ (*i.e.* $\forall xyz, (x \overset{\text{grf}_2 \setminus 1}{\rightarrow} y \wedge y \overset{\text{ab}_1}{\rightarrow} z) \Rightarrow x \overset{\text{ghb}_1}{\rightarrow} z$), we create a cycle in $\overset{\text{ghb}_1}{\rightarrow}$, which forbids the execution: $(d) \overset{\text{ghb}_1}{\rightarrow} (b) \overset{\text{ghb}_1}{\rightarrow} (d)$. Indeed, we show that if $\overset{\text{ab}_1}{\rightarrow}$ is A-cumulative $w.r.t.$ $\overset{\text{grf}_2 \setminus 1}{\rightarrow}$ then $\overset{\text{ab}_1}{\rightarrow}$ is covering:

Lem. 4. $(\forall xyz, (x \overset{\text{grf}_2 \setminus 1}{\rightarrow} y \wedge y \overset{\text{ab}_1}{\rightarrow} z) \Rightarrow x \overset{\text{ghb}_1}{\rightarrow} z) \Rightarrow \text{covering}(\overset{\text{frag}}{\rightarrow}, \overset{\text{ab}_1}{\rightarrow})$

Proof Consider an execution (E, X) valid on A_1 and covered. Suppose by contradiction that there is a cycle in $\overset{\text{ghb}_2}{\rightarrow}$, which is by definition a cycle in $\overset{\text{ws}}{\rightarrow} \cup \overset{\text{fr}}{\rightarrow} \cup \overset{\text{grf}_2}{\rightarrow} \cup \overset{\text{ppo}_2}{\rightarrow}$. Since $\overset{\text{ws}}{\rightarrow}$, $\overset{\text{fr}}{\rightarrow}$ and $\overset{\text{ppo}_1}{\rightarrow}$ are included in $\overset{\text{ghb}_1}{\rightarrow}$, this cycle is a cycle in $\overset{\text{ghb}_1}{\rightarrow} \cup \overset{\text{ppo}_2 \setminus 1}{\rightarrow} \cup (\overset{\text{grf}_2 \setminus 1}{\rightarrow}; \overset{\text{ppo}_2}{\rightarrow})$. Since $\overset{\text{slf}}{\rightarrow}$ orders all fragile pairs, and is compatible with $\overset{\text{ppo}_2}{\rightarrow}$, we know that $\overset{\text{ppo}_2 \setminus 1}{\rightarrow}$ is included in $\overset{\text{slf}}{\rightarrow}$ and $(\overset{\text{grf}_2 \setminus 1}{\rightarrow}; \overset{\text{ppo}_2}{\rightarrow}) \subseteq (\overset{\text{grf}_2 \setminus 1}{\rightarrow}; \overset{\text{slf}}{\rightarrow})$. Since $\overset{\text{slf}}{\rightarrow}$ is A-cumulative, $(\overset{\text{grf}_2}{\rightarrow}; \overset{\text{slf}}{\rightarrow})$ is in $\overset{\text{ab}_1}{\rightarrow}$, hence in $\overset{\text{ghb}_1}{\rightarrow}$. Thus there is a cycle in $\overset{\text{slf}}{\rightarrow} \cup \overset{\text{ghb}_1}{\rightarrow}$, which contradicts their compatibility. □

Arch.	Fragile pair	Barriers (mapping F)
Power	$r \overset{\text{po}}{\rightarrow} r$	$r \overset{\text{sync}}{\rightarrow} r$ (need A-cumulativity)
	$r \overset{\text{po}}{\rightarrow} w$	$r \overset{\text{lwsync}}{\rightarrow} w$ (A-cumulativity OK)
	$w \overset{\text{po}}{\rightarrow} w$	$w \overset{\text{lwsync}}{\rightarrow} w$ (no need for A-cumulativity)
	$w \overset{\text{po}}{\rightarrow} r$	$w \overset{\text{sync}}{\rightarrow} r$ (need for write-read non-cumulativity)
x86	$w \overset{\text{po}}{\rightarrow} r$	$w \overset{\text{mfence}}{\rightarrow} r$ (need for write-read non-cumulativity)

Fig. 5. Mapping F: barriers

This lemma leads to a mapping which we call F (for fences), given in Fig. 5. This mapping places a barrier between each fragile pair of a program; following Lem. 4, it gives a SC behaviour to this program. Recall that we give the semantics of the barriers that we use in the mapping F in Sec. 1, § *Architectures*, on p. 4.

In x86, stores are atomic, and only the write-read pairs in program order are not preserved, *i.e.* the fragile pairs are the write-read pairs $w \overset{\text{po}}{\rightarrow} r$. We do not need cumulativity in x86, *i.e.* we only need a non-cumulative write-read barrier, the `mfence` barrier: $w \overset{\text{mfence}}{\rightarrow} r$.

In Power, no pair is preserved in program order except the read-read and read-write pairs with a dependency between the accesses [5]. But since stores are not atomic, even the dependent read-read and read-write pairs are fragile. For a read-read pair $r_1 \overset{\text{po}}{\rightarrow} r_2$, since r_1 can read from a non-atomic write w, we need a cumulative barrier between r_1 and r_2. But `lwsync` does not order write to read chains, *i.e.* `lwsync` between r_1 and r_2 will not order w and r_2. Therefore we need a `sync`: $r_1 \overset{\text{sync}}{\rightarrow} r_2$. For a read-write pair $r \overset{\text{po}}{\rightarrow} w$, we need a cumulative barrier as well, but `lwsync` is sufficient here, for it will order the read w' from which r may read and w. In the write-write and write-read cases, there is no need for cumulativity. In the write-write case, a `lwsync` is enough, for it orders write-write pairs; but in the write-read case, we need a `sync`.

3 Synchronisation idioms

As an illustration to Sec. 2, we now study the semantics of Power's locks and rmw [1]. As noted by S. Adve and H.-J. Boehm in [4] *"[o]n hardware that relaxes write atomicity [..., such as Power] even the fully fenced implementation may not be sequentially consistent."* Thus it is unclear whether the synchronisation primitives provided by the architecture actually restore SC: it could perfectly be the architect's intent (*e.g.* `lwsync` is not strong enough to restore SC, but is faster than `sync`, as we show in Sec. 5), or a bug in the implementation [5]. Hence we need to define the semantics of the synchronisation primitives given in the documentation, and study whether they allow us to restore SC, *i.e.* that we can use them to build covering relations, as defined in Sec. 2.

We first define *atomic pairs*, which are the stepping stone to build locks, studied in Sec. 3.1 and rmw, studied in Sec. 3.2. We show how to use these primitives

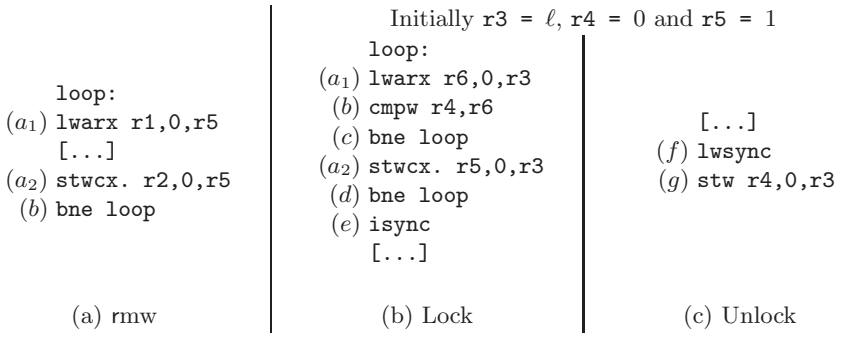


Fig. 6. Read-modify-write, lock and unlock in Power

to build covering relations. Second, because cumulativity might be too costly in practice, or its implementation challenging, we propose in Sec. 3.2 two lock-free mappings restoring a strong architecture from Power without using cumulativity, as an alternative to the mapping F (see Sec. 2) which uses cumulativity.

Atomicity Fig. 6(a) gives a generic Power rmw. The `lwarx` (a_1) loads from its source address (in register `r5`) and *reserves* it. Any subsequent store to the reserved address from another processor and any subsequent `lwarx` from the same processor invalidates the reservation. The `stwcx.` (a_2) checks if the reservation is valid; if so, it is said to be *successful*: it stores into the reserved address and the code exits the loop. Otherwise, the `stwcx.` does not store and the code loops. Thus these instructions ensure *atomicity* to the code they surround (provided this code does not contain any `lwarx` nor `stwcx.`), as no other processor can write to the reserved location between the `lwarx` and the successful `stwcx.`.

We distinguish the read and write events issued by such instructions from the plain ones: we write $\mathbb{R}^*$ (resp. $\mathbb{W}^*$) for the subset of $\mathbb{R}$ (resp. $\mathbb{W}$) issued by a `lwarx` (resp. a successful `stwcx.`), and define two events r and w to form an atomic pair $w.r.t.$ a location ℓ if (a) w was issued by a successful `stwcx.` to ℓ, (b) r was issued by the last `lwarx` (in $\xrightarrow{\text{po}}$) from ℓ before the `stwcx.` that issued w, and (c) no other processor wrote to ℓ between r and w:

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

3.1 Locks

Atomic pairs are used *e.g.* in *lock* and *unlock* primitives [1, App. B]. The idiomatic Power lock (resp. unlock) is shown in Fig. 6(b) (resp. Fig. 6(c)).

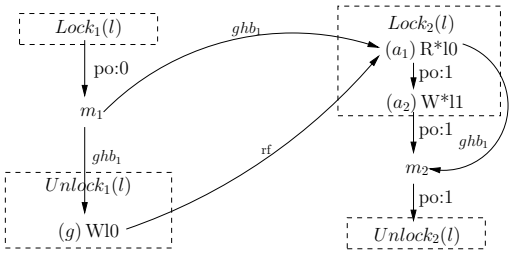


Fig. 7. Opening lock and unlock

Critical sections A lock reads the lock variable ℓ to see if it is free; an unlock writes to ℓ to free it. The instructions between a lock and an unlock form a *critical section*. Thus, a critical section consists of a lock $\mathrm{Lock}(\ell, r)$ and an unlock $\mathrm{Unlock}(\ell, r, w)$ (we define these two predicates in the next §) with the same variable ℓ, and the events in $\overset{\mathrm{po}}{\rightarrow}$ between the lock's read and the unlock's write:

$$\mathrm{cs}(\mathcal{E}, \ell, r, w) \triangleq \mathrm{Lock}(\ell, r) \wedge \mathcal{E} = \{e \mid r \overset{\mathrm{po}}{\rightarrow} e \overset{\mathrm{po}}{\rightarrow} w\} \wedge \mathrm{Unlock}(\ell, r, w)$$

We write loc(cs) for the location of a critical section cs. Two critical sections cs_1 and cs_2 with the same location ℓ are *serialised* if cs_2 reads from cs_1, as depicted in Fig. 7: the left-hand side of the picture is cs_1, composed of a lock $\mathrm{Lock}_1(\ell)$, an event m_1 and an unlock $\mathrm{Unlock}_1(\ell)$, which writes into ℓ *via* the write (g). The second critical section cs_2 is on the right: the read (a_1) of its lock $\mathrm{Lock}_2(\ell)$ reads from (g). Thus, cs_1 and cs_2 are serialised if cs_2 Lock's read (written $\mathrm{R}(\mathrm{cs}_2)$) reads from cs_1 Unlock's write (written $\mathrm{W}(\mathrm{cs}_1)$):

$$\mathrm{cs}_1 \overset{\mathrm{css}_\ell}{\rightarrow} \mathrm{cs}_2 \triangleq \mathrm{loc}(\mathrm{cs}_1) = \mathrm{loc}(\mathrm{cs}_2) = \ell \wedge \mathrm{W}(\mathrm{cs}_1) \overset{\mathrm{rf}}{\rightarrow} \mathrm{R}(\mathrm{cs}_2)$$

Given a location ℓ, two events m_1 and m_2 are in $\overset{\mathrm{lock}_\ell}{\rightarrow}$ if they are in two serialised critical sections (as in Fig. 7), or m_1 is in $\overset{\mathrm{lock}_\ell}{\rightarrow}$ with an event itself in $\overset{\mathrm{lock}_\ell}{\rightarrow}$ with m_2 ($m \in \mathrm{cs}$ ensures m is between cs import and export barriers in $\overset{\mathrm{po}}{\rightarrow}$):

$$m_1 \overset{\mathrm{lock}_\ell}{\rightarrow} m_2 \triangleq (\exists \mathrm{cs}_1 \overset{\mathrm{css}_\ell}{\rightarrow} \mathrm{cs}_2, m_1 \in \mathrm{cs}_1 \wedge m_2 \in \mathrm{cs}_2) \vee (\exists m, m_1 \overset{\mathrm{lock}_\ell}{\rightarrow} m \overset{\mathrm{lock}_\ell}{\rightarrow} m_2)$$

Finally, two events m_1 and m_2 are in $\overset{\mathrm{lock}}{\rightarrow}$ if there exists ℓ such that $m_1 \overset{\mathrm{lock}_\ell}{\rightarrow} m_2$.

Lock and unlock In the Power lock of Fig. 6(b), the lines (a_1) to (a_2) form an atomic pair, as in Fig. 6(a); this sequence loops until it acquires the lock. Here, acquiring the lock means that the `lwarx` read the lock variable ℓ, and that ℓ was later written to by a successful `stwcx.`. Thus, the read r of the `lwarx` takes a lock ℓ if it forms an atomic pair with the write w from the successful `stwcx.`:

$$\mathrm{taken}(\ell, r) \triangleq \exists w, \mathrm{atom}(r, w, \ell)$$

The acquisition is followed by a sequence `bne;isync` (lines (d) and (e)), forming an *import barrier* [1, p. 721]. An import barrier prevents any event to

float above a read issued by a `lwarx`: in Fig. 7, the event m_2 in cs_2 is in $\overset{\text{ghb}_1}{\rightarrow}$ with the read (a_1) from its Lock's `lwarx`. Hence the read r of a lock's `lwarx` satisfies the import predicate when no access m after r can be speculated before r:

$$\text{import}(r) \triangleq \forall rm, (r \in \mathbb{R}^* \wedge r \overset{\text{po}}{\rightarrow} m) \Rightarrow (r \overset{\text{ab}_1}{\rightarrow} m)$$

Fig. 6(c) shows Power's unlock. It starts (line (f)) with an *export barrier* [1, p. 722] (here a `lwsync`). The export barrier forces the accesses before the write w of the unlock to be committed to memory before the next lock primitive takes the lock: in Fig. 7, the event m_1 in cs_1 is in $\overset{\text{ghb}_1}{\rightarrow}$ with the read (a_1) of cs_2's Lock. This means that we define an export barrier to be B-cumulative, but only $w.r.t.$ reads issued by the `lwarx` of an atomic pair:

$$\text{export}(w) \triangleq \forall rm, (r \in \mathbb{R}^* \wedge (m \overset{\text{po}}{\rightarrow} w \overset{\text{rf}}{\rightarrow} r)) \Rightarrow (m \overset{\text{ab}_1}{\rightarrow} r)$$

Then a store to the lock variable (line (g)), or more precisely the next write event to ℓ in program order after a lock acquisition, frees the lock:

$$\begin{aligned}\text{free}(\ell, r, w) \triangleq w \in \mathbb{W} \wedge \text{loc}(w) = \ell \wedge r \overset{\text{po}}{\rightarrow} w \wedge \text{taken}(\ell, r) \wedge \\ \neg(\exists w' \in \mathbb{W}, \text{loc}(w') = \ell \wedge r \overset{\text{po}}{\rightarrow} w' \overset{\text{po}}{\rightarrow} w)\end{aligned}$$

A lock primitive thus consists of a taken operation (see Fig. 6(b), lines (a_1) to (a_2)) followed by an import barrier. An unlock consists of an export barrier (line (f)) followed by a write freeing the lock (line (g)):

$$\text{Lock}(\ell, r) \triangleq \text{taken}(\ell, r) \wedge \text{import}(r)$$

$$\text{Unlock}(\ell, r, w) \triangleq \text{free}(\ell, r, w) \wedge \text{export}(w)$$

We show that this semantics ensures the acyclicity of $\overset{\text{lock}}{\rightarrow} \cup \overset{\text{po}}{\rightarrow}$, *i.e.* following Lem. 3, $(\overset{\text{lock}}{\rightarrow} \cup \overset{\text{po}}{\rightarrow})^+$ is covering for the competing accesses. Hence locks on the competing accesses ensures a SC behaviour to Power programs:

Lem. 5. $\forall EX, A_1 \,.\, \text{valid}(E, X) \Rightarrow \text{acyclic}(\overset{\text{lock}}{\rightarrow} \cup \overset{\text{po}}{\rightarrow})$

Proof Suppose by contradiction a cycle in $\overset{\text{lock}}{\rightarrow} \cup \overset{\text{po}}{\rightarrow}$. This cycle is a cycle in $((\overset{\text{lock}}{\rightarrow})^+; \overset{\text{po}}{\rightarrow})$ since $\overset{\text{po}}{\rightarrow}$ is transitive. Let us show by induction that any path of $(\overset{\text{lock}}{\rightarrow})^+; \overset{\text{po}}{\rightarrow}$ from an event x to an event y is a path in $(\overset{\text{ghb}_1}{\rightarrow})^+$. Hence, a cycle is in $(\overset{\text{lock}}{\rightarrow})^+; \overset{\text{po}}{\rightarrow}$ is a cycle in $(\overset{\text{ghb}_1}{\rightarrow})^+$, which contradicts the validity of (E, X) on A_1.

Consider the base case with three events $m_1 \overset{\text{lock}}{\rightarrow} m_2 \overset{\text{po}}{\rightarrow} m_3$. Let us do an induction over $m_1 \overset{\text{lock}}{\rightarrow} m_2$. Consider the base case where m_1 and m_2 belong respectively to the critical sections cs_1 and cs_2, such that $\text{cs}_1 \overset{\text{css}}{\rightarrow} \text{cs}_2$.

In this case, m_3 is in $\overset{\text{po}}{\rightarrow}$ after cs_2's import barrier, which prevents any event to float above the read issued by cs_2's `lwarx`. Thus we have $\text{R}(\text{cs}_2) \overset{\text{ab}_1}{\rightarrow} m_3$, hence $\text{R}(\text{cs}_2) \overset{\text{ghb}_1}{\rightarrow} m_3$. Moreover, m_1 is in $\overset{\text{po}}{\rightarrow}$ before cs_1's export barrier (*i.e.* $m_1 \overset{\text{ab}_1}{\rightarrow} \text{W}(\text{cs}_1)$) hence $m_1 \overset{\text{ghb}_1}{\rightarrow} \text{W}(\text{cs}_1)$. Since $\text{W}(\text{cs}_1) \overset{\text{rfe}}{\rightarrow} \text{R}(\text{cs}_2)$, by B-cumulativity of cs_1's export barrier, we have $m_1 \overset{\text{ab}_1}{\rightarrow} \text{R}(\text{cs}_2)$, hence $m_1 \overset{\text{ghb}_1}{\rightarrow} \text{R}(\text{cs}_2)$. Thus $m_1 (\overset{\text{ghb}_1}{\rightarrow})^+ m_3$.

The transitive cases follows by induction. □

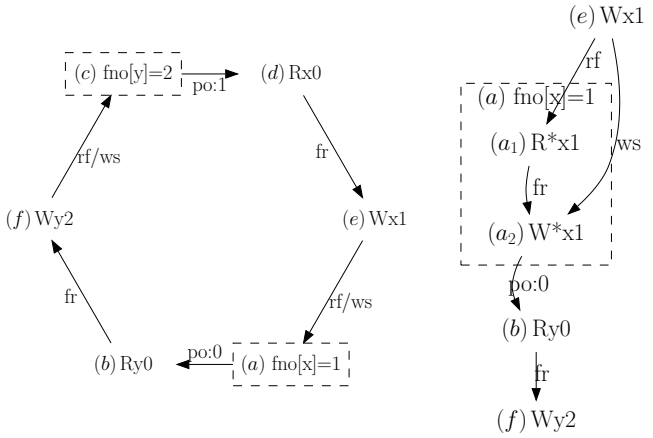


Fig. 8. (a) **iriw** after P mapping **5.**(b) Opening fno on P_0

Our import barrier allows events to be delayed so that they are performed inside the critical section. Our export barrier allows the events after the unlock to be speculated before the lock is released. Such relaxed semantics already exist for high-level lock and unlock primitives [8, 22].

In the documentation [1, p. 721], the import barrier is a sequence `bne;isync` (*i.e.* a read-read, read-write non-cumulative barrier) or a `lwsync`, *i.e.* cumulative [1, p.721]. Lem. 5 shows that the first one is enough, for our import barrier does not need cumulativity. The export barrier is a `sync` (*i.e.* cumulative for all pairs) or a `lwsync` [1, p. 722]. Lem. 5 shows that we only need a B-cumulative barrier towards reads issued by a `lwarx`, *i.e.* a `sync` is unnecessarily costly. Moreover, although a `lwsync` is not B-cumulative towards plain reads, its implementations appear experimentally to treat the reads issued by the `lwarx` of an atomic pair specially. We tested this semantics of `lwsync` with our `diy` tool [5], and ran our automatically generated tests up to 10^9 times each (see the logs online[1]). Hence our semantics for the export barrier is experimentally sound.

3.2 Read-modify-write primitives

By Lem. 4, we can restore SC in the **iriw** test of Fig. 1 using A-cumulative barriers between the fragile pairs (a) and (b) on P_0, and (c) and (d) on P_1. Yet, cumulativity may be challenging to implement or too costly in practice [5]. We propose a mapping of certain reads to rmw (as in Fig. 6(a)), and show that this restores a strong architecture from a weaker one without using cumulativity.

In Fig. 8(a), we replaced the fragile reads (a) and (c) of **iriw** by rmw: we say these fragile reads are *protected* (a notion defined below). In the example we use *fetch and no-op* (fno) primitives [1, p.719] to implement atomic reads. Yet, our results hold for any kind of rmw. We show that when the fragile reads are

protected, we do not need cumulative barriers, but just non-cumulative ones. If a read is protected by a rmw, then the rmw compensates the need for cumulativity by enforcing enough order to the write from which the protected read reads.

Protecting the fragile reads with rmw We consider that two events r and w form a rmw *w.r.t.* a location ℓ if they form an atomic pair *w.r.t.* ℓ (*i.e.* the code in Fig. 6(a) does not loop), or there is a read r' after r in the program order forming an atomic pair *w.r.t.* ℓ with w, such that r' is the last read issued by the loop before the `stwcx.` succeeds (*i.e.* the code in Fig. 6(a) loops). We do not consider the case where the loop never terminates:

$$\mathrm{rmw}(r,w,\ell) \triangleq \mathrm{atom}(r,w,\ell) \vee (\exists r', r \xrightarrow{\mathrm{po}} r' \wedge \mathrm{loc}(r) = \mathrm{loc}(r') \wedge \mathrm{atom}(r',w,\ell))$$

In Fig. 8(b), we open up the fno box protecting the read (a) from x on P_0. We suppose that the fno is immediately successful, *i.e.* the code in Fig. 6(a) does not loop. Hence we expand the fno event (a) on P_0 to the r^* (a_1) (from the `lwarx`) in program order with the w^* (a_2) (from the successful `stwcx.`).

We define a read to be *protected* when it is issued by the `lwarx` of a rmw immediately followed in program order by a non-cumulative barrier; an execution (E, X) is protected when its fragile reads are:

$$\mathrm{protected}(r) \triangleq \exists w, \mathrm{rmw}(r,w,\mathrm{loc}(r)) \wedge (\forall m, w \xrightarrow{\mathrm{po}} m \Rightarrow w \xrightarrow{\mathrm{ab}_1} m)$$
$$\mathrm{protected}(E,X) \triangleq \forall r, \mathrm{fragile}(r) \Rightarrow \mathrm{protected}(r)$$

In Fig. 8(b), the write (e) from which (a_1) reads hits the memory before (a_2), *i.e.* $(e) \xrightarrow{\mathrm{ws}} (a_2)$. Hence there are two paths from (e) to (b): $(e) \xrightarrow{\mathrm{rf}} (a_1) \xrightarrow{\mathrm{po}} (b)$ and $(e) \xrightarrow{\mathrm{ws}} (a_2) \xrightarrow{\mathrm{po}} (b)$. Thus we can trade the fragile pair $(a_1), (b)$ for the pair $(a_2), (b)$: we compensate the lack of write atomicity of (e) (*i.e.* $(e) \xrightarrow{\mathrm{rfe}} (a)$ not global) by using the write serialisation between (e) and (a_2) (thanks to the rmw) instead of cumulativity before. Formally, we prove that a sequence $w \xrightarrow{\mathrm{grf}_2 \setminus 1} r \xrightarrow{\mathrm{ppo}_2} m$ with r protected is in $\xrightarrow{\mathrm{ws}}; \xrightarrow{\mathrm{ghb}_1}$, *i.e.* globally ordered on A_1:

Lem. 6. $\forall wrm, (\mathrm{protected}(r) \wedge w \xrightarrow{\mathrm{grf}_2 \setminus 1} r \xrightarrow{\mathrm{ppo}_2} m) \Rightarrow w \xrightarrow{\mathrm{ws}}; \xrightarrow{\mathrm{ghb}_1} m$

Proof Since r is protected, there are r' and w' such that $\mathrm{rmw}(r', w', \mathrm{loc}(r))$ where r' is r or a subsequent read in $\xrightarrow{\mathrm{po}}$. In both cases, we have $w \xrightarrow{\mathrm{ws}} w'$. Moreover, since there is a barrier between w' and m (*i.e.* $w' \xrightarrow{\mathrm{ab}_1} m$), we know that $w' \xrightarrow{\mathrm{ghb}_1} m$. □

Thus, if we protect the fragile reads, the only remaining fragile pairs are the ones in $\xrightarrow{\mathrm{ppo}_2 \setminus 1}$. In Fig. 8(a), we have $(e) \xrightarrow{\mathrm{ws}} (a_2) \xrightarrow{\mathrm{po}} (b) \xrightarrow{\mathrm{fr}} (f)$ and $(f) \xrightarrow{\mathrm{ws}} (c_2) \xrightarrow{\mathrm{po}} (d) \xrightarrow{\mathrm{fr}} (e)$, hence a cycle in $\xrightarrow{\mathrm{ws}} \cup \xrightarrow{\mathrm{fr}} \cup \xrightarrow{\mathrm{po}}$. Since $\xrightarrow{\mathrm{ws}}$ and $\xrightarrow{\mathrm{fr}}$ are global, to invalidate this cycle, we need to order globally (*e.g.* by a barrier) the accesses (a_2) and (b) on P_0 and (c_2) and (d) on P_1. Indeed, if an execution is protected, non-cumulative barriers placed between the remaining fragile pairs in $\xrightarrow{\mathrm{ppo}_2 \setminus 1}$ ensure stability:

Lem. 7. $A_1 . \mathrm{valid}(E,X) \wedge \mathrm{protected}(E,X) \wedge (\xrightarrow{\mathrm{ppo}_2 \setminus 1} \subseteq \xrightarrow{\mathrm{ab}_1}) \Rightarrow A_2 . \mathrm{valid}(E,X)$

Arch.	Fragile pair	rmw (mapping A)	rmw (mapping P)
Power	$r \overset{\text{po}}{\rightarrow} r$	$\texttt{fno} \overset{\text{po}}{\rightarrow} \texttt{fno}$	$\texttt{fno} \overset{\text{sync}}{\rightarrow} r$
	$r \overset{\text{po}}{\rightarrow} w$	$\texttt{fno} \overset{\text{po}}{\rightarrow} \texttt{sta}$	$\texttt{fno} \overset{\text{lwsync}}{\rightarrow} w$
	$w \overset{\text{po}}{\rightarrow} w$	$\texttt{sta} \overset{\text{po}}{\rightarrow} \texttt{sta}$	$w \overset{\text{lwsync}}{\rightarrow} w$
	$w \overset{\text{po}}{\rightarrow} r$	$\texttt{sta} \overset{\text{po}}{\rightarrow} \texttt{fno}$	$w \overset{\text{sync}}{\rightarrow} r$
x86	$w \overset{\text{po}}{\rightarrow} r$	$\texttt{xchg} \overset{\text{po}}{\rightarrow} r$	na

Fig. 9. Mappings A and P: rmw

Proof Barriers are by definition compatible with $\overset{\text{ghb}_1}{\rightarrow}$. The A-cumulativity is handled by the protection of the fragile reads as shown in Lem. 6. Finally, the barriers order globally the remaining fragile pairs. Hence the barriers induce a covering synchronisation relation by Lem. 4. □

This lemma leads to a mapping which we call P (for protected reads), given in Fig. 9. This mapping places a fno on the first read of a fragile pair, and a barrier between this fno and the second access of the fragile pair of a given piece of code. If the first access of the fragile pair is a write, it remains unchanged and we only place a barrier between the two accesses, following the mapping F. For the read-read (resp. read-write) case, since replacing a read by a fno amounts to replacing the read by a sequence of events ending with a write, we choose a barrier ordering write-read (resp. write-write) pairs, *i.e.* Power `sync` (resp. `lwsync`). Following Lem. 7, the mapping P gives a SC behaviour to a program.

H.-J. Boehm and S. Adve propose in [10] a mapping of all stores into rmw (*i.e.* `xchg`) on x86 (which has no fragile reads), to provide a SC semantics to C++ atomics. We call this mapping A-x86 (for atomics), and give it in Fig. 9. For models with fragile reads, *e.g.* Power, they question in [4] the existence of *"more efficient mappings (than the use of locks)"*. The mapping P could be more efficient, since it removes the need for cumulativity. Yet, mapping reads to rmw introduces additional stores (issued by `stwcx.`), which may impair the performance. Moreover, we have to use cumulative barriers in the mapping P, for Power does not provide non-cumulative barriers. Yet, we show in Sec. 5 that the mapping P is more efficient than locks on Power machines.

We propose another mapping, given in Fig. 9, which we call A-Power. All reads and writes are mapped into rmw (using fno for reads and fetch-and-store (sta) [1, p. 719] for writes). The documentation stipulates indeed that *"a processor has at most one reservation at any time"* [1, p. 663]. Hence two rmw on the same processor in program order may be preserved in this order, because the writes issued by their `stwcx.`, though to different locations, would be ordered by a dependency over the reservation. Although the documentation does not state whether this dependency even exists, we show (see Sec. 5) that the mapping A-Power restores SC experimentally and is more efficient than locks as well.

4 Stability from a weak architecture to SC

We now want to minimise the synchronisation that we use, *i.e.* we would like to synchronise only the conflicting accesses (either competing accesses or fragile pairs) that are actually necessary. For example, if in the **iriw** test of Fig. 1, we add a write (g) to a fresh variable z after (in program order) the write (e) to x on P_2, (e) and (g) may not be preserved in program order, *i.e.* (e) and (g) may form a fragile pair. Yet, there is no need to maintain them, since they do not contribute to the cycle we want to forbid.

Critical cycles D. Shasha and M. Snir provide in [23] an analysis to place barriers in a program, in order to enforce a SC behaviour. They examine in [23, Thm. 3.9 p. 297] the *critical cycles* of an execution, and show that placing a barrier along each program order arrow of such a cycle (each *delay* arrow) is enough to restore SC. Yet, this work does not provide any semantics of weak memory models. We show in Coq that their technique applies to the models embraced by our framework, *e.g.* models with store buffering, like TSO or relaxing store atomicity, like Power.

Given an event structure E, a cycle $\xrightarrow{\sigma} \subseteq (\overset{\text{cmp}}{\leftrightarrow} \cup \overset{\text{po}}{\rightarrow})^+$ (where $\overset{\text{cmp}}{\leftrightarrow}$ is the competing relation of Sec. 2) is critical, written $\text{critical}(E, \xrightarrow{\sigma})$, when it satisfies the two following properties: **(i)** Per processor, there are at most two memory accesses (x, y) on this processor, such that $x \overset{\text{po}}{\rightarrow} y$ and $\text{loc}(x) \neq \text{loc}(y)$. **(ii)** For a given memory location x, there are at most three accesses relative to x, and these accesses are from distinct processors ($w \overset{\text{cmp}}{\leftrightarrow} w$, $w \overset{\text{cmp}}{\leftrightarrow} r$, $r \overset{\text{cmp}}{\leftrightarrow} w$ or $r \overset{\text{cmp}}{\leftrightarrow} w \overset{\text{cmp}}{\leftrightarrow} r$). For example, the execution of **iriw** in Fig. 1 has a critical cycle.

Note that a critical cycle does not forbid an execution on SC: the execution of **iriw** in Fig. 10(a) exhibits a critical cycle, *i.e.* $(a) \overset{\text{po}}{\rightarrow} (b) \overset{\text{cmp}}{\leftrightarrow} (f) \overset{\text{cmp}}{\leftrightarrow} (c) \overset{\text{po}}{\rightarrow} (d) \overset{\text{cmp}}{\leftrightarrow} (e) \overset{\text{cmp}}{\leftrightarrow} (a)$, but is authorised on SC (since there is no cycle in $\overset{\text{com}}{\rightarrow} \cup \overset{\text{po}}{\rightarrow}$). Yet, a critical cycle does not require any information about the execution witness. A critical cycle becomes a *violation* of SC *w.r.t.* A (*w.r.t.* an execution witness X, written $\text{viol}_{A,SC}(E, X, \xrightarrow{\sigma})$) when it is *oriented*, *i.e.* when it is included in $\overset{\text{com}}{\rightarrow} \cup \overset{\text{po}}{\rightarrow}$ (but not in $\overset{\text{ghb}_A}{\rightarrow}$, otherwise it would violate A). For example, the execution of **iriw** in Fig. 1 exhibits a violation of SC *w.r.t.* Power: $(a) \overset{\text{po}}{\rightarrow} (b) \overset{\text{fre}}{\rightarrow} (f) \overset{\text{rfe}}{\rightarrow} (c) \overset{\text{po}}{\rightarrow} (d) \overset{\text{fre}}{\rightarrow} (e) \overset{\text{po}}{\rightarrow} (a)$.

Covering the critical cycles We show in the following that, given a conflict relation $\overset{\text{c}}{\rightarrow}$ and a synchronisation relation $\overset{\text{s}}{\rightarrow}$, it is enough to synchronise by $\overset{\text{s}}{\rightarrow}$ the conflicting accesses of $\overset{\text{c}}{\rightarrow}$ that belong to a critical cycle.

We call these conflicts *critical conflicts*. Given an event structure E, we write $\overset{\text{c}_{\text{min}}}{\rightarrow}$ for the critical conflicts of E, *i.e.* $\overset{\text{c}}{\rightarrow}$ restricted to the critical cycles of E:

Definition 1 (Critical conflicts).

$$m_1 \overset{\text{c}_{\text{min}}}{\rightarrow} m_2 \triangleq m_1 \overset{\text{c}}{\rightarrow} m_2 \wedge (\exists \xrightarrow{\sigma}, \text{mv}(E, \xrightarrow{\sigma}) \wedge m_1 \xrightarrow{\sigma} m_2)$$

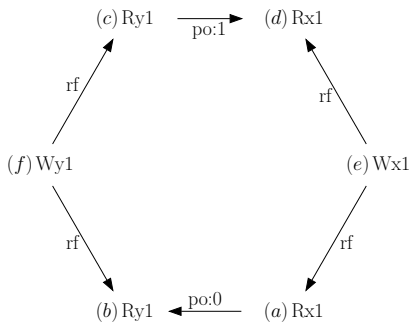


Fig. 10. A SC execution of **iriw**

We show that if all the critical conflicts of an execution (E, X) valid on A are covered, there cannot be any violations in (E, X). A violation is what makes an execution invalid on SC. Hence forbidding them ensures that (E, X) is valid on SC as well:

Lem. 8 (Covering critical conflicts).

$$\forall \overset{\mathrm{c}}{\rightarrow} \overset{\mathrm{s}}{\rightarrow}, \left((\overset{\mathrm{c}}{\rightarrow}, \overset{\mathrm{s}}{\rightarrow}) = (\overset{\mathrm{c_{drf}}}{\leftrightarrow}, \overset{\mathrm{s_{drf}}}{\rightarrow}) \vee (\overset{\mathrm{c}}{\rightarrow}, \overset{\mathrm{s}}{\rightarrow}) = (\overset{\mathrm{c_{lf}}}{\rightarrow}, \overset{\mathrm{s_{lf}}}{\rightarrow}) \right) \Rightarrow \mathrm{covering}(\overset{\mathrm{c_{min}}}{\rightarrow}, \overset{\mathrm{s}}{\rightarrow})$$

Proof Let (E, X) be an execution valid on A where $\overset{\mathrm{c_{min}}}{\rightarrow}$ is arbitrated by $\overset{\mathrm{s}}{\rightarrow}$. Consider by contradiction a cycle in $\mathrm{SC.ghb}(E, X)$. Hence, there exists a violation $\overset{\sigma}{\rightarrow}$ in (E, X).

We know that $\overset{\sigma}{\rightarrow}$ is a cycle in $(\overset{\mathrm{oc_{drf}}}{\rightarrow} \cup \overset{\mathrm{po}}{\rightarrow})^+$. Therefore, we know that $\overset{\sigma}{\rightarrow}$ is equal to $(\overset{\sigma}{\rightarrow} \cap \overset{\mathrm{oc_{drf}}}{\rightarrow}) \cup (\overset{\sigma}{\rightarrow} \cap \overset{\mathrm{po}}{\rightarrow})$. By hypothesis, since $\overset{\sigma}{\rightarrow}$ is a violation, all the conflicts in $\overset{\sigma}{\rightarrow}$ are arbitrated by $\overset{\mathrm{s}}{\rightarrow}$.

- Suppose $(\overset{\mathrm{c}}{\rightarrow}, \overset{\mathrm{s}}{\rightarrow}) = (\overset{\mathrm{c_{drf}}}{\leftrightarrow}, \overset{\mathrm{s_{drf}}}{\rightarrow})$. Since $\overset{\mathrm{s_{drf}}}{\rightarrow}$ is compatible with $\overset{\mathrm{com}}{\rightarrow}$, all the pairs in $\overset{\mathrm{oc_{drf}}}{\rightarrow}$ are in $\overset{\mathrm{s_{drf}}}{\rightarrow}$. Hence $\overset{\sigma}{\rightarrow} = (\overset{\sigma}{\rightarrow} \cap \overset{\mathrm{s_{drf}}}{\rightarrow}) \cup (\overset{\sigma}{\rightarrow} \cap \overset{\mathrm{po}}{\rightarrow})$. Thus, we have $\overset{\sigma}{\rightarrow} \subseteq (\overset{\mathrm{s_{drf}}}{\rightarrow} \cup \overset{\mathrm{po}}{\rightarrow})^+$. Since $\overset{\mathrm{s_{drf}}}{\rightarrow}$ is compatible with $\overset{\mathrm{po}}{\rightarrow}$, there cannot be any such cycle $\overset{\sigma}{\rightarrow}$.
- Suppose $(\overset{\mathrm{c}}{\rightarrow}, \overset{\mathrm{s}}{\rightarrow}) = (\overset{\mathrm{c_{lf}}}{\rightarrow}, \overset{\mathrm{s_{lf}}}{\rightarrow})$. Since $\overset{\sigma}{\rightarrow}$ is a cycle in $(\overset{\mathrm{oc_{drf}}}{\rightarrow} \cup \overset{\mathrm{po}}{\rightarrow})^+$, we know that $\overset{\sigma}{\rightarrow}$ is included in $(\overset{\mathrm{ws}}{\rightarrow} \cup \overset{\mathrm{fr}}{\rightarrow} \cup \overset{\mathrm{rf}}{\rightarrow} \cup \overset{\mathrm{po}}{\rightarrow})^+$. Since $\overset{\mathrm{ws}}{\rightarrow}$ and $\overset{\mathrm{fr}}{\rightarrow}$ are in $\overset{\mathrm{ghb_1}}{\rightarrow}$, $\overset{\sigma}{\rightarrow}$ is included in $(\overset{\mathrm{ghb_1}}{\rightarrow} \cup \overset{\mathrm{ppo_2 \setminus 1}}{\rightarrow} \cup (\overset{\mathrm{grf_2 \setminus 1}}{\rightarrow}; \overset{\mathrm{po}}{\rightarrow}))^+$, *i.e.* in $(\overset{\mathrm{ghb_1}}{\rightarrow} \cup \overset{\mathrm{c_{lf}}}{\rightarrow})^+$. Since $\overset{\mathrm{s_{lf}}}{\rightarrow}$ covers $\overset{\mathrm{c_{lf}}}{\rightarrow}$ and is compatible with $\overset{\mathrm{rf}}{\rightarrow}$ and $\overset{\mathrm{po}}{\rightarrow}$, $\overset{\sigma}{\rightarrow}$ is included in $(\overset{\mathrm{ghb_1}}{\rightarrow} \cup \overset{\mathrm{s_{lf}}}{\rightarrow})^+$. Since $\overset{\mathrm{s_{lf}}}{\rightarrow}$ is compatible with $\overset{\mathrm{ghb_1}}{\rightarrow}$, there cannot be any such cycle $\overset{\sigma}{\rightarrow}$. □

Yet, even if a critical cycles is not necessarily an actual violation of SC, we can build an execution Y (by transforming X) associated with the same event structure, which contains an actual violation of SC. In Fig. 10(a), if the $\overset{\mathrm{rf}}{\rightarrow}$

relations $(f) \xrightarrow{\text{rf}} (b)$ and $(e) \xrightarrow{\text{rf}} (d)$ become $\xrightarrow{\text{fr}}$ ones (*i.e.* $(b) \xrightarrow{\text{fr}} (f)$ and $(d) \xrightarrow{\text{fr}} (e)$), then we build the execution of **iriw** depicted in Fig. 1, which is forbidden on SC. Hence we build an execution violating SC (the one in Fig. 1), from an execution which does not violate SC, but exhibits a critical cycle (the one in Fig. 10(a)). We formalise this idea in the following lemma:

Lem. 9. $\forall E \xrightarrow{\sigma}, \text{critical}(E, \xrightarrow{\sigma}) \Rightarrow (\exists X, A\,.\,\text{valid}(E, X) \wedge \text{viol}_{A,SC}(E, X, \xrightarrow{\sigma}))$

Proof Let (E, X) be an execution valid on A. Let $\xrightarrow{\sigma}$ be a critical cycle in E. By definition of critical cycle, we know that $\text{acyclic}((\xrightarrow{\sigma} \cap \overset{\text{c}_{\text{drf}}}{\leftrightarrow}) \cup \overset{\text{ppo}_A}{\rightarrow})$. Hence we know by Lem. 2 that we can build an execution Y associated with E and valid on A from any linear extension $\xrightarrow{\text{le}}$ of the order $((\xrightarrow{\sigma} \cap \overset{\text{c}_{\text{drf}}}{\leftrightarrow}) \cup \overset{\text{ppo}_A}{\rightarrow})^+$.

Let us show that $\xrightarrow{\sigma}$ is a violation of SC *w.r.t.* A in (E, Y). Since $\xrightarrow{\sigma}$ is a criticall cycle in E, we know it is a critical cycle in (E, Y). Hence, we need to show that $\xrightarrow{\sigma}$ is included in $(\overset{\text{oc}_{\text{drf}}}{\rightarrow} \cup \xrightarrow{\text{po}})^+$ in (E, Y). Thus, for all x and y such that $x \xrightarrow{\sigma} y$, we need to show that $(x, y) \in (\xrightarrow{\text{r}})^+$, where:

$$m_1 \xrightarrow{\text{r}} m_2 \triangleq ((m_1, m_2) \in \text{com}(Y) \wedge (\text{proc}(m_1) \neq \text{proc}(m_2))) \vee (m_1 \xrightarrow{\text{po}} m_2)$$

Consider two events $x \xrightarrow{\sigma} y$. Since $\xrightarrow{\sigma}$ is a critical cycle, we have $(x, y) \in (\overset{\text{c}_{\text{drf}}}{\leftrightarrow} \cup \xrightarrow{\text{po}})^+$. Let us reason by induction over this statement. In the base case, let us do a case disjunction.

- When $(x, y) \in \overset{\text{c}_{\text{drf}}}{\leftrightarrow}$, we do case disjunction over the directions of x and y. Since they are in $\overset{\text{c}_{\text{drf}}}{\leftrightarrow}$, we know that they cannot be both reads.
  - If x and y are both writes, we know since they are in $\overset{\text{c}_{\text{drf}}}{\leftrightarrow}$ that they are to the same location and from distinct processors. Moreover, we know that $x \xrightarrow{\sigma} y$. Therefore by definition of extracted write serialisation (see Lem. 2), they are in $\text{ws}(\xrightarrow{\text{le}})$, *i.e.* in $\text{ws}(Y)$.
  - If x is a read and y a write, we know since (E, Y) is valid on A that there exists a write w_x such that $(w_x, x) \in \text{rf}(Y)$. Hence, by definition of $\text{rf}(Y)$, we know that $w_x \xrightarrow{\text{le}} x$. Moreover, we know that $(x, y) \in (\xrightarrow{\sigma} \cap \overset{\text{c}_{\text{drf}}}{\leftrightarrow})$ by hypothesis, hence $x \xrightarrow{\text{le}} y$. Thus by transitivity we have $w_x \xrightarrow{\text{le}} y$. Since w_x and y are both writes, and to the same location, we know by definition that $(w_x, y) \in \text{ws}(Y)$. Hence, since $(w_x, x) \in \text{rf}(Y)$ and $(w_x, y) \in \text{ws}(Y)$, we have $(x, y) \in \text{fr}(Y)$.
  - If x is a write and y a read, we know since (E, Y) is valid on A that there exists a write w_y such that $(w_y, y) \in \text{rf}(Y)$. Suppose $x = w_y$; in this case, we have $(x, y) \in \text{rf}(Y)$, hence the result. Suppose $x \neq w_y$. In this case, since x and w_y are both writes to the same location, we have $(x, w_y) \in \text{ws}(Y) \vee (w_y, x) \in \text{ws}(Y)$.
    - When $(x, w_y) \in \text{ws}(Y)$, suppose $\text{proc}(x) = \text{proc}(w_y)$. In this case, we know that $x \xrightarrow{\text{po}} w_y$ (since in $\xrightarrow{\text{ws}}$ and from the same processor). Otherwise (*i.e.* if $\text{proc}(x) \neq \text{proc}(w_y)$), (x, w_y) is in $\text{ws}(Y)$ and the two events are from distinct processors. The same reasoning applies

to (w_y, y): we have either $w_y \overset{\text{po}}{\rightarrow} y$ if they are on the same processor, or $(w_y, y) \in \text{rf}(Y)$ if not.
  * When $(w_y, x) \in \text{ws}(Y)$, since $(w_y, y) \in \text{rf}(Y)$, we know that w_y is the maximal previous write to $\text{loc}(y)$ before y in $\overset{\text{le}}{\rightarrow}$. Since $(x, y) \in (\overset{\sigma}{\rightarrow} \cap \overset{\text{c}_{\text{drf}}}{\leftrightarrow})$ by hypothesis, we have $x \overset{\text{le}}{\rightarrow} y$. Since $(w_y, x) \in \text{ws}(Y)$, we have $w_y \overset{\text{le}}{\rightarrow} x$. Hence x occurs in between w_y and y in $\overset{\text{le}}{\rightarrow}$, a contradiction.
- When $x \overset{\text{po}}{\rightarrow} y$, we trivially have $(x, y) \in ((\overset{\sigma}{\rightarrow} \cap \overset{\text{oc}_{\text{drf}}}{\rightarrow}) \cup \overset{\text{po}}{\rightarrow})^+$.

The transitive case follows by immediate induction. □

In our framework, we show that the execution witnesses X of an event structure E are stable from A to SC if and only if there is no critical cycle in E, *i.e.* that an execution valid on A is SC if and only if there is no critical cycle in E:

Thm. 1. $\forall E, (\forall X, \text{stable}_{\text{A,SC}}(E, X)) \Leftrightarrow \neg(\exists \overset{\sigma}{\rightarrow}, \text{critical}(E, \overset{\sigma}{\rightarrow}))$

Proof Let E be an event structure.

$\Rightarrow$ Let X be an associated execution witness; (E, X) is stable by hypothesis. Suppose by contradiction the existence of a critical cycle $\overset{\sigma}{\rightarrow}$ in E. In this case, by Lem. 9, we know that there exists another execution witness Y, such that (E, Y) is valid on A, in which $\overset{\sigma}{\rightarrow}$ is a violation of SC *w.r.t.* A. Since all the executions associated with E are stable, (E, Y) is stable. Since (E, Y) is valid on A and stable, we know by definition of stable that (E, Y) is valid on SC. But since (E, Y) contains a violation of SC *w.r.t.* A, (E, Y) cannot be valid on SC.

$\Leftarrow$ Suppose $(\overset{\text{c}}{\rightarrow}, \overset{\text{s}}{\rightarrow}) = (\overset{\text{c}_{\text{drf}}}{\leftrightarrow}, \overset{\text{s}_{\text{drf}}}{\rightarrow})$ or $(\overset{\text{c}}{\rightarrow}, \overset{\text{s}}{\rightarrow}) = (\overset{\text{c}_{\text{lf}}}{\leftrightarrow}, \overset{\text{s}_{\text{lf}}}{\rightarrow})$. Let (E, X) be an associated execution valid on A. We know that there is always such a X since any execution valid on SC is valid on a weaker architecture by Thm. 1: take for instance any linear extension of $\overset{\text{po}}{\rightarrow}$. Suppose there is no critical ctcle in E. Thus, $\overset{\text{c}_{\text{min}}}{\rightarrow}$ is empty, hence we know that $\overset{\text{c}_{\text{min}}}{\rightarrow}$ is trivially covered by $\overset{\text{s}}{\rightarrow}$ in (E, X). Moreover, $\overset{\text{s}}{\rightarrow}$ is covering for $\overset{\text{c}_{\text{min}}}{\rightarrow}$ by Lem. 8. Therefore, we know by definition of covering that (E, X) is valid on SC. □

This theorem means that it is enough to arbitrate (with a covering relation) the conflicting accesses (either competing accesses or fragile pairs) occurring in the critical cycles of a given event structure to give it a SC behaviour. Hence we do not have to synchronise all the conflicts to ensure stability from a weak architecture to SC, but only those occurring in critical cycles.

5 Offence: a synchronisation tool

We implemented our study in our new offence tool, illustrating techniques that can be included in a compiler. Given a program in x86 or Power assembly, offence places either lock-based or lock-free synchronisation along the critical cycles of its input, following the mapping A, P, L or F, to enforce a SC behaviour.

5.1 Control flow graphs and critical cycles

Offence builds one control flow graph (cfg) per thread of the input program, containing *static events* (*i.e.* nodes representing memory accesses), and control flow instructions. A static memory event f has a direction, a location, originating instruction and processor, as events do, but no value component.

Given an event structure and two events $e_1 \overset{\text{po}}{\rightarrow} e_2$, mapping to static events f_1 and f_2, we compute the *static program order* relation $\overset{\text{pos}}{\rightarrow}$, such that $e_1 \overset{\text{po}}{\rightarrow} e_2$ entails $f_1 \overset{\text{pos}}{\rightarrow} f_2$, using a standard forward data flow analysis. If memory locations accessed by a given instruction are constant, we have $\text{loc}(e_1) = \text{loc}(f_1)$ and $\text{loc}(e_2) = \text{loc}(f_2)$. Hence static conflicts computed from the cfg, written $\overset{\text{cmps}}{\leftrightarrow}$, abstract the conflicts of the event structures. When locations are not constant, we need alias analysis to compute an over-approximation of the locations of each static event. One can safely consider that all pairs of memory accesses by distinct processors conflict, as soon as one of the accesses is a write.

With F the set of static events, we call the triple $(F, \overset{\text{pos}}{\rightarrow}, \overset{\text{cmps}}{\leftrightarrow})$ *static event structure*. Following Sec. 4, we enumerate the cycles of F that have properties **(i)** and **(ii)**, *i.e.* we build an over-approximation of the runtime critical cycles.

5.2 Placing synchronisation primitives

We then collect the fragile pairs (*i.e.* the write-read pairs in x86 and all pairs in Power) occurring in the critical cycles of F. By Thm. 1 it is necessary and sufficient to maintain these fragile pairs to reach stability, *i.e.* to restore SC.

Barriers Then, offence follows the mapping F on these fragile pairs. Given a pair (f_1, f_2), offence issues the barrier request (i_1, i_2, b) where $i_1 = \text{ins}(f_1)$, $i_2 = \text{ins}(f_2)$ and b is the required barrier. Every path from i_1 to i_2 in the cfg should pass through a barrier instruction b. We use the global barrier placement of [18], which maximises the number of pairs maintained by a given barrier.

Alternative to barriers Offence can also follow the mappings A and P. For A-x86, the `xchg` instruction has an implicit write-read barrier semantics [10]. Thus, we use the global barrier placement of [18] for `xchg`. For locks, offence follows the mapping L on the conflict edges of the cfg. Sec. 3.1 describes the lock and unlock idioms that we use for Power. For x86, lock uses the `xchg` instruction to build a compare-and-swap loop, while unlock uses a single store instruction.

5.3 Experiments

Generating tests We generated two kinds of tests to exercise offence, using our previous diy tool [5], which computes tests in x86 or Power assembly from a given cycle of relations. First, we generate tests built from critical cycles, *e.g.* **iriw** in Fig. 1. Second, using a new tool, we mix such tests: given two tests built from critical cycles, we randomly permute processors of one of the given tests, alpha-convert its memory locations and registers to fresh ones, and interleave the codes of the programs. The process yields two series of tests, written X in the following, each series consisting of 209 tests for Power and 58 tests for x86.

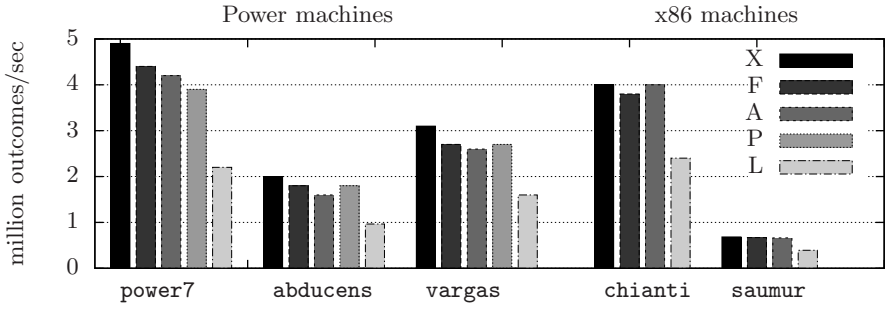


Fig. 11. Productivity observed during soundness experiments.

Experimental soundness We run these tests against hardware using our litmus tool [6]. We observed that all tests from the initial X series exhibit violations of SC and that the tests transformed by offence (following the mappings F, A, P and L) do *not* exhibit violations of SC, running each test at least 10^9 times. Thus we confirmed experimentally that our mappings enforce SC, which we established formally for the mappings F (Lem. 4), P (Lem. 7) and L (Lem. 3 and 5).

Measures of synchronisation cost We also estimate the cost of synchronisation constructs. We show in Fig. 11 the *productivity*, *i.e.* the number of outcomes collected per second, for the initial series of tests X, and for the tests transformed by offence following the mappings F, A, P and L. We ran tests on three Power machines: `power7` (Power7, 8 cores 4-ways SMT), `abducens` (Power6, 4 cores 2-ways SMT) and `vargas` (Power6, 32 cores 2-ways SMT); and on two AMD64 machines `chianti` (Intel Xeon, 8 cores, 2-ways HT) and `saumur` (Intel Xeon, 4 cores, 2-ways HT). Our mappings F, P and A outperform the L one, *i.e.* provide *"more efficient mappings (than the use of locks)"*, answering the question of [4].

To compare the barriers and rmw more precisely, we consider 8 specific tests from 1 to 8 threads, where we add synchronisation with offence so that there is only one synchronisation primitive per thread, and insert the code for each thread inside a tight loop. We then measure running times on our two 8 core machines, `power7` and `chianti`, substract the time of the original test from the time of synchronised tests and divide the result by loop size. We give the results in Fig. 12. While fences and rmw are fast in isolation (10–20 ns on one thread), their cost raises to hundreds of ns when communication by shared memory occurs.

6 Related Work and Conclusion

Related work The DRF guarantee [3, 10], the semantics of synchronisation idioms [9, 8], and the insertion of barriers [23, 14, 11, 16] have been extensively studied,

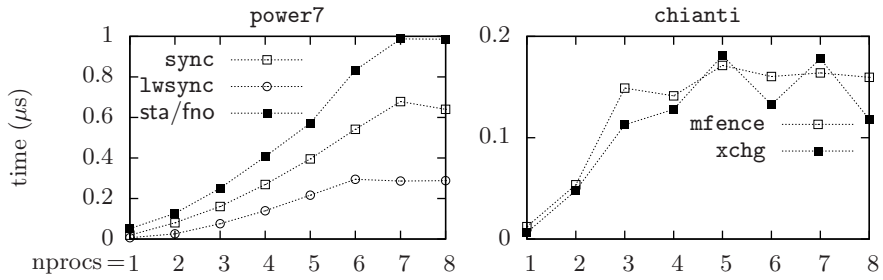


Fig. 12. Time of synchronisation constructs, in microseconds.

but most of these works focus on one kind of synchronisation, and none of them addresses Power features such as *cumulativity* or the lack of write atomicity.

S. Burckhardt and M. Musuvathi examine in [12] whether we can simulate a program running on TSO by enumerating only its SC executions. They distinguish a class of such executions, the *TSO-safe* ones. We believe these executions to be an instance of our stable ones, *i.e.* the stable executions from TSO to SC. Yet, our characterisation of stability in the general case is a novel contribution.

J. Lee and D. Padua examine in [18] the issue of restoring SC at the compiler level. We have re-used their global fence placement algorithm. Yet, our work improves on [18] *w.r.t.* semantical fundations: as a result, we use Power `lwsync` when possible and we do not use the x86 barriers `lfence` and `sfence` that are irrelevant to user-level code. We believe that our mappings could be included in their Java compiler [25], *i.e.* using `lwsync` for Power, and `xchg` for x86.

Conclusion We propose a formal study of stability in weak memory models. This allows us to define several mappings of Power or x86 assembly code, which, as we prove in Coq, give a SC behaviour to a given program. Along the way, we give a semantics to Power's `lwarx` and `stwcx.` instructions and show how to use the lightweight Power barrier `lwsync`, which are novel contributions. In addition, we characterise the executions stable from a weak architecture to SC, hence generalise the result of [23] to weak memory models. Finally, we implement our study in our offence tool, to measure the cost of these mappings: our lock-free mappings outperform locks. Our work could benefit to compiler writers, for we propose several mappings ensuring that an assembly program is SC: for example, the problem of ensuring a SC semantics to C++ atomics is of great interest. Moreover, our work could benefit to the verification community because it allows to separate the verification of a program running on a weak architecture into two steps: first, check the stability of a program from a weak architecture to SC; second, apply SC-sound verification techniques on this program.

References

1. *Power ISA Version 2.06*. 2009.
2. S. V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, 29:66–76, 1995.
3. S. V. Adve and M. D. Hill. Weak Ordering - A New Definition. In *ISCA 1990*.
4. S.V. Adve and H.-J. Boehm. Memory Models: A Case for Rethinking Parallel Languages and Hardware. To appear in CACM.
5. J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Fences in Weak Memory Models. In *CAV 2010*.
6. J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Litmus: Running Tests Against Hardware. In *TACAS 2011*.
7. Alpha Architecture Reference Manual, Fourth Edition, 2002.
8. H.-J. Boehm. Reordering Constraints for Pthread-Style Locks. In *PPoPP 2007*.
9. H.-J. Boehm. Threads Cannot Be Implemented As a Library. In *PLDI 2005*.
10. H.-J. Boehm and S. V. Adve. Foundations of the C++ Concurrency Memory Model. In *PLDI 2008*.
11. S. Burckhardt, R. Alur, and M. K. Martin. Checkfence: Checking consistency of concurrent data types on relaxed memory models. In *PLDI 2007*.
12. S. Burckhardt and M. Musuvathi. Effective Program Verification for Relaxed Memory Models. In *CAV 2008*.
13. J. Cantin, M. Lipasti, and J. Smith. The Complexity of Verifying Memory Coherence. In *SPAA 2003*.
14. X. Fang, J. Lee, and S. Midkiff. Automatic fence insertion for shared memory multiprocessing. In *ICS 2003*.
15. T. Huynh and A. Roychoudhury. A memory model sensitive checker for C#. In *FM 2006*.
16. M. Kuperstein, M. Vechev, and E. Yahav. Automatic inference of memory fences. In *FMCAD 2010*.
17. L. Lamport. How to Make a Correct Multiprocess Program Execute Correctly on a Multiprocessor. *IEEE Trans. Comput.*, 46(7):779–782, 1979.
18. J. Lee and D.A. Padua. Hiding relaxed memory consistency with a compiler. *IEEE Transactions on Computers*, 50:824–833, 2001.
19. J. Manson, W. Pugh, and S. V. Adve. The Java Memory Model. In *POPL 2005*.
20. S. Park and D. L. Dill. An executable specification, analyzer and verifier for RMO. In *SPAA 1995*.
21. M. Rinard. Analysis of Multithreaded Programs. In *SAS 2001*.
22. J. Sevcik. *Program Transformations in Weak Memory Models*. PhD thesis, University of Edinburgh, 2008.
23. D. Shasha and M. Snir. Efficient and Correct Execution of Parallel Programs that Share Memory. In *TOPLAS 1988*.
24. Sparc Architecture Manual Version 9, 1994.
25. Z. Sura, X. Fang, C.-L. Wong, S.P. Midkiff, J. Lee, and D.A. Padua. Compiler techniques for high performance SC Java programs. In *PPoPP'05*. ACM, 2005.
26. Y. Yang, G. Gopalakrishnan, and G. Lindstrom. Memory model sensitive data race analysis. In *ICFEM 2004*.