Stability in Weak Memory Models

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

[1] Oxford University
[2] INRIA

Abstract. Concurrent programs running on weak memory models exhibit relaxed behaviours, making them hard to understand and to debug. To use standard verification techniques on such programs, we can force them to behave as if running on a Sequentially Consistent (SC) model. Thus, 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, e.g. 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 our offence tool which places 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 [18]. Hence standard analyses for concurrent programs might be unsound, as noted by M. Rinard in [25]. Memory model aware verification tools exist, $e.g.$ [24,11,15,30], but they often focus on one model at a time, or cannot handle the write atomicity relaxation exhibited $e.g.$ 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*; this underlies the *data race free guarantee* (DRF guarantee) of S. Adve and M. Hill [3].

Hence, as observed $e.g.$ 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 the *Total Store Order* (TSO) [28] memory model. We generalise their idea to a wider class of models (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 [27] 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, but are costly from a compilation point of view. As noted by S. Adve and H.-J. Boehm in [4], *"on 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 ensure stability to a program), but also their cost. Thus, we present 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 [27] 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 aforementioned characterisation. We detail how we used offence to test and measure the cost of our synchronisation mappings.

We formalised our results in Coq; we omit the proofs for brevity. A long version with proofs, the Coq development, the documentation and sources of offence and the experimental details can be found at `http://offence.inria.fr`.

1 Context

We give here the background on which we build our results. This section summarises our previous generic model [5], which embraces SC [18], Sun TSO, PSO and RMO [28], Alpha [7] and a fragment of Power [1]. Fig. 1 shows a table of our relations. The **iriw** test [10] (independent reads of independent writes), in Fig. 2, is our running example.

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 $\mathrm{val}(e)$, a processor $\mathrm{proc}(e)$, and a unique identifier. We represent each instruction by the events it issues. In Fig. 2, 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.

Name	Notation	Comment
program order	$m_1 \xrightarrow{\text{po}} m_2$	per-processor total order
preserved program order	$m_1 \xrightarrow{\text{ppo}} m_2$	pairs maintained in program order; $\xrightarrow{\text{ppo}} \subseteq \xrightarrow{\text{po}}$
read-from map	$w \xrightarrow{\text{rf}} r$	links a write to a read reading its value
write serialisation	$w_1 \xrightarrow{\text{ws}} w_2$	total order on writes to the same location
from-read map	$r \xrightarrow{\text{fr}} w$	r reads from a write preceding w in $\xrightarrow{\text{ws}}$
barriers	$m_1 \xrightarrow{\text{ab}} m_2$	ordering induced by barriers

Fig. 1. Table of relations

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`;

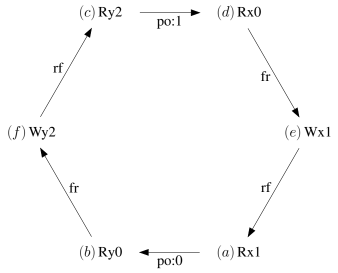

Fig. 2. The **iriw** test and a non-SC execution

We associate a program with an *event structure* $E \triangleq (\mathbb{E}, \xrightarrow{\text{po}})$, composed of its events $\mathbb{E}$ and the *program order* $\xrightarrow{\text{po}}$, a per-processor total order over $\mathbb{E}$. In Fig. 2, the read (a) from x on P_0 is in program order with the read (b) from y on P_0, $i.e.$ $(a)\text{R}x1 \xrightarrow{\text{po}} (b)\text{R}y0$. The $\xrightarrow{\text{dp}}$ relation (included in $\xrightarrow{\text{po}}$, 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 $X \triangleq (\xrightarrow{\text{ws}}, \xrightarrow{\text{rf}})$ of the corresponding program by two relations over $\mathbb{E}$. The *write serialisation* $\xrightarrow{\text{ws}}$ 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* $\xrightarrow{\text{rf}}$ links a write w to a read r from the same location that reads from w. We derive the *from-read map* $\xrightarrow{\text{fr}}$ from $\xrightarrow{\text{ws}}$ and $\xrightarrow{\text{rf}}$. A read r is in $\xrightarrow{\text{fr}}$ with a write w when the write w' from which r reads hit the memory before w did: $r \xrightarrow{\text{fr}} w \triangleq \exists w', w' \xrightarrow{\text{rf}} r \wedge w' \xrightarrow{\text{ws}} w$.

In Fig. 2, 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 $(e) \xrightarrow{\text{rf}} (a)$. If `r2=0`, the read (b) read its value from the initial state, thus before the write (f) on P_3, hence $(b) \xrightarrow{\text{fr}} (f)$. Similarly, we have $(f) \xrightarrow{\text{rf}} (c)$ from `r3=2`, and $(d) \xrightarrow{\text{fr}} (e)$ 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.

Code	Comment	Doc
`mfence`	WR non-cumulative barrier	[16, p. 291]
`cmp;bne;isync`	this sequence forms a RW, RR non-cumulative barrier	[1, p. 661]
`lwsync`	RW, RR, WW non-, A- and B-cumulative barrier	[1, p. 700]
`sync`	RW, RR, WW, WR non-, A- and B-cumulative barrier	[1, p. 700]

Fig. 3. Table of x86 and Power barriers

We model this by some subrelation of $\xrightarrow{\text{rf}}$ being *non-global*: they can be ignored by some processors. We write $\xrightarrow{\text{rfi}}$ (resp. $\xrightarrow{\text{rfe}}$) 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 $w \xrightarrow{\text{rfi}} r$ 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]), $w \xrightarrow{\text{rfe}} r$ is non-global, and w is said to be *non-atomic*. TSO authorises $e.g.$ store buffering ($i.e.$ $\xrightarrow{\text{rfi}}$ is non-global) but considers stores to be atomic ($i.e.$ $\xrightarrow{\text{rfe}}$ is global). We write $\xrightarrow{\text{grf}}$ for the global subrelation of $\xrightarrow{\text{rf}}$. We consider $\xrightarrow{\text{ws}}$ and $\xrightarrow{\text{fr}}$ global, since $\xrightarrow{\text{ws}}$ 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 $\xrightarrow{\text{po}}$, gathered in a subrelation $\xrightarrow{\text{ppo}}$ (*preserved program order*), is guaranteed to occur in this order. TSO for example authorises write-read pairs to be reordered, but nothing else: $\xrightarrow{\text{ppo}} = \xrightarrow{\text{po}} \setminus (\mathbb{W} \times \mathbb{R})$.

Finally, architectures provide barrier instructions to order certain pairs of events; Fig. 3 gives the x86 and Power ones that we use. We gather the orderings induced by barriers in the global relation $\xrightarrow{\text{ab}}$. Following [5], the relation $\xrightarrow{\text{fence}} \subseteq \xrightarrow{\text{po}}$ induced by a barrier `fence` is *non-cumulative* when it orders certain pairs of events surrounding the barrier: $\mathrm{NC}(\xrightarrow{\text{fence}}) \triangleq (\xrightarrow{\text{fence}} \subseteq \xrightarrow{\text{ab}})$. For example, the x86 `mfence` barrier is a non-cumulative barrier ordering write-read pairs only: $(w \xrightarrow{\text{mfence}} r) \Rightarrow (w \xrightarrow{\text{ab}} r)$. If there is a *dataflow dependency*, $e.g.$ *via* a comparison `cmp`, from a read to a conditional branch ($e.g.$ `bne`), Power `isync` forms a non-cumulative barrier when placed in $\xrightarrow{\text{po}}$ after the `cmp;bne` sequence, for read-read and read-write pairs : $(r \xrightarrow{\text{cmp;bne;isync}} m) \Rightarrow (r \xrightarrow{\text{ab}} m)$.

The relation $\xrightarrow{\text{fence}}$ is *cumulative* $w.r.t.$ another relation $\xrightarrow{\text{s}} \subseteq \xrightarrow{\text{rf}}$ when it makes the writes of $\xrightarrow{\text{s}}$ atomic ($e.g.$ by flushing the store buffers and caches). Formally, we define an A-cumulative (resp. B-cumulative) barrier as $\mathrm{AC}(\xrightarrow{\text{fence}}, \xrightarrow{\text{s}}) \triangleq (\xrightarrow{\text{s}};\xrightarrow{\text{fence}}) \subseteq \xrightarrow{\text{ab}}$ (resp. $\mathrm{BC}(\xrightarrow{\text{fence}}, \xrightarrow{\text{s}}) \triangleq (\xrightarrow{\text{fence}};\xrightarrow{\text{s}}) \subseteq \xrightarrow{\text{ab}}$). For example, Power `sync` barrier is non- (resp. A- and B-) cumulative for all pairs: we have $(m_1 \xrightarrow{\text{sync}} m_2)$ (resp. $(m_1 \xrightarrow{\text{rf}} w \xrightarrow{\text{sync}} m_2)$ and $(m_1 \xrightarrow{\text{sync}} w \xrightarrow{\text{rf}} m_2)$) implies $(m_1 \xrightarrow{\text{ab}} m_2)$. Power `lwsync` is non- (resp. A- and B-) cumulative for all pairs except write-read ones; we have $(m_1 \xrightarrow{\text{lwsync}} m_2)$ (resp. $(m_1 \xrightarrow{\text{rf}} r \xrightarrow{\text{lwsync}} m_2)$ and $(m_1 \xrightarrow{\text{lwsync}} w \xrightarrow{\text{rf}} m_2)$) implies $(m_1 \xrightarrow{\text{ab}} m_2)$ if $(m_1, m_2) \notin (\mathbb{W} \times \mathbb{R})$.

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

Validity The $\text{uniproc}(E, X) \triangleq \text{acyclic}(\xrightarrow{\text{ws}} \cup \xrightarrow{\text{fr}} \cup \xrightarrow{\text{rf}} \cup \xrightarrow{\text{po-loc}})$ condition (where $\xrightarrow{\text{po-loc}}$ 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 $\text{thin}(E, X) \triangleq \text{acyclic}(\xrightarrow{\text{rf}} \cup \xrightarrow{\text{dp}})$ condition prevents executions where values seem to come *out of thin air* [21]. 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:

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

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 checks above):

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

Finally, 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. TSO is weaker than SC, hence all the SC executions of a program are valid on TSO. In the following, we consider A_2 to be without barriers, $i.e.$ $\xrightarrow{\text{ab}_2} = \emptyset$.

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. 2 is not stable from Power to SC, for it is valid on Power yet not on SC. We can stabilise it 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 $\xrightarrow{\text{c}}$ over $\mathbb{E}$, such that $\forall xy, x \xrightarrow{\text{c}} y \Rightarrow \neg(y \xrightarrow{\text{po}} x)$ and a *synchronisation* relation $\xrightarrow{\text{s}}$ over $\mathbb{E}$. An execution (E, X) is *covered* when $\xrightarrow{\text{s}}$ *arbitrates* $\xrightarrow{\text{c}}$:

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

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

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

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, $i.e.$ locks are covering $w.r.t.$ the competing accesses. Two events are *competing* if they are from distinct processors, to the same location, and at least one of them is a write ($e.g.$ in Fig. 2, 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 $\xrightarrow{\text{lock}}$ (instantiated in Sec. 3.1) over $\mathbb{E}$, such that acyclic($\xrightarrow{\text{lock}} \cup \xrightarrow{\text{ws}} \cup \xrightarrow{\text{fr}} \cup \xrightarrow{\text{rf}}$), corresponding in Fig. 2 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 $\xrightarrow{\text{lock}} \cup \xrightarrow{\text{po}}$: $(a) \xrightarrow{\text{po}} (b) \xrightarrow{\text{lock}} (f) \xrightarrow{\text{lock}} (c) \xrightarrow{\text{po}} (d) \xrightarrow{\text{lock}} (e) \xrightarrow{\text{lock}} (a)$. If $\xrightarrow{\text{lock}} \cup \xrightarrow{\text{po}}$ is acyclic, then the execution of Fig. 2 is forbidden. Formally, we have:

Lem. 1. acyclic($\xrightarrow{\text{lock}} \cup \xrightarrow{\text{po}}$) $\Rightarrow$ covering($\overset{\text{cmp}}{\leftrightarrow}, (\xrightarrow{\text{lock}} \cup \xrightarrow{\text{po}})^+$)

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. By Lem. 1, it ensures stability to a program for any pair (A_1, A_2).

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 ($i.e.$ $x \xrightarrow{\text{ppo}_2} y$) but not on A_1 ($i.e.$ $\neg(x \xrightarrow{\text{ppo}_1} y)$). In Fig. 2, we have $(a) \xrightarrow{\text{po}} (b)$. Hence on a strong architecture A_2 such as SC where $\xrightarrow{\text{ppo}_2} = \xrightarrow{\text{po}}$, we have $(a) \xrightarrow{\text{ppo}_2} (b)$. On a weak architecture A_1 such as Power, where the read-read pairs in program order are not maintained, we have $\neg((a) \xrightarrow{\text{ppo}_1} (b))$.

Second, if the program reads from a write atomic on A_2 but not on A_1. In Fig. 2, we have $(e) \xrightarrow{\text{rfe}} (a)$. On a strong architecture A_2 such as SC where the writes are atomic, $i.e.$ $\xrightarrow{\text{grf}} = \xrightarrow{\text{rf}}$, we have $(e) \xrightarrow{\text{grf}} (a)$. On a weak architecture A_1 such as Power, which relaxes write atomicity, we have $\neg((e) \xrightarrow{\text{grf}} (a))$. We call such reads *fragile reads* and define them as ($\xrightarrow{\text{r}_{2\setminus 1}} \triangleq \xrightarrow{\text{r}_2} \setminus \xrightarrow{\text{r}_1}$ being the set difference):

$$\text{fragile}(r) \triangleq \exists w, w \xrightarrow{\text{grf}_{2\setminus 1}} r$$

We consider such differences between architectures as conflicts, and formalise this notion as follows. We consider that two events form a *fragile pair* (written $\xrightarrow{\text{frag}}$) 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:

$$m_1 \xrightarrow{\text{frag}} m_2 \triangleq m_1 \xrightarrow{\text{ppo}_2} m_2 \wedge \big(\neg(m_1 \xrightarrow{\text{ppo}_1} m_2) \vee \text{fragile}(m_1)\big)$$

An execution is covered if the relation $\xrightarrow{\text{ab}_1}$ arbitrates the fragile pairs. In Fig. 2, this corresponds to placing a barrier between (c) and (d) on P_1, $i.e.$ $(c) \xrightarrow{\text{ab}_1} (d)$, and another barrier between (a) and (b) on P_0, $i.e.$ $(a) \xrightarrow{\text{ab}_1} (b)$. Hence we have a cycle in $\xrightarrow{\text{ab}_1} \cup \xrightarrow{\text{rf}}$: $(d) \xrightarrow{\text{rfe}} (a) \xrightarrow{\text{ab}_1} (b) \xrightarrow{\text{rfe}} (c) \xrightarrow{\text{ab}_1} (d)$. If $\xrightarrow{\text{ab}_1}$ is A-cumulative $w.r.t.$ $\xrightarrow{\text{grf}_{2\setminus 1}}$, we create a cycle in $\xrightarrow{\text{ghb}_1}$, which forbids the execution: $(d) \xrightarrow{\text{ghb}_1} (b) \xrightarrow{\text{ghb}_1} (d)$. Formally, we have:

Lem. 2. AC($\xrightarrow{\text{ab}_1}, \xrightarrow{\text{grf}_{2\setminus 1}}$) $\Rightarrow$ covering($\xrightarrow{\text{frag}}, \xrightarrow{\text{ab}_1}$)

This lemma leads to a mapping which we call F (for fences), given in Fig. 4. This mapping places a barrier between each fragile pair of a program. Following Lem. 2, it enforces stability to a program for any pair (A_1, A_2). Recall that we give the semantics of the barriers that we use in the mapping F in Sec. 1, § *Architectures*, on p. 4 and Fig. 3.

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

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

Fig. 4. Mapping F: barriers

Name	Code	Comment	Doc [1]
load reserve	`lwarx r1,0,r2`	loads from the address in `r2` into `r1` and reserves the address in `r2`	p. 718
store conditional	`stwcx. r1,0,r2`	checks if the address in `r2` is reserved; if so, stores from `r1` into this address and writes 1 into register `cr`; if not, writes 0 into `cr`	p. 721
branch not equal	`bne L`	checks if register `cr` holds 0, if not branches to `L`	p. 63
compare	`cmpw r4, r6`	compares values in `r4` and `r6`	p. 102

Fig. 5. Table of Power assembly instructions, excluding barriers

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 $r_1 \xrightarrow{\text{po}} r_2$, 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`: $r_1 \xrightarrow{\text{sync}} r_2$. For a read-write pair $r \xrightarrow{\text{po}} w$, we need a cumulative barrier as well, but `lwsync` is sufficient here, for it will order the write 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`.

The mapping F agrees with D. Lea's JSR-133 Cookbook for Compiler Writers [19] for write-write and write-read pairs. Our mapping is much more conservative than D. Lea's for read-read and read-write pairs: it is unclear whether D. Lea's mapping (meant to implement Java's volatiles) intends to restore SC like ours, or rather a weaker memory model. The mapping F on write-write and write-read pairs corresponds to the optimised version of P. McKenney and R. Silvera's Example Power Implementation for C/C++ Memory Model [22] for "Store Seq Cst". Their "Load Seq Cst" is implemented by `sync;ld;cmp;bc; isync`. The use of `sync` before a load access corresponds to our mapping on read-read and read-write pairs. The sequence `cmp;bc;isync` after the same load access ensures that the Load Seq Cst has, in addition to an SC semantics, a *load acquire* semantics.

3 Synchronisation Idioms

To illustrate 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] *"on 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

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

(a) rmw

Initially r3 = ℓ, r4 = 0 and r5 = 1

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

(b) Lock

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

(c) Unlock

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

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 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 (see Fig. 5 for the instructions we use). 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 *successful*: it stores into the reserved address and the code exits the loop. Otherwise, `stwcx.` does not store and the code loops. Thus these instructions ensure *atomicity* to the code they surround (if 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 reads and writes 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` from ℓ before (in $\xrightarrow{\text{po}}$) the `stwcx.` that issued w, and (c) no other processor wrote to ℓ between r and w:

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

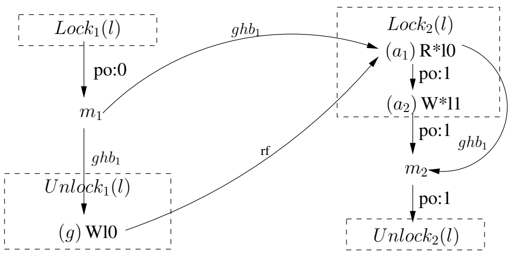

Fig. 7. Opening lock and unlock

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)).

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 $\text{Lock}(\ell, r)$ and an unlock $\text{Unlock}(\ell, r, w)$ (we define these two predicates in the next paragraph) with the same variable ℓ, and the events in $\xrightarrow{\text{po}}$ between the lock's read and the unlock's write:

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

We write $\text{loc}(\text{cs})$ 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 in Fig. 7: on the left is cs_1, composed of a lock $\text{Lock}_1(\ell)$, an event m_1 and an unlock $\text{Unlock}_1(\ell)$, 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 $\text{Lock}_2(\ell)$ reads from (g). Thus, cs_1 and cs_2 are serialised if cs_2 Lock's read (written $\text{R}(\text{cs}_2)$) reads from cs_1 Unlock's write (written $\text{W}(\text{cs}_1)$):

$$\text{cs}_1 \xrightarrow{\text{css}_\ell} \text{cs}_2 \triangleq \text{loc}(\text{cs}_1) = \text{loc}(\text{cs}_2) = \ell \wedge \text{W}(\text{cs}_1) \xrightarrow{\text{rf}} \text{R}(\text{cs}_2)$$

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

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

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

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.`:

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

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 $\xrightarrow{\mathrm{ghb}_1}$ 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:

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

Fig. 6(c) shows Power's unlock, starting (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 $\xrightarrow{\mathrm{ghb}_1}$ with the read (a_1) of cs_2's Lock. Thus we define an export barrier as B-cumulative, but only $w.r.t.$ reads issued by the `lwarx` of an atomic pair:

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

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:

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

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)):

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

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

We show that this semantics ensures the acyclicity of $\xrightarrow{\mathrm{lock}} \cup \xrightarrow{\mathrm{po}}$, $i.e.$ following Lem. 1, $(\xrightarrow{\mathrm{lock}} \cup \xrightarrow{\mathrm{po}})^+$ is covering for the competing accesses. Hence locks on the competing accesses ensures a SC behaviour to Power programs:

Lem. 3. $\forall EX, A_1 . \mathrm{valid}(E, X) \Rightarrow \mathrm{acyclic}(\xrightarrow{\mathrm{lock}} \cup \xrightarrow{\mathrm{po}})$

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 [8,26]. 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. 3 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. 3 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 and confirmed this semantics of `lwsync` with our diy tool [5], by running our automatically generated tests up to 10^{10} times each (see the logs online).

3.2 Read-Modify-Write Primitives

By Lem. 2, we can restore SC in the **iriw** test of Fig. 2 using A-cumulative barriers between the fragile pairs (a) and (b) on P_0, and (c) and (d) on P_1. Yet, cumulativity

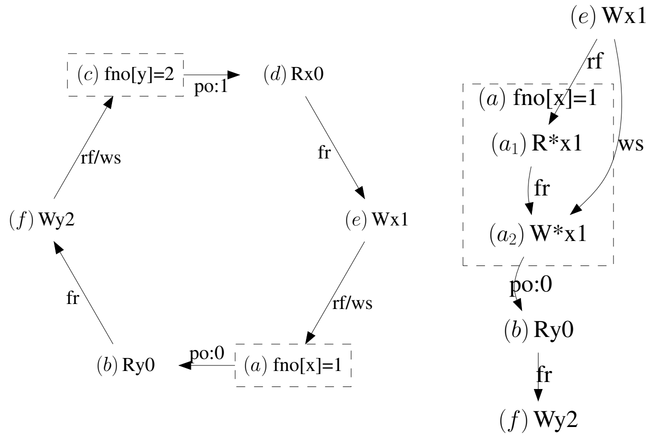

Fig. 8. (a) **iriw** after mapping P (b) Opening fno on P_0

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. 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:

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

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:

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

In Fig. 8(b), the write (e) from which (a_1) reads hits the memory before (a_2), $i.e.$ $(e) \xrightarrow{\text{ws}} (a_2)$. Hence there are two paths from (e) to (b): $(e) \xrightarrow{\text{rf}} (a_1) \xrightarrow{\text{po}} (b)$ and $(e) \xrightarrow{\text{ws}}$

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

Fig. 9. Mappings A and P: rmw

$(a_2) \xrightarrow{\text{po}} (b)$. Thus we can trade the fragile pair (a_1, b) for (a_2, b) and compensate the lack of write atomicity of (e) ($i.e.$ $(e) \xrightarrow{\text{rfe}} (a)$ not global) with the write serialisation between (e) and (a_2) (thanks to the rmw) instead of cumulativity before. Formally, we prove that a sequence $w \xrightarrow{\text{grf}_2 \setminus 1} r \xrightarrow{\text{ppo}_2} m$ with r protected is globally ordered on A_1:

Lem. 4. $\forall wrm, (\text{protected}(r) \wedge w \xrightarrow{\text{grf}_2 \setminus 1} r \xrightarrow{\text{ppo}_2} m) \Rightarrow w \xrightarrow{\text{ws}}; \xrightarrow{\text{ghb}_1} m$

Thus, if we protect the fragile reads, the only remaining fragile pairs are the ones in $\xrightarrow{\text{ppo}_2 \setminus 1}$. In Fig. 8(a), we have $(e) \xrightarrow{\text{ws}} (a_2) \xrightarrow{\text{po}} (b) \xrightarrow{\text{fr}} (f)$ and $(f) \xrightarrow{\text{ws}} (c_2) \xrightarrow{\text{po}} (d) \xrightarrow{\text{fr}} (e)$, hence a cycle in $\xrightarrow{\text{ws}} \cup \xrightarrow{\text{fr}} \cup \xrightarrow{\text{po}}$. Since $\xrightarrow{\text{ws}}$ and $\xrightarrow{\text{fr}}$ 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 $\xrightarrow{\text{ppo}_2 \setminus 1}$ ensure stability:

Lem. 5. $A_1 . \text{valid}(E, X) \wedge \text{protected}(E, X) \wedge (\xrightarrow{\text{ppo}_2 \setminus 1} \subseteq \xrightarrow{\text{ab}_1}) \Rightarrow A_2 . \text{valid}(E, X)$

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 the fragile pairs, and a barrier between this fno and the second access of the fragile pairs. 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. 5, it enforces stability to a program for any pair (A_1, A_2).

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 if this dependency exists, we show in 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. 2, 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.

D. Shasha and M. Snir provide in [27] an analysis to place barriers in a program, in order to enforce a SC behaviour. They examine in [27, 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 architecture A and event structure E, a cycle $\xrightarrow{\sigma} \subseteq (\overset{\text{cmp}}{\leftrightarrow} \cup \xrightarrow{\text{po}})^+$ (where $\overset{\text{cmp}}{\leftrightarrow}$ is the competing relation of Sec. 2) is *critical on* A, written $\text{critical}_A(E, \xrightarrow{\sigma})$, when it is not a cycle in $(\overset{\text{cmp}}{\leftrightarrow} \cup \xrightarrow{\text{ppo}_A})^+$ and satisfies the two following properties. **(i)** Per processor, there are at most two memory accesses (x, y) on this processor 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 $r \overset{\text{cmp}}{\leftrightarrow} w \overset{\text{cmp}}{\leftrightarrow} r$). In Fig. 2, the execution of **iriw** has a critical cycle on Power.

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 E contains no critical cycle on A, $i.e.$ that an execution valid on A is SC if and only if E contains no critical cycle on A:

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

This theorem means that 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. Hence to restore SC, we should arbitrate (with a covering relation) the conflicting accesses (competing accesses or fragile pairs) occurring in the 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 $e_1 \xrightarrow{\text{po}} e_2$, mapping to static events f_1 and f_2, we compute the *static program order* $\xrightarrow{\text{pos}}$ such that $e_1 \xrightarrow{\text{po}} e_2$ entails $f_1 \xrightarrow{\text{pos}} f_2$ 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 would need alias analysis to compute an over-approximation of the locations of each static event, considering for example that all pairs of memory accesses by distinct processors conflict, if one of them is a write.

With F the set of static events, we call the triple $(F, \xrightarrow{\text{pos}}, \overset{\text{cmps}}{\leftrightarrow})$ *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 [20], 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 [20] 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 cycle of relations. First, we generate tests from critical cycles, $e.g.$ **iriw** in Fig. 2. 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, turn its memory locations and registers to fresh ones, and interleave the codes of the programs. We produced two series of tests, written X, each series consisting of 209 tests for Power and 58 tests for x86.

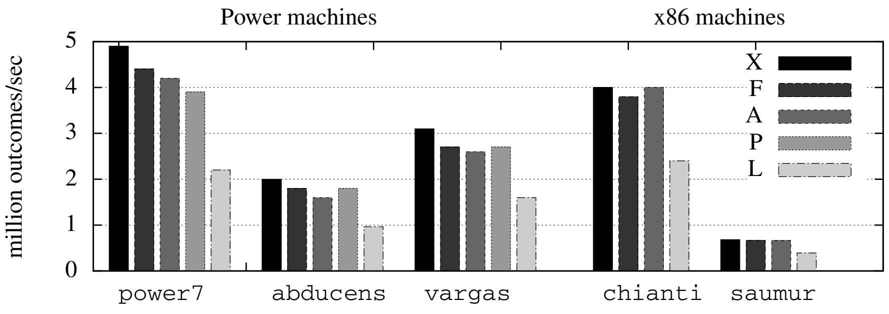

Fig. 10. Productivity observed during soundness experiments

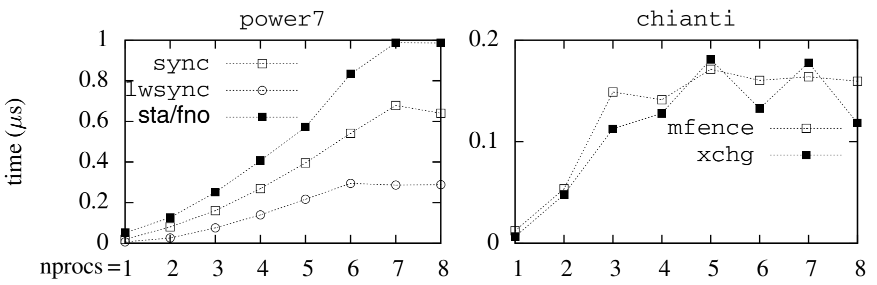

Fig. 11. Time of synchronisation constructs, in microseconds

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. 2), P (Lem. 5) and L (Lem. 1 and 3).

Cost measures. Fig. 10 shows the *productivity*, *i.e.* the number of outcomes 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 our 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 with offence 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. 11. 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 [3,10,23], the semantics of synchronisation idioms [9,8], and the insertion of barriers [27,14,11,17] have been extensively studied, but most of these works focus on one kind of synchronisation at a time, and none of them addresses Power traits 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 [20] how to restore SC at compiler level: we used their global fence placement algorithm. Our work improves on [20] $w.r.t.$ semantical fundations: as a result, we use Power `lwsync` when possible and we do not use x86 `lfence` and `sfence` barriers, irrelevant in user-level code. Our mappings could be included in their Java compiler [29], $i.e.$ using `lwsync` for Power, and `xchg` for x86.

Conclusion. Our formal study of stability in weak memory models allows us to define several mappings of Power or x86 assembly code, which, as we prove in Coq, give a SC behaviour to a 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 [27] 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 on our test set. Our work could for example benefit to compiler writers and semanticists interested in standardisation and implementability ($e.g.$ of Java volatiles or C++ atomics on Power platforms).

Acknowledgements. We thank Susmit Sarkar, Peter Sewell, Michael Tautschnig, Jules Villard and Boris Yakobowski for comments on a draft.

References

1. Power ISA Version 2.06 (2009)
2. Adve, S.V., Gharachorloo, K.: Shared Memory Consistency Models: A Tutorial. IEEE Computer 29, 66–76 (1995)
3. Adve, S.V., Hill, M.D.: Weak Ordering - A New Definition. In: ISCA (1990)
4. Adve, S.V., Boehm, H.-J.: Memory Models: A Case for Rethinking Parallel Languages and Hardware. To appear in CACM
5. Alglave, J., Maranget, L., Sarkar, S., Sewell, P.: Fences in weak memory models. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 258–272. Springer, Heidelberg (2010)
6. Alglave, J., Maranget, L., Sarkar, S., Sewell, P.: litmus: Running tests against hardware. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 41–44. Springer, Heidelberg (2011)
7. Alpha Architecture Reference Manual, 4th edn. (2002)
8. Boehm, H.-J.: Reordering Constraints for Pthread-Style Locks. In: PPoPP (2007)

9. Boehm, H.-J.: Threads Cannot Be Implemented As a Library. In: PLDI (2005)
10. Boehm, H.-J., Adve, S.V.: Foundations of the C++ Concurrency Memory Model. In: PLDI (2008)
11. Burckhardt, S., Alur, R., Martin, M.K.: Checkfence: Checking consistency of concurrent data types on relaxed memory models. In: PLDI (2007)
12. Burckhardt, S., Musuvathi, M.: Effective program verification for relaxed memory models. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 107–120. Springer, Heidelberg (2008)
13. Cantin, J., Lipasti, M., Smith, J.: The Complexity of Verifying Memory Coherence. In: SPAA (2003)
14. Fang, X., Lee, J., Midkiff, S.: Automatic fence insertion for shared memory multiprocessing. In: ICS (2003)
15. Huynh, T.Q., Roychoudhury, A.: A memory model sensitive checker for C#. In: Misra, J., Nipkow, T., Karakostas, G. (eds.) FM 2006. LNCS, vol. 4085, pp. 476–491. Springer, Heidelberg (2006)
16. Intel 64 and IA-32 Architectures Software Developer's Manual, vol. 3A, rev. 30 (March 2009)
17. Kuperstein, M., Vechev, M., Yahav, E.: Automatic inference of memory fences. In: FMCAD (2010)
18. Lamport, L.: How to Make a Correct Multiprocess Program Execute Correctly on a Multiprocessor. IEEE Trans. Comput. 46(7), 779–782 (1979)
19. Lea, D.: The JSR-133 Cookbook for Compiler Writers (September 2006), `http://gee.cs.oswego.edu/dl/jmm/cookbook.html`
20. Lee, J., Padua, D.A.: Hiding relaxed memory consistency with a compiler. IEEE Transactions on Computers 50, 824–833 (2001)
21. Manson, J., Pugh, W., Adve, S.V.: The Java Memory Model. In: POPL (2005)
22. McKenney, P., Silvera, R.: Example Power Implementation for C/C++ Memory Model (August 2008), `http://www.rdrop.com/users/paulmck/scalability/paper/N2745r.2010.02.19a.html`
23. Owens, S.: Reasoning about the implementation of concurrency abstractions on x86-TSO. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 478–503. Springer, Heidelberg (2010)
24. Park, S., Dill, D.: An executable specification, analyzer and verifier for RMO. In: SPAA 1995 (1995)
25. Rinard, M.: Analysis of multithreaded programs. In: Cousot, P. (ed.) SAS 2001. LNCS, vol. 2126, p. 1. Springer, Heidelberg (2001)
26. Sevcik, J.: Program Transformations in Weak Memory Models. PhD thesis, University of Edinburgh (2008)
27. Shasha, D., Snir, M.: Efficient and Correct Execution of Parallel Programs that Share Memory. In: TOPLAS (1988)
28. Sparc Architecture Manual Version 9 (1994)
29. Sura, Z., Fang, X., Wong, C.-L., Midkiff, S.P., Lee, J., Padua, D.A.: Compiler techniques for high performance SC Java programs. In: PPoPP 2005. ACM Press, New York (2005)
30. Yang, Y., Gopalakrishnan, G.C., Lindstrom, G.: Memory-model-sensitive data race analysis. In: Davies, J., Schulte, W., Barnett, M. (eds.) ICFEM 2004. LNCS, vol. 3308, pp. 30–45. Springer, Heidelberg (2004)