Noname manuscript No.
(will be inserted by the editor)

Fences in Weak Memory Models (Extended Version)

Jade Alglave[1,3] · **Luc Maranget**[1] · **Susmit Sarkar**[2] · **Peter Sewell**[2]

the date of receipt and acceptance should be inserted later

Abstract We present a class of relaxed memory models, defined in Coq, parameterised by the chosen permitted local reorderings of reads and writes, and by the visibility of inter- and intra-processor communications through memory (*e.g.* store atomicity relaxation). We prove results on the required behaviour and placement of memory fences to restore a given model (such as Sequential Consistency) from a weaker one. Based on this class of models we develop a tool, diy, that systematically and automatically generates and runs litmus tests. These tests can be used to explore the behaviour of processor implementations and the behaviour of models, and hence to compare the two against each other. We detail the results of experiments on Power and a model we base on them.

1 Introduction

Most multiprocessors exhibit subtle relaxed-memory behaviour, with writes from one thread not immediately visible to all others; they do not provide sequentially consistent memory [21]. For some, such as x86 [2,1,26,24,28] and Power [25], the vendor documentation is in inevitably-ambiguous informal prose, leading to confusion. Thus we have no foundation for software verification of concurrent systems code, and no target specification for hardware verification of microarchitecture. To remedy this state of affairs, we take a firmly empirical approach, developing, in tandem, testing tools and models of multiprocessor behaviour—the test results guiding model development and the modelling suggesting interesting tests. In this paper we make five contributions:

1. We introduce a class of axiomatic memory models, defined in Coq [13], which we show how to instantiate to produce Sequential Consistency (SC), Sparc Total Store Order (TSO) [30], and a Power model (see item 4 below).
2. We describe our diy testing tool. Much discussion of memory models has been in terms of *litmus tests* (*e.g.* **iriw** [14]): ad-hoc multiprocessor programs for which particular final states may be allowed on a given architecture. Given a

[1] INRIA [2] University of Cambridge [3] Oxford University

potential violation of *SC*, diy *systematically* and *automatically* generates litmus tests (including classical ones such as **iriw**) and runs them on the hardware. These tests can be used to explore the behaviour of processor implementations and also the outcomes permitted by a model, and hence to compare the two; we illustrate this by our exploration of Power machines.

3. We use diy to generate about 800 tests, running them up to $1e12$ times on three Power machines. They identified a rarely occurring implementation error in Power 5 memory barriers (for which IBM is providing a workaround), and further suggest that Power 6 does not suffer from this.
4. Based on these, and on other test results, we developed an axiomatic memory model (the *CAV 2010 model*) for Power which captures several important aspects of the processor's behaviour. Notably, it describes the lack of *multi-copy store atomicity* [5,12], despite being in a simple global-time style rather than the per-processor timelines implied by the architecture text. It also models the *ordering relaxations* we observe and *A-cumulative barriers* [25]. The model is sound with respect to all our experimental results, though for some of the Power barriers it is weaker than one might like; we discuss this in detail.
5. We prove in Coq theorems about the strength and placement of memory barriers required to regain a strong model from a weaker model.

The experimental details and the sources and documentation of diy are available online[1], as are the Coq development and typeset outlines of the proofs[2], which are further described in the first author's PhD thesis [7]. This paper extends a conference paper in CAV 2010 [9], adding more explanation and more details of diy and of the axiomatic Power model introduced there.

2 Our class of models

A memory model determines whether a candidate execution of a program is *valid*. For example, Fig. 1(a) shows a simple litmus test, comprising an initial state (which gathers the initial values of registers and memory locations used in the test), a program in pseudo- or assembly code, and a final condition on registers and memory (we write `x`, `y` for memory locations and `r1`, `r2` for registers). If each location initially holds 0 (henceforth we omit the initial state if so), then, *e.g.* on x86 processors, there are valid executions with the specified final state [24].

2.1 Informal overview of our approach

We start here by explaining the concepts that we use at a high level. We then define these concepts formally in the forthcoming subsections.

Describing executions of programs We study concurrent programs such as the one given in Fig. 1(a). Each of these programs gives an initial state describing the initial values in memory locations and registers initially, *e.g.* `x=0; y=0` in Fig. 1(a), meaning that we suppose that the memory locations `x` and `y` hold the value `0`

[1] http://diy.inria.fr/

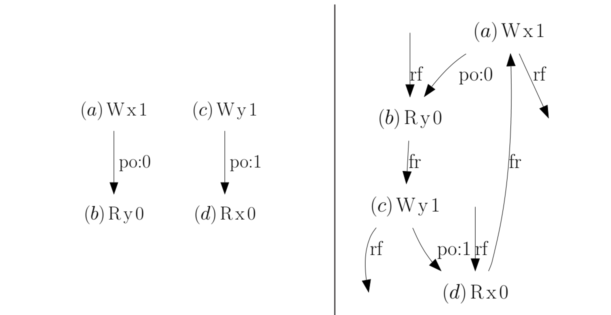
[2] http://diy.inria.fr/wmm/

Init: `x=0; y=0;`

P_0	P_1
(a) `x` ← `1`	(c) `y` ← `1`
(b) `r1` ← `y`	(d) `r2` ← `x`

Observed? `r1=0; r2=0;`

(a) Program



(b) Events and Program Order

(c) An execution witness

Fig. 1 A program and a candidate execution

initially. Except when the examples are specific to our study of the Power architecture, *i.e.* in Sec. 6 and 7, we write the instructions in pseudo-code; for example `x` ← `1` is a store of value `1` into memory location `x`, and `r1` ← `y` is a load from memory location `y` into a register `r1`. We depict a concurrent program as a table, where the columns are processors (*e.g.* P_0 and P_1 in Fig. 1(a)), and the lines are labelled with letters—for example in Fig. 1(a), the first line, which holds `x` ← `1`, is labelled (a).

We describe a candidate execution of a given program using *memory events*, corresponding to the memory accesses yielded by executing the instructions of the program. For example, we give in Fig. 1(b) the memory events of one candidate execution of the program of Fig. 1(a): the write event (a) W x 1 corresponds to the store `x` ← `1` at line (a). In this candidate execution both reads read value `0`.

In addition to these memory events, a candidate execution of a program consists of several relations over them. One of these relations represents the program order as given by an unfolding of a control-flow path through the text of the program—in any execution of the program Fig. 1(a), the execution of the instruction at line (a) is program-order-before execution of the instruction at line (b). This is expressed as the $\xrightarrow{\text{po}}$ relation between the corresponding events in Fig. 1(b). Other relations represent the interaction with memory: the reads-from relation $\xrightarrow{\text{rf}}$ indicates which write the value of a each read event comes from; and the write serialisation $\xrightarrow{\text{ws}}$ represents the *coherence order* for each location (for each location, there is a total order over the writes to that location). Reads-from edges with no source or target represent reads from the initial state or writes that appear in the final state respectively.

Defining the validity of an execution We then define the validity of a given candidate execution as acyclicity checks of certain unions of these relations.

Many interesting candidate executions (and all the executions that we will show in this paper) contain at least one cycle, such as that depicted in Fig. 1(c). Typically, this cycle will exhibit the fact that the execution that we choose to depict is invalid in the Sequential Consistency (SC) model [21]. The execution in Fig. 1(c) is allowed in TSO and in Power, but not in SC, where at least one of the reads would have to read `1`.

In addition, our test programs contain a constraint on the final state. For example, the program in Fig. 1(a) shows the line "Observed? `r1=0;r2=0`". We use several different keywords to express the final state of our programs. The keyword "Observed" (or its counterpart "Not observed") refers to empirical results. This means that we actually observed an execution satisfying the final state constraint on a given machine. When there is a question mark, as in "Observed?", this means that we question whether the outcome is observable or not on a given machine. The keyword "Allowed" (or its counterpart "Forbidden") refers to whether a given model allows (or forbids) the specified outcome. This means that we can deduce from the definition of the model that this outcome is allowed (or forbidden).

The fact that the specified final state of a given program—such as "Observed? `r1=0; r2=0`" in Fig. 1(a)—is observable or allowed relates to the graphs describing the executions of this program—such as the one given in Fig. 1(c).

Let us examine the Allowed/Forbidden case first. As we said above, the validity of an execution in the model we present here boils down to the presence of certain cycles in the execution graph. Thus, if an execution graph contains a cycle, then we have to examine if the model that we are studying allows some *'relaxation'* of the relations that are involved in this cycle. If some relaxations are allowed, then the cycle does not forbid the execution, and the final state is allowed by the model. For example in Fig. 1(c), on a model such as SC where no relaxation is allowed, the cycle forbids the execution. On a model such as x86, where the program order between a write and a read may be relaxed, the cycle does not forbid the execution, for the program order relation (written $\overset{\text{po}}{\to}$ in Fig. 1(c)) between (a) and (b) (and similarly (c) and (d)) is relaxed.

For the Observed/Not observed case, we have to run the test against hardware to check whether the specified final outcomes appears. If we observe a given final state, we sometimes can deduce which is the feature of the hardware—as represented by our model—that allows this outcome. For example, we were able to observe the final state of Fig. 1(a) on x86 machines. From this we deduce that the cycle in Fig. 1(c) does not forbid the execution on some x86 machines, and furthermore that the x86 model allows the reordering of write-read pairs. Of course, as usual with black-box testing, one cannot deduce anything with certainty from the absence of an empirical observation.

2.2 Events and program order

As sketched above, rather than dealing directly with programs, our models are in terms of the *events* $\mathbb{E}$ occurring in a candidate program execution. A *memory event* m represents a memory access, specified by its direction (write or read), its location $\text{loc}(m)$, its value $\text{val}(m)$, its processor $\text{proc}(m)$, and a unique label. The store to `x` with value 1 marked (a) in Fig. 1(a) generates the event $(a)\,\text{W x 1}$ in Fig. 1(b). Henceforth, we write r (resp. w) for a read (resp. write) event. We write $\mathbb{M}_{\ell,v}$ (resp. $\mathbb{R}_{\ell,v}$, $\mathbb{W}_{\ell,v}$) for the set of memory events (resp. reads, writes) to a location ℓ with value v (we omit ℓ and v when quantifying over all of them). A barrier instruction generates a *barrier event* b; we write $\mathbb{B}$ for the set of all such events.

Name	Notation	Comment	Sec.
program order	$m_1 \overset{\text{po}}{\to} m_2$	per-processor total order	2.2
dependencies	$m_1 \overset{\text{dp}}{\to} m_2$	dependencies	2.2
po-loc	$m_1 \overset{\text{po-loc}}{\to} m_2$	program order restricted to the same location	2.5
preserved program order	$m_1 \overset{\text{ppo}}{\to} m_2$	pairs maintained in program order	2.4
read-from map	$w \overset{\text{rf}}{\to} r$	links a write to a read reading its value	2.3
external read-from map	$w \overset{\text{rfe}}{\to} r$	$\overset{\text{rf}}{\to}$ between events from distinct processors	2.4
internal read-from map	$w \overset{\text{rfi}}{\to} r$	$\overset{\text{rf}}{\to}$ between events from the same processor	2.4
global read-from map	$w \overset{\text{grf}}{\to} r$	$\overset{\text{rf}}{\to}$ considered global	2.4
write serialisation	$w_1 \overset{\text{ws}}{\to} w_2$	total order on writes to the same location	2.3
from-read map	$r \overset{\text{fr}}{\to} w$	r reads from a write preceding w in $\overset{\text{ws}}{\to}$	2.3
barriers	$m_1 \overset{\text{ab}}{\to} m_2$	ordering induced by barriers	2.4
global happens-before	$m_1 \overset{\text{ghb}}{\to} m_2$	union of global relations	2.4
communication	$m_1 \overset{\text{com}}{\to} m_2$	$(m_1, m_2) \in (\overset{\text{rf}}{\to} \cup \overset{\text{ws}}{\to} \cup \overset{\text{fr}}{\to})$ (written $\overset{\text{hb-seq}}{\to}$ in [9])	2.5

Fig. 2 Table of relations

The models are defined in terms of binary relations over these events, and Fig. 2 has a first table of the relations we use. The relations given in Fig. 2 are entirely generic; we give some more relations that are specific to Power in Fig. 17.

As usual, the *program order* $\overset{\text{po}}{\to}$ is a total order amongst the events from the same processor that never relates events from different processors. It reflects the sequential execution of instructions on a single processor: given two instruction execution instances i_1 and i_2 that generate events e_1 and e_2, $e_1 \overset{\text{po}}{\to} e_2$ means that a sequential processor would execute i_1 before i_2. When instructions may perform several memory accesses, we take intra-instruction dependencies [26] into account to build a total order.

We postulate a $\overset{\text{dp}}{\to}$ relation to model the dependencies between instructions, such as *data* or *control dependencies* [25, pp. 653-668]. This relation is a subrelation of $\overset{\text{po}}{\to}$, and always has a read as its source.

2.3 Execution witnesses

Although $\overset{\text{po}}{\to}$ conveys important features of program execution, *e.g.* branch resolution, it does not characterise an execution. To do so, we postulate, as part of the data of a candidate execution, two additional relations $\overset{\text{ws}}{\to}$ and $\overset{\text{rf}}{\to}$ over memory events.

Write serialisation We assume all values written to a given location ℓ to be serialised, following a *coherence order*. This means, following the Power documentation [25, p.657, 1st col, last §], that all stores to a given memory location ℓ are totally ordered:

> Memory coherence refers to the ordering of stores to a single location. Atomic stores to a given location are *coherent* if they are serialised in some order, and no processor or mechanism is able to observe any subset of those stores as occurring in a conflicting order.

This property is widely assumed by modern architectures, including for example Sparc TSO [30] and x86-TSO [24].

Consequently, we define $\overset{\text{ws}}{\to}$ as the union of the coherence orders for all memory locations, which must be well formed following the wf-ws predicate:

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

Reads-from map We write $w \overset{\text{rf}}{\to} r$ to mean that r loads the value stored by w (so w and r must share the same location and value). Given a read r there exists a unique write w such that $w \overset{\text{rf}}{\to} r$. The write w can be an *init* store when r loads from the initial state. The initial store to a location x is defined as the first write in the coherence order for x. Thus, $\overset{\text{rf}}{\to}$ must be well formed following the wf-rf predicate:

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

From-read map It is useful to define a derived relation $\overset{\text{fr}}{\to}$ (as in [6]) which gathers all pairs of reads r and writes w such that r reads from a write that is before w in $\overset{\text{ws}}{\to}$ (as in Fig. 3):

$$r \overset{\text{fr}}{\to} w \triangleq \exists\, w'.\, w' \overset{\text{rf}}{\to} r \wedge w' \overset{\text{ws}}{\to} w$$

The significance of $\overset{\text{fr}}{\to}$ is as follows. Some of the weaknesses of multiprocessor memory models arises from the fact that a write is not necessarily made available to all potential reading threads in one atomic step. In TSO models (e.g. for Sparc and x86-TSO), this is true in a relatively benign way: the writing thread might read 'early' from its own write before that is propagated to all other threads. In more relaxed models (e.g. for Power and ARM) there may be more complex behaviour, for example with a write being made available first to the writing thread itself, then to near-neighbours of that thread (that share some level of the cache hierarchy), and later to other threads. If one thinks of a write event as representing the point in time when a write has been made available to all threads, and a read event as the point when the value of the read is determined, then reads-from $\overset{\text{rf}}{\to}$ edges are not necessarily forwards in time (a read can have read from a write before the write is made available to all), but a from-read $\overset{\text{fr}}{\to}$ edge, from a read to a coherence-successor of the write it read from, is necessarily forwards in time (otherwise the read would have to read from the coherence-later write).

Execution witnesses We define an *execution witness* X as follows:

$$X \triangleq (\mathbb{E}, \overset{\text{po}}{\to}, \overset{\text{dp}}{\to}, \overset{\text{rf}}{\to}, \overset{\text{ws}}{\to})$$

The well-formedness predicate wf on execution witnesses is the conjunction of those for $\overset{\text{ws}}{\to}$ and $\overset{\text{rf}}{\to}$:

$$\text{wf}(X) \triangleq \text{wf-ws}(\overset{\text{ws}}{\to}) \wedge \text{wf-rf}(\overset{\text{rf}}{\to})$$

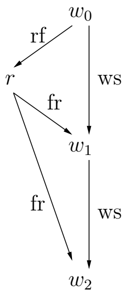
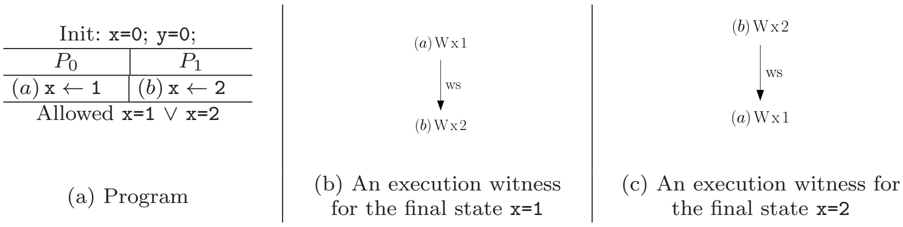


Fig. 3 The $\xrightarrow{\text{fr}}$ relation is derived from $\xrightarrow{\text{ws}}$ and $\xrightarrow{\text{rf}}$



Init: `x=0; y=0;`

P_0	P_1
(a) `x` ← `1`	(b) `x` ← `2`

Allowed `x=1` ∨ `x=2`

(a) Program

(b) An execution witness for the final state `x=1`

(c) An execution witness for the final state `x=2`

Fig. 4 A program and its two different write serialisations

Fig. 1(c) shows one particular execution witness for the test of Fig. 1(a). The load (d) reads the initial value of `x`, later overwritten by the store (a). Since the initial store to `x` comes first in $\xrightarrow{\text{ws}}$, hence before (a), we have $(d) \xrightarrow{\text{fr}} (a)$.

Note that an execution witness describes only one candidate execution of a program. A given program can have many candidate execution witnesses, with different control-flow paths, different values read from memory, different reads-from relations, and different write serialisations; a memory model will permit some of these and forbid others. Consider for example the program given in Fig. 4(a). The processors P_0 and P_1 both write to the same location x. This program has two candidate executions, shown in Fig. 4(b) and (c), with different write serialisations. The one in Fig. 4(b) covers the case where P_0's write is coherence-before P_1's. The second one, in Fig. 4(c), covers the converse.

For an example of candidate executions differing in the writes that are the sources of a read-from edge, consider the program in Fig. 5(a). The processors P_0 and P_1 both write the value 1 into memory location x. The processor P_2 reads from x. Hence, this program can have at least two distinct execution witnesses, given in Fig. 5(b) and (c). The one in Fig. 5(b) covers the case where P_2 reads from P_0. The second one, given in Fig. 5(c), covers the case where P_2 reads from P_1.

2.4 Global Happens-Before

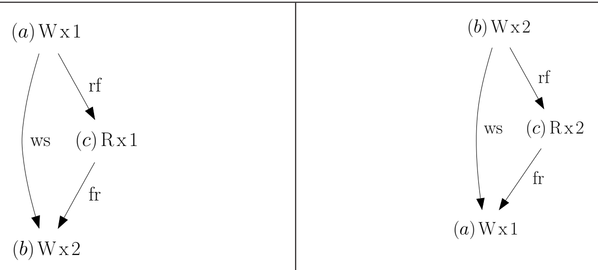
In the family of memory models we use in this paper, a candidate execution witness is *valid* if the memory events can be embedded in an acyclic *global happens-before* relation $\xrightarrow{\text{ghb}}$ (together with two auxiliary conditions detailed in Sec. 2.5). Each

Init: `x=0`;

P_0	P_1	P_2
(a) `x` $\leftarrow$ `1`	(b) `x` $\leftarrow$ `1`	(c) `r1` $\leftarrow$ `x`

Allowed `r1=1`;

(a) Program



(b) An execution witness where P_2 reads from P_0

(c) An execution witness where P_2 reads from P_1

Fig. 5 A program and two possible read-from maps

model is determined by the choice of relations which we include in this global happens-before relation. The position of a write in $\xrightarrow{\text{ghb}}$ represents a point in global time when this write becomes visible to all processors (we say it is *globally performed* then); whereas the position of a read in $\xrightarrow{\text{ghb}}$ represents the point in global time when the read takes place. Note that these are concepts associated with the model; the precise relationship between the model events and concrete microarchitectural events in processor implementations may be subtle — for example, in an implementation write might never actually be propagated to threads that do not read the associated cache line. There remain key choices as to which relations we include in $\xrightarrow{\text{ghb}}$ (*i.e.* which we consider to be in global time), which leads us to define a class of models.

Globality Writes are not necessarily made available to all processors at once, so $\xrightarrow{\text{rf}}$ edges are not necessarily included in $\xrightarrow{\text{ghb}}$. Let us distinguish between internal (resp. external) $\xrightarrow{\text{rf}}$, when the two events in $\xrightarrow{\text{rf}}$ are on the same (resp. distinct) processor(s), written $\xrightarrow{\text{rfi}}$ (resp. $\xrightarrow{\text{rfe}}$) : $w \xrightarrow{\text{rfi}} r \triangleq w \xrightarrow{\text{rf}} r \wedge \text{proc}(w) = \text{proc}(r)$ and $w \xrightarrow{\text{rfe}} r \triangleq w \xrightarrow{\text{rf}} r \wedge \text{proc}(w) \neq \text{proc}(r)$. Some architectures allow *store forwarding* (or *read own writes early* [5]): the processor issuing a given write can read its value before any other participant accesses it. Then $\xrightarrow{\text{rfi}}$ is not included in $\xrightarrow{\text{ghb}}$. Other architectures allow two processors sharing a cache to read a write issued by their neighbour *w.r.t.* the cache hierarchy before any other participant that does not share the same cache—a particular case of *read others' writes early* [5]. Then $\xrightarrow{\text{rfe}}$ is not considered global. We write $\xrightarrow{\text{grf}}$ for the subrelation of $\xrightarrow{\text{rf}}$ included in $\xrightarrow{\text{ghb}}$.

In our class of models, $\xrightarrow{\text{ws}}$ and $\xrightarrow{\text{fr}}$ are always included in $\xrightarrow{\text{ghb}}$. Indeed, the write serialisation for a given location ℓ is the order in which writes to ℓ are globally performed. Moreover, as $r \xrightarrow{\text{fr}} w$ expresses that the write w' from which r reads is globally performed before w, it forces the read r to be globally performed (since a read is globally performed as soon as it is performed) before w is globally performed.

Preserved program order In any given architecture, certain pairs of events in the program order are guaranteed to occur in that order. We postulate a global relation $\xrightarrow{\text{ppo}}$ gathering all such pairs. For example, the execution witness in Fig. 1(c) is only valid if the writes and reads to different locations on each processor have been reordered. Indeed, if these pairs were forced to be in program order, we would have a cycle in $\xrightarrow{\text{ghb}}$: $(a) \xrightarrow{\text{ppo}} (b) \xrightarrow{\text{fr}} (c) \xrightarrow{\text{ppo}} (d) \xrightarrow{\text{fr}} (a)$.

Barrier constraints Architectures also provide *barrier* instructions, *e.g.* the Power `sync` (discussed in Sec. 3) to enforce ordering between pairs of events. We postulate a global relation $\xrightarrow{\text{ab}}$ gathering all such pairs.

Architectures We call a particular model of our class an *architecture*, written A (or A^{ϵ} when $\xrightarrow{\text{ab}}$ is empty). Let *ppo* (resp. *grf*, *ab*, *ghb*) be the function returning the $\xrightarrow{\text{ppo}}$ (resp. $\xrightarrow{\text{grf}}$,$\xrightarrow{\text{ab}}$ and $\xrightarrow{\text{ghb}}$) relation when given an execution witness. Thus, we have:

$$A \triangleq (ppo, grf, ab)$$

We define $\xrightarrow{\text{ghb}}$ as the union of the global relations:

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

2.5 Validity of an execution *w.r.t.* an architecture

We now add two sanity conditions to the above.

Uniprocessor First, we require each processor to respect memory coherence for each location [16]. If a processor writes *e.g.* v to ℓ and then reads v' from ℓ, v' should not precede v in the write serialisation. We define the relation $\xrightarrow{\text{po-loc}}$ over accesses to the same location in the program order, and require $\xrightarrow{\text{po-loc}}$, $\xrightarrow{\text{rf}}$, $\xrightarrow{\text{ws}}$ and $\xrightarrow{\text{fr}}$ to be compatible (writing $\xrightarrow{\text{com}}$ for $\xrightarrow{\text{rf}} \cup \xrightarrow{\text{ws}} \cup \xrightarrow{\text{fr}}$):

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

$$\text{uniproc}(X) \triangleq \text{acyclic}(\xrightarrow{\text{com}} \cup \xrightarrow{\text{po-loc}})$$

For example, in Fig. 6 (a), the final value of `x` shows that the write (a) is the last one in coherence order. Since $\xrightarrow{\text{ws}}$ is a total order of the writes to `x`, we have $(c) \xrightarrow{\text{ws}} (a)$. Similarly, the final value of `r1` shows that the read (b) read its value from the write (a), hence $(a) \xrightarrow{\text{rf}} (b)$. The uniproc check on this program formalises the fact that the cycle $(a) \xrightarrow{\text{rf}} (b) \xrightarrow{\text{po-loc}} (c) \xrightarrow{\text{ws}} (a)$ invalidates this execution: (b) cannot read from (a) as it is a future value of `x` in $\xrightarrow{\text{ws}}$.

P_0	P_1
(a) `x` ← `1`	(b) `r1` ← `x`
	(c) `x` ← `2`

(a) uniproc Forbidden: `x=1; r1=1;`

P_0	P_1
(a) `r1` ← `x`	(c) `r4` ← `y`
`r9` ← `xor r1,r1`	`r9` ← `xor r4,r4`
(b) `y` ← `1+r9`	(d) `x` ← `1+r9`

(b) thin Forbidden: `r1=1; r4=1;`

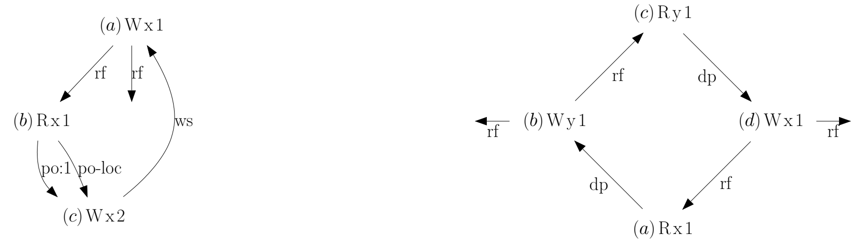


Fig. 6 Invalid executions according to the uniproc and thin criteria

Thin air Second, we rule out programs where values come *out of thin air* [23]. This means that we forbid certain *causal loops*, following the terminology employed in the Alpha documentation [10].

Consider the test given in Fig. 6 (b). In this example, the write (b) to y on P_0 is dependent on the read (a) from x on P_0, because the `xor` instruction between them does a calculation on the value written by (a) in `r1`, and writes the result into `r9`, later used by (b). Similarly on P_1, (c) and (d) are dependent. Suppose the read (a) from x on P_0 reads from the write (d) to x on P_1, and similarly the read (c) from y on P_1 reads from the write (b) to y on P_0, as depicted by the execution in Fig. 6 (b). In this case, the values read by (a) and (c) seem to come out of thin air, because they cannot be determined. We formalise the check that forbids such a scenario as follows:

$$\mathrm{thin}(X) \ \triangleq\ \mathrm{acyclic}(\xrightarrow{\mathrm{rf}} \cup \xrightarrow{\mathrm{dp}})$$

All together We define the validity of an execution *w.r.t.* an architecture A as the conjunction of three checks independent of the architecture, namely $\mathrm{wf}(X)$, $\mathrm{uniproc}(X)$ and $\mathrm{thin}(X)$ with a last one that characterises the architecture:

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

2.6 Comparing architectures *via* validity predicates

From our definition of validity arises a simple notion of comparison among architectures. $A_1 \le A_2$ means that A_1 is *weaker* than A_2:

$$A_1 \le A_2 \ \triangleq\ (\xrightarrow{\mathrm{ppo}_1} \subseteq \xrightarrow{\mathrm{ppo}_2}) \ \wedge\ (\xrightarrow{\mathrm{grf}_1} \subseteq \xrightarrow{\mathrm{grf}_2})$$

The validity of an execution is decreasing *w.r.t.* the strength of the predicate; *i.e.* a weak architecture exhibits at least all the behaviours of a stronger one:

Theorem 1 (Validity is decreasing)

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

Programs running on an architecture A_1^ϵ exhibit executions that would be valid on a stronger architecture A_2^ϵ; we characterise all such executions as follows:

$$A_1.\text{check}_{A_2}(X) \triangleq \text{acyclic}(\xrightarrow{\text{grf}_2} \cup \xrightarrow{\text{ws}} \cup \xrightarrow{\text{fr}} \cup \xrightarrow{\text{ppo}_2})$$

Then, we show that executions that are valid on A_1 and that satisfy this predicate are valid on A_2 and conversely:

Theorem 2 (Characterisation)

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

These two theorems, though fairly simple, will be useful to compare two models and to restore a strong model from a weaker one, as in Sec. 3.

2.7 Examples

We propose here alternative formulations of Sequential Consistency (SC) [21] and Sparc's Total Store Ordering (TSO) [30] in our framework, which we proved equivalent to the original definitions. We omit proofs and the formal details for brevity, but they can be found at `http://diy.inria.fr/wmm`. We write $\text{po}(X)$ (resp. $\text{rf}(X)$, $\text{rfe}(X)$) for the function extracting the $\xrightarrow{\text{po}}$ (resp. $\xrightarrow{\text{rf}}$, $\xrightarrow{\text{rfe}}$) relation from X. We define notations to extract pairs of memory events from the program order: $MM \triangleq \lambda X.\, ((\mathbb{M} \times \mathbb{M}) \cap \text{po}(X))$, $RM \triangleq \lambda X.\, ((\mathbb{R} \times \mathbb{M}) \cap \text{po}(X))$ and $WW \triangleq \lambda X.\, ((\mathbb{W} \times \mathbb{W}) \cap \text{po}(X))$.

SC allows no reordering of events ($\xrightarrow{\text{ppo}}$ equals $\xrightarrow{\text{po}}$ on memory events). In addition, SC makes writes available to all processors as soon as they are issued ($\xrightarrow{\text{rf}}$ are global). By this we mean that write events take immediately their place in the global-happens before relation, *i.e.* that once they become to one processor, they are visible to all processors. Thus, there is no need for barriers, and any architecture is weaker than SC:

$$\text{SC} \triangleq (MM, \text{rf}, \lambda X.\emptyset).$$

The following criterion characterises, as in Sec. 2.6, valid SC executions on any architecture: $A.\text{check}_{SC}(X) = \text{acyclic}(\xrightarrow{\text{com}} \cup \xrightarrow{\text{po}})$. Thus, the outcome of Fig. 1 will never be the result of an *SC* execution, as it exhibits the cycle: $(a) \xrightarrow{\text{po}} (b) \xrightarrow{\text{fr}} (c) \xrightarrow{\text{po}} (d) \xrightarrow{\text{fr}} (a)$.

TSO allows two relaxations [5]: *write to read program order*, meaning its $\xrightarrow{\text{ppo}}$ includes all pairs but the store-load ones ($ppo_{tso} \triangleq (\lambda X.\, (RM(X) \cup WW(X)))$ and *read own write early* ($\xrightarrow{\text{rfi}}$ are not global). We elide barrier semantics, detailed in Sec. 3: $TSO^\epsilon \triangleq (ppo_{tso}, \text{rfe}, \lambda X.\emptyset)$. Sec. 2.6 shows the following criterion characterises valid executions (*w.r.t.* any $A \leq TSO$) that would be valid on TSO^ϵ, *e.g.* in Fig. 1: $A.\text{check}_{TSO}(X) = \text{acyclic}(\xrightarrow{\text{ws}} \cup \xrightarrow{\text{fr}} \cup \xrightarrow{\text{rfe}} \cup \xrightarrow{\text{ppo-tso}})$.

3 Semantics of barriers

In this section we characterise the semantics and placement in the code that barriers should have to restore a stronger model from a weaker one.

It is clearly enough to have $w \xrightarrow{\text{ab}_1} r$ whenever $w \xrightarrow{\text{grf}_{2\setminus 1}} r$ holds to restore store atomicity, *i.e.* a barrier ensuring $\xrightarrow{\text{rf}}$ is global. But then a processor holding such a barrier placed after r would have to wait until w is globally performed before executing the read. We provide a less costly requirement: consider the case where $w \xrightarrow{\text{rf}} r \xrightarrow{\text{po}} m$, where r may take its value before w is visible to all processors. Inserting a barrier instruction that has our semantics between the instructions generating r and m only forces the processor generating r and m to delay m until w is globally performed.

We give here an intuition of the strength that the barriers of the architecture A_1 should have to restore the stronger A_2. They should:

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

Formally, we write $\xrightarrow{\text{r}_{2\setminus 1}} \triangleq \xrightarrow{\text{r}_2} \setminus \xrightarrow{\text{r}_1}$ for the set difference. In addition, we write $x \xrightarrow{\text{r}_1};\xrightarrow{\text{r}_2} y \triangleq \exists z. x \xrightarrow{\text{r}_1} z \wedge z \xrightarrow{\text{r}_2} y$ for the sequence of two relations. Given $A_1 \leq A_2$, we define the predicate fb (*fully barriered*) on executions X by

$$A_1.\text{fb}_{A_2}(X) \triangleq \big((\xrightarrow{\text{ppo}_{2\setminus 1}}) \cup (\xrightarrow{\text{grf}_{2\setminus 1}};\xrightarrow{\text{ppo}_2})\big) \subseteq \xrightarrow{\text{ab}_1}$$

We can then prove that the above condition on $\xrightarrow{\text{ab}_1}$ is sufficient to regain A_2^{ϵ} from A_1:

Theorem 3 (Barrier guarantee)

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

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

The dynamic property of barriers is expressed by the condition $\xrightarrow{\text{grf}_{2\setminus 1}};\xrightarrow{\text{ppo}_2} \subseteq \xrightarrow{\text{ab}_1}$. A barrier provided by A_1 should ensure store atomicity to the write events that have this property on A_2. This is how we interpret the *cumulativity* of barriers, as stated by Power [25], in our framework: the *A-cumulativity* (resp. *B-cumulativity*) property applies to barriers that enforce ordering of pairs in $\xrightarrow{\text{rf}};\xrightarrow{\text{po}}$ (resp. $\xrightarrow{\text{po}};\xrightarrow{\text{rf}}$). We consider a barrier that only preserves pairs in $\xrightarrow{\text{po}}$ to be *non-cumulative.* Thm. 3 states that, to restore A_2 from A_1, it suffices to insert an A-cumulative barrier between each pair of instructions such that the first one in the program order reads from a write which is to be globally performed on A_2 but is not on A_1.

iriw

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

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

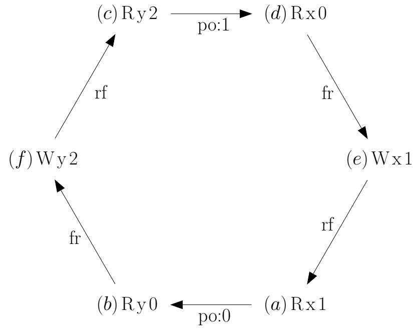


Fig. 7 The **iriw** test and a non-SC execution

Restoring SC We model an A-cumulative barrier as a function returning an ordering relation when given a placement of the barriers in the code:

$$\begin{aligned} m_1 \overset{\text{fenced}}{\rightarrow} m_2 &\triangleq \exists b.\, m_1 \overset{\text{po}}{\rightarrow} b \overset{\text{po}}{\rightarrow} m_2 \\ \text{A-cumul}(X, \overset{\text{fenced}}{\rightarrow}) &\triangleq \overset{\text{fenced}}{\rightarrow} \cup \overset{\text{rf}}{\rightarrow}; \overset{\text{fenced}}{\rightarrow} \end{aligned}$$

The following corollary of Thm. 3 (with $A_1 = A$ and $A_2 = SC$) shows that inserting such a barrier between all $\overset{\text{po}}{\rightarrow}$ pairs restores SC:

Corollary 1 (Barriers restoring SC)

$$\forall A\ X, (A.\text{valid}(X) \wedge \textit{A-cumul}(X, \textit{MM}) \subseteq \overset{ab}{\rightarrow}) \Rightarrow SC.\text{valid}(X)$$

Consider *e.g.* the **iriw** test depicted in Fig. 7. The specified outcome may be the result of a non-SC execution on a weak architecture in the absence of barriers, as shown in Fig. 7. Our A-cumulative barrier placed between each pair of reads on P_0 and P_1 forbids this outcome, as shown in Fig. 8. The non-cumulative property of the barrier (expressed formally by $\overset{\text{fenced}}{\rightarrow}$ being included in the definition of A-cumulative) ensures that none of the two pairs of reads can be reordered. The A-cumulative property of the barrier (expressed formally by $\overset{\text{rf}}{\rightarrow}; \overset{\text{fenced}}{\rightarrow}$ being included in the definition of A-cumulative) also ensures that the write (e) on P_2 (resp. (y) P_3) is globally performed before the second read (b) on P_0 (resp. (d) on P_1).

Thus, we force a program to have an SC behaviour by fencing all pairs in $\overset{\text{po}}{\rightarrow}$. Yet, it would be enough to invalidate non-SC executions, by fencing only the $\overset{\text{po}}{\rightarrow}$ pairs in the $\overset{\text{com}}{\rightarrow} \cup \overset{\text{po}}{\rightarrow}$ cycles of these executions. We believe the static analysis of [29] (based on compile-time approximation of $\overset{\text{com}}{\rightarrow} \cup \overset{\text{po}}{\rightarrow}$ cycles) applies to architectures relaxing store atomicity, if their barriers offer A-cumulativity.

iriw

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

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

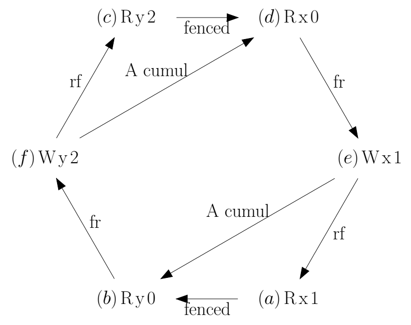


Fig. 8 Study of **iriw** with A-cumulative barriers

4 diy: a testing tool

We now present our diy (*do it yourself*) tool, which generates litmus tests in x86 or Power assembly code by enumerating possible violations of *SC* in terms of our axiomatic model, *i.e.*, cycles in $\xrightarrow{\text{com}} \cup \xrightarrow{\text{po}}$. A diy tutorial is available at `http://diy.inria.fr`.

4.1 Informal overview of our method

We start here by explaining the idea of the method employed by diy at a high level, then give further details in the following subsections.

As we described in the previous section, the outcome of the **iriw** test in Fig. 7 leads to the $\xrightarrow{\text{com}} \cup \xrightarrow{\text{po}}$ cycle depicted there. Conversely, that test can be *built* from the sequence $\xrightarrow{\text{rfe}};\xrightarrow{\text{po}};\xrightarrow{\text{fre}};\xrightarrow{\text{rfe}};\xrightarrow{\text{po}};\xrightarrow{\text{fre}}$. The specified outcome ensures that the input cycle appears in at least one of the execution witnesses of the test. If that outcome is observed, then, in terms of our framework, at least one subsequence in the cycle is not global, *i.e.* not in $\xrightarrow{\text{ghb}}$: in the case of Fig. 7, either the $\xrightarrow{\text{po}}$ or the $\xrightarrow{\text{rfe}}$ relations might not be included in $\xrightarrow{\text{ghb}}$.

More precisely, a global cycle in an execution graph forbids the specified outcome of a test in the model, as we explained in Sec. 2. This means that if a machine implements exactly the studied model, then the outcome of this test should not be observed. To check this, we thus build tests from cycles in execution graphs. If the outcome of a test is observed, we conclude that the cycle from which it is generated does not actually forbid the outcome.

In addition, to make the analysis of the results of such tests feasible, we generate our cycles so that there can be only one possible reason—in the model—why the outcome can be exhibited. For example, in the cycle $\overset{\text{rfe}}{\rightarrow};\overset{\text{po}}{\rightarrow};\overset{\text{fre}}{\rightarrow};\overset{\text{rfe}}{\rightarrow};\overset{\text{po}}{\rightarrow};\overset{\text{fre}}{\rightarrow}$, there can be several reasons why this cycle is not global: either $\overset{\text{rfe}}{\rightarrow}$ is not global, or the $\overset{\text{po}}{\rightarrow}$ relation between two reads is not preserved, or both. Hence, if we want to state precisely why a given outcome is exhibited, we need to pick one relation to test, for example $\overset{\text{rfe}}{\rightarrow}$, and build a cycle where $\overset{\text{rfe}}{\rightarrow}$ is the only non-global relation possible.

Of course, this practice supposes that the other relations from which a cycle is built are global. We check this assumption by generating cycles where all the relations are global in the model, and run them to check that their outcomes are not observed.

4.2 Candidate Relaxations

We wrote the diy testing tool to automatically generate litmus tests that exercise relations specified by the user. When given a certain sequence of relations, diy produces tests such that one of their executions contains at least one occurrence of the given sequence. Hence, if we want to check whether the external read-from maps are relaxed on a given machine, we specify $\overset{\text{rfe}}{\rightarrow}$ to be relaxed to diy, following the concrete syntax we give in Fig. 9.

We write Po for a program order candidate relaxation and Dp for a dependency. We handle the communication relaxations as follows: we write Rf for a read-from, Ws for a write serialisation, Fr for a from-read. We also deal with barrier candidate relaxation: thus, we write Fence for a non-cumulative barrier, ACFence for an A-cumulative one, BCFence for a B-cumulative one, and ABCFence for the sequence of an A- and a B-cumulative ones.

We specify if the two accesses related by the candidate relaxation access the same location by the letter s; we use the letter d if they access different locations. We specify the directions of the accesses related by the candidate relaxation, using W for write and R for read. Note that in the case of a dependency relation Dp, we just need to specify the direction of the target access, since dependency candidate relaxations always have a read as their source, following the definition of the $\overset{\text{dp}}{\rightarrow}$ relation given in Sec. 2.2.

For communication candidate relaxations such as Rf, Fr or Ws, we specify whether the candidate relaxation is internal (resp. external)—*i.e.* relating two accesses that belong to the same processor (resp. distinct processors)— by the letter i (resp. e).

Thus Rfe represents a $\overset{\text{rfe}}{\rightarrow}$ arrow and Fre a $\overset{\text{fre}}{\rightarrow}$ arrow. The candidate relaxation DpdR should be read as (1) Dp, which means that we generate a $\overset{\text{dp}}{\rightarrow}$ arrow, (2) R, which means that this arrow targets a read, and (3) d, which means that the two accesses have different source and target locations. For Power, we instantiate the barrier candidate relaxation Fence with either Sync or LwSync.

Note that some of the candidate relaxations might be redundant. For example PosWW (two write events to the same location in program, hence on the same processor) is a particular case of Wsi (internal write serialisation). Yet, we actually care for this redundancy, for two reasons. First, this helps us covering all possibilities in enumerating test cases for model exploration. Second, it helps us to spot precisely the reason why a given test might reveal a bug in a hardware implementation.

Name	Notation
program order	Po(s|d)(W|R)(W|R)
dependencies	Dp(s|d)(W|R)
read-from map	Rf(i|e)
write serialisation	Ws(i|e)
from-read map	Fr(i|e)
barriers	Fence(s|d)(W|R)(W|R)
A-cumulativity	ACFence$ld_1 d_2 \triangleq$ [Rfe; Fence$ld_1 d_2$]
B-cumulativity	BCFence$ld_1 d_2 \triangleq$ [Fence$ld_1 d_2$; Rfe]
AB-cumulativity	ABCFence$ld_1 d_2 \triangleq$ [Rfe; Fence$ld_1 d_2$; Rfe]

Fig. 9 Table of Candidate Relaxations

```
#rfe PPC conf file
-arch PPC
-nprocs 4
-size 6
-name rfe
-safe Fre DpdR
-relax Rfe
```

Fig. 10 Example Configuration File for diy

4.3 Input and Output of the diy tool

In practice, the diy tool takes as input a configuration file such as the one in Fig. 10. This configuration file forces diy to generate tests in Power assembly up to 4 processors, as specified by the `-arch PPC` and `-nprocs 4` arguments, so that the number of relations involved in the generated cycles is 6 at most, because of the `-size 6` argument. Moreover, the candidate relaxations Fre (external from-read map) and DpdR (data dependency between two reads from distinct locations) are considered global, and Rfe is considered relaxed, as specified by the `-safe Fre DpdR` and `-relax Rfe` arguments. Finally, all the tests generated by diy running on this configuration file will have the prefix rfe in their name, followed by a fresh number, as specified by the `-name rfe` argument.

The tool outputs x86 or Power assembly tests. More precisely, the tool internally generates cycles from the candidate relaxations supplied as argument to the `-safe` and `-relax` specifications, up to the specified bounds on cycle length and number of processors. Each cycle then commands the generation of one test.

4.4 Exercising One Relaxation at a Time

So as to make the analysis of the testing results feasible, we focus on tests which exercise a single weakness of the memory model at a time. Hence, if the outcome of a given test is exhibited, we know that the feature we tested is used by the machine on which we ran the test.

For example, suppose that we modify the test of Fig. 7 and impose dependencies between the pairs of reads on P_0 and P_1, so that these dependencies are global, *e.g.* by being included in $\xrightarrow{\text{ppo}}$. Hence (a) is in data dependency with (b),

```
{ 0:r2=y; 0:r5=x; 1:r2=x; 2:r2=x; 2:r5=y; 3:r2=y; }

 P0                | P1                | P2                | P3
(c) lwz r1,0(r2)   |     li r1,1       | (a) lwz r1,0(r2)  |     li r1,1
    xor r3,r1,r1   | (e) stw r1,0(r2)  |     xor r3,r1,r1  | (f) stw r1,0(r2)
(d) lwzx r4,r3,r5  |                   | (b) lwzx r4,r3,r5 |

exists (0:r1=1 /\ 0:r4=0 /\ 2:r1=1 /\ 2:r4=0)
```

Fig. 11 **iriw** with dependencies in Power assembly

and so is (c) with (d), resulting in the cycle $\xrightarrow{\text{rfe}};\xrightarrow{\text{ppo}};\xrightarrow{\text{fre}};\xrightarrow{\text{rfe}};\xrightarrow{\text{ppo}};\xrightarrow{\text{fre}}$. This corresponds to the test given in Fig. 11, written in Power assembly code. The `xor r3,r1,r1` between the load (b) and the load (c) on P_0 implements such a dependency.

In this case the only reason why the specified outcome may arise is the non-globality of external read-from maps. Hence, the test of Fig. 11 is significant to check whether an architecture relaxes the atomicity of stores. More generally, we say that a test built from a cycle containing global relations only and a relation r *targets* r, because it can exhibit its outcome only because of r being relaxed. We shall also make use of tests built from global relations only, such a test is called a *safe* test.

4.5 Cycle Generation

The input to diy must specify which candidate relaxations are to be assumed *not* relaxed (considered global, or *safe*) and which are to be investigated. When given a pool of safe candidate relaxations, a single potential relaxation to be investigated, and a size n (*i.e.* the number of arrows in the cycle, *e.g.* 6 for the **iriw** test of Fig. 7), diy generates cycles up to size n that contains at least one occurrence of the non-global candidate relaxation. If no non-global candidate relaxation is specified, diy generates cycles up to size n that contain the specified global candidate relaxations.

We do not generate tests for all these cycles: we eliminate some sequences of candidate relaxations. First, we eliminate any sequence of two candidate relaxations when the target of the first one is incompatible with the source of the second one: for example, we eliminate a sequence $\xrightarrow{\text{Rfe}};\xrightarrow{\text{Rfe}}$ because the target of the first $\xrightarrow{\text{Rfe}}$ is a read, whereas the source of the second one is a write. Notice that no assembly program exists that corresponds to such cycles. However, eliminating impossible sequences early accelerates the production of cycles. More significantly, diy eliminate some additional sequences of candidate relaxations:

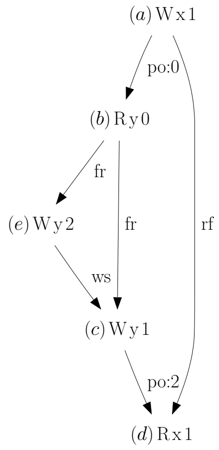
1. For communication candidate relaxations (*i.e.* Ws, Fr and Rf), we do not generate the sequence [Ws_;Ws_] for it reduces to Ws_ in the model. Similarly, we do not generate the sequence [Ws_;Fr_], for it reduces to Fr_ in the model. Hence we generate only five sequences of communication candidate relaxations, namely Ws_, Fr_, Rf_, [Ws_;Rf_] and [Fr_;Rf_].
2. For program order candidate relaxations, we do not generate sequences [po_1;po_2] when the sequence is subsumed by another safe candidate relaxation (where po_1 and po_2 range over Po, Dp, Fence, and internal communication (*i.e.* Rfi, Fri and Wsi)). This means for example that, if FencedWR and Fence_WW

Init: `x=0; y=0;`

P_0	P_1	P_2
(a) `x` ← `1` (b) `r1` ← `y`	(e) `y` ← `2`	(c) `y` ← `1` (d) `r2` ← `x`

Observed? `r1=0; r2=0;`

(a) Program



(b) An execution witness

Fig. 12 A program and a candidate execution

are safe, we do not generate [FencedWR;PodRW], for it reduces to Fence_WW (assumed safe).

As a result, we do not generate all cycles up to a given size. However, the executions associated to the cycles we discard (for example a cycle involving a sequence [Fr_;Ws_] feature other shorter cycles (the cycle where [Fr_;Ws_] is reduced to Fr_), which are generated.

Consider for example the test given in Fig. 12(a), which corresponds to the test of Fig. 1(a), where we added a new processor holding the store (e) `y` ← `2`. In this case, the final state of interest to us is still the one specified in Fig. 1, *i.e.* `r1=0; r2=0;`. This final state corresponds to the execution witness given in Fig. 12(b). More precisely, this final state corresponds to the cycle $(a) \xrightarrow{\text{PodWR}} (b) \xrightarrow{\text{Fre}} (e) \xrightarrow{\text{Wse}} (c) \xrightarrow{\text{PodWR}} (d) \xrightarrow{\text{Fre}} (a)$.

Observe that the read from y at line (b) on P_0 is in $\overset{\text{fr}}{\to}$ with the write (e) to y on P_2, because of the final value of register `r1`. Moreover, in this particular execution, the write (e) to y on P_1 is in $\overset{\text{ws}}{\to}$ before the write (c) to y on P_2. Therefore, by definition of $\overset{\text{fr}}{\to}$ and since $\overset{\text{ws}}{\to}$ is transitive, we know that $(b) \overset{\text{fr}}{\to} (c)$ as well. Thus, the final state corresponds to the shorter cycle $(a) \xrightarrow{\text{PodWR}} (b)[\xrightarrow{\text{Fre}};\xrightarrow{\text{Wse}}](c) \xrightarrow{\text{PodWR}} (d) \xrightarrow{\text{Fre}} (a)$, *i.e.* $(a) \xrightarrow{\text{PodWR}} (b) \xrightarrow{\text{Fre}} (c) \xrightarrow{\text{PodWR}} (d) \xrightarrow{\text{Fre}} (a)$, from which we generate the test of Fig. 1(a). Thus in such a case, we would generate only the test of Fig. 1(a), and not the test of Fig. 12(a).

4.6 Code Generation

diy interprets a sequence of candidate relaxations as a cycle from which it either computes a litmus test or fails. The final condition of a generated test is a conjunction of equalities on the values held by registers and memory locations in the final state, which ensures that at least one of the execution witnesses of this test includes a cycle following the input sequence.

4.6.1 Algorithm

Test generation performs the following successive steps:

1. We map the edge sequence to a circular double-linked list. The cells represent memory events, with direction, location, and value fields, together with the edge starting from the event. This list represents the *input cycle* and will appear in at least one of the execution witnesses of the produced test.
2. A linear scan sets the directions (write or read) of the events, by comparing each target direction with the following source direction. When they are equal, the in-between cell direction is set to the common value; otherwise (*e.g.* Rfe; Rfe), the generation fails.
3. We pick an event e which is the target of a candidate relaxation specifying a location change. If there are none, the generation fails. Otherwise, a linear scan starting from e sets the locations of each event. At the end of the scan, if e and its predecessor have the same location (*e.g.* $\xrightarrow{\text{Rfe}} e \xrightarrow{\text{PodRW}}$), the generation fails, since we picked e to correspond to a location change.
4. We cut the input cycle into maximal sequences of events with the same location, each being scanned *w.r.t.* the cycle order: we give the value 1 to the first write in this sequence, 2 to the second one, *etc.* For each location in cycle, the sequence of values 0, 1, *etc.* defines a certain write serialisation order, which the final condition of test will characterise (step 7 below).
5. We define *significant reads* as the sources of $\xrightarrow{\text{fr}}$ edges and the targets of $\xrightarrow{\text{rf}}$ edges. We associate each significant read with the write on the other side of the edge. In the $\xrightarrow{\text{rf}}$ case, the value of the read is the one of its associated write. In the $\xrightarrow{\text{fr}}$ case, the value of the read is the value of the predecessor of its associated write in $\xrightarrow{\text{ws}}$, *i.e.* by construction the value of its associated write minus 1 (see step 4). Non-significant reads do not appear in the test condition.
6. We cut the cycle into maximal sequences of events from the same processor, each being scanned, generating load instructions to (resp. stores from) fresh registers for reads (resp. writes). We insert some code implementing a dependency in front of events targeting $\xrightarrow{\text{dp}}$ and the appropriate barrier instruction for events targeting $\xrightarrow{\text{fenced}}$ edges. We build the initial state at this step: stores and loads take their addresses from fresh registers, and their contents (addresses of memory locations) are defined in the initial state. Part of the final condition is also built: for any significant read with value v resulting in a load instruction to register r, we add the equality $r = v$.
7. We complete the final condition to characterise write serialisations. The write serialisation for a given location x is defined by the sequence of values 0 (initial value of x), …, n, where n is the last value allocated for location x at step 4. If n is 0 or 1 then no addition to the final condition needs to be performed, because the write serialisation is either a singleton or a pair. If n is 2, we add the equality $x = 2$. Otherwise ($n > 2$), we add an *observer* to the program, *i.e.* we add a thread performing n loads from x to registers `r1`, …, `rn` and add the equalities $\texttt{r1} = 1 \wedge \ldots \wedge \texttt{rn} = n$ to the final condition.

4.6.2 Example

We show here how to generate a Power litmus test from a given cycle of candidate relaxations by an example. We write $\_$ for the information not yet set by diy: $\_\_\_$ is an undetermined event, $W\_\_$ a write with as-yet unset location and value, and $Rx\_$ a read from x with undetermined value.

1. Consider *e.g.* the input cycle, issued by diy's cycles generation phase, with the input being the configuration file given in Fig. 10:
$$(a)\_\_\_ \xrightarrow{\text{Rfe}} (b)\_\_\_ \xrightarrow{\text{DpdR}} (c)\_\_\_ \xrightarrow{\text{Fre}} (d)\_\_\_ \xrightarrow{\text{Rfe}} (e)\_\_\_ \xrightarrow{\text{DpdR}} (f)\_\_\_ \xrightarrow{\text{Fre}} (a)$$
2. A linear scan sets the directions from the edges. Observe *e.g.* the last edge; $\xrightarrow{\text{Fre}}$ requires a R source and a W target:
$$(a)W\_\_ \xrightarrow{\text{Rfe}} (b)R\_\_ \xrightarrow{\text{DpdR}} (c)R\_\_ \xrightarrow{\text{Fre}} (d)W\_\_ \xrightarrow{\text{Rfe}} (e)R\_\_ \xrightarrow{\text{DpdR}} (f)R\_\_ \xrightarrow{\text{Fre}} (a)$$
3. As $\xrightarrow{\text{DpdR}}$ specifies a location change, we pick (c) to be the first event and rewrite the cycle as:
$$(c)R\_\_ \xrightarrow{\text{Fre}} (d)W\_\_ \xrightarrow{\text{Rfe}} (e)R\_\_ \xrightarrow{\text{DpdR}} (f)R\_\_ \xrightarrow{\text{Fre}} (a)W\_\_ \xrightarrow{\text{Rfe}} (b)R\_\_ \xrightarrow{\text{DpdR}} (c)$$
We set the locations starting from (c), with a change of location *e.g.* between (e) and (f) since $\xrightarrow{\text{DpdR}}$ specifies a location change:
$$(c)Rx\_ \xrightarrow{\text{Fre}} (d)Wx\_ \xrightarrow{\text{Rfe}} (e)Rx\_ \xrightarrow{\text{DpdR}} (f)Ry\_ \xrightarrow{\text{Fre}} (a)Wy\_ \xrightarrow{\text{Rfe}} (b)Ry\_ \xrightarrow{\text{DpdR}} (c)$$
4. We cut the input cycle into maximal sequences of events with the same location (*i.e.* $(c)(d)(e)$ and $(f)(a)(b)$), each being scanned *w.r.t.* the cycle order. The values then reflect the write serialisation order for the specified location:
$$(c)Rx\_ \xrightarrow{\text{Fre}} (d)Wx1 \xrightarrow{\text{Rfe}} (e)Rx\_ \xrightarrow{\text{DpdR}} (f)Ry\_ \xrightarrow{\text{Fre}} (a)Wy1 \xrightarrow{\text{Rfe}} (b)Ry\_ \xrightarrow{\text{DpdR}} (c)$$
5. All the reads are significant here; we set their values according to step 5:
$$(c)Rx0 \xrightarrow{\text{Fre}} (d)Wx1 \xrightarrow{\text{Rfe}} (e)Rx1 \xrightarrow{\text{DpdR}} (f)Ry0 \xrightarrow{\text{Fre}} (a)Wy1 \xrightarrow{\text{Rfe}} (b)Ry1 \xrightarrow{\text{DpdR}} (c)$$
6. We generate the litmus test given in Fig. 11 for Power according to the steps 6 and 7 given in Sec. 4.6.1. For example on P_0, we add a `xor` instruction between the instructions `lwz r1,0(r2)` and `lwzx r4,r3,r5` associated with the events (b) and (c) to implement the dependency required by the $\xrightarrow{\text{DpdR}}$ relation between them. The events (d) and (e), associated respectively to `stw r1,0(r2)` on P_1 and `lwz r1,0(r2)` on P_2, are specified in the cycle to be in $\overset{\text{rfe}}{\to}$. Hence, we specify in the final state that the register `r1` on P_2 holds finally 1. Indeed the store associated with (d) writes 1 into the address x addressed by `r2` on P_1 , since the contents of the register `r1` on P_1 is 1 (because of the preceding `li r1, 1` instruction). Since $(d) \overset{\text{rfe}}{\to} (e)$, the load associated with (e) on P_2 reads the value 1 from the address x addressed by `r2`, and writes 1 into the register `r2`.

The test in Fig. 11 is a Power implementation of **iriw** [14] with dependencies. It can be obtained by running diy on the configuration file given in Fig. 10.

4.7 Coverage

Given a test generation tool such as diy, one should ask in what sense it provides good coverage and whether it generates useful tests.

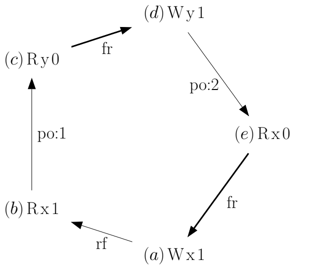
One way to assess coverage is to check that it can generate 'classical' litmus tests in the literature. We just explained how one can generate the **iriw**

rwc

P_0	P_1	P_2
(a) `x` ← `1`	(b) `r1` ← `x` (c) `r2` ← `y`	(d) `y` ← `1` (e) `r3` ← `x`

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

(a) The **rwc** test



(b) An Execution Witness

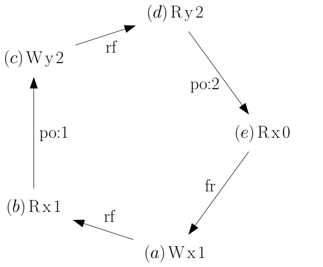
Fig. 13 The **rwc** Test and a Candidate Execution

wrc

P_0	P_1	P_2
(a) `x` ← `1`	(b) `r1` ← `x` (c) `y` ← `2`	(d) `r1` ← `y` (e) `r2` ← `x`

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

(a) The **wrc** test



(b) An Execution Witness

Fig. 14 The **wrc** Test and a Candidate Execution

from [14]. The **rwc** test of [14], given in Fig. 13(a) can be obtained from the cycle $\xrightarrow{\text{Rfe}};\xrightarrow{\text{PodRR}};\xrightarrow{\text{Fre}};\xrightarrow{\text{PodWR}};\xrightarrow{\text{Fre}}$, as one can deduce from the execution given in Fig. 13(b). The **wrc** test of [14], given in Fig. 14(a) can be obtained from the cycle $\xrightarrow{\text{Rfe}};\xrightarrow{\text{PodRW}};\xrightarrow{\text{Rfe}};\xrightarrow{\text{PodRR}};\xrightarrow{\text{Fre}}$, as shown by the execution of Fig. 14(b).

Further, it generates all the variations of such tests with different choices of barriers and dependencies (e.g. 56 variations of **wrc**), and several families of tests that we have not previously seen. There are some interesting classes of tests that it does not generate, *e.g.* tests exhibiting observable register shadowing [3], where we still rely on hand-written tests.

In general, diy will be able to generate any classical litmus test, as soon as this test can be generated from a cycle of candidate relaxations as defined in Sec. 4.2, and is not subject to the restrictions described in Sec. 4.5.

Comparing directly with the tests used within processor vendors is difficult, as those are commercially sensitive. However, the fact that we have found issues in deployed processors (as detailed in Sec. 5) is indicative.

Another sense in which it is demonstrably useful (indeed, indispensable) in practice has been in our model-building work, as illustrated for example in Sec. 6.2. Our initial explorations relied on several hundred hand-written tests, and it was hard to maintain consistency and ensure coverage of these. Using diy, while we still use some hand-written tests, most of our work can be done with automatically

Relaxation	Definition	`hpcx`	`squale`	`vargas`
PosRR	$r_\ell \overset{\mathrm{po}}{\to} r'_\ell$	2/40M	3/2M	0/4745M
PodRR	$r_\ell \overset{\mathrm{po}}{\to} r'_{\ell'}$	2275/320M	12/2M	0/16725M
PodRW	$r_\ell \overset{\mathrm{po}}{\to} w'_{\ell'}$	0/6000M	0/6000M	0/6000M
PodWW	$w_\ell \overset{\mathrm{po}}{\to} w'_{\ell'}$	2029/4M	2299/2M	2092501/32M
PodWR	$w_\ell \overset{\mathrm{po}}{\to} r'_{\ell'}$	51085/40M	178286/2M	672001/32M
Rfi	$\overset{\mathrm{rfi}}{\to}$	7286/4M	1133/2M	145/32M
Rfe	$\overset{\mathrm{rfe}}{\to}$	177/400M	0/1776M	9/32M
LwSyncsWR	$w_\ell \overset{\mathrm{lwsync}}{\to} r'_\ell$	243423/600M	2/40M	385/32M
LwSyncdWR	$w_\ell \overset{\mathrm{lwsync}}{\to} r'_{\ell'}$	103814/640M	11/2M	117670/32M
ACLwSyncsRR	$w_\ell \overset{\mathrm{rfe}}{\to} r'_\ell \overset{\mathrm{lwsync}}{\to} r''_\ell$	11/320M	0/960M	1/21M
ACLwSyncdRR	$w_\ell \overset{\mathrm{rfe}}{\to} r'_\ell \overset{\mathrm{lwsync}}{\to} r''_{\ell'}$	124/400M	0/7665M	2/21M
BCLwSyncsWW	$w_\ell \overset{\mathrm{lwsync}}{\to} w'_\ell \overset{\mathrm{rfe}}{\to} r''_\ell$	68/400M	0/560M	2/160M
BCLwSyncdWW	$w_\ell \overset{\mathrm{lwsync}}{\to} w'_{\ell'} \overset{\mathrm{rfe}}{\to} r''_{\ell'}$	158/400M	0/11715M	1/21M

Fig. 15 Selected Results of the diy Experiment Matching Our Model

generated tests. In addition, as we explain in our more recent work on modelling the behaviour of Power multiprocessors [27], some key issues of the model were identified with these automatically generated tests.

5 Using diy: The Phat Experiment

We ran a case study for the diy tool, the Phat Experiment, from December 2009 to January 2010. We tested three Power machines, and present here a summary of the experimental results. More details can be found online at `http://diy.inria.fr/phat`.

5.1 Relaxations Observed on `squale`, `vargas` and `hpcx`

We used diy to generate 800 Power tests and run them up to 10^{12} times each on three machines: `squale`, a 4-processor Power G5 running Mac OS X; `hpcx`, a Power 5 with 16 processors per node and `vargas`, a Power 6 with 32 processors per node, both of them running AIX.

We ran the tests supposed to exhibit relaxations, *i.e.* the tests targeting any possible relation of Fig. 9 that our CAV 2010 model (see Fig. 22) does not include in $\overset{\mathrm{ghb}}{\to}$. We observed all of them at least on one machine, except PodRW. Not observing a given candidate relaxation does not contradict our model, since our model should authorise at least all the behaviours that we observed on hardware. We give in Fig. 15 the number of times the outcome was observed (where M stands for million). For each relaxation observed on a given machine, we write the highest number of outcomes. When a candidate relaxation was not observed, we write the total of outcomes: thus we write *e.g.* 0/16725M for PodRR on `vargas`.

For a given candidate relaxation, we generated tests with diy by writing a simple configuration file setting its relax list to this candidate relaxation, and some of the candidate relaxations that we considered to be safe.

Cycle	`hpcx`	In [14]
Rfe SyncdRR Fre Rfe SyncdRR Fre	2/320M	iriw
Rfe SyncdRR Fre SyncdWR Fre	3/400M	rwc
DpdR Fre Rfe SyncsRR DpdR Fre Rfe SyncsRR	1/320M	
Wse LwSyncdWW Wse Rfe SyncdRW	1/800M	
Wse SyncdWR Fre Rfe LwSyncdRW	1/400M	

Fig. 16 Anomalies Observed on Power 5

We did not observe the PodRW relaxation; but the documentation does not specify this candidate relaxation to be safe, therefore we still consider it to be relaxed.

5.2 Safe Relaxations

Following our informal model, we assumed that the candidate relaxations corresponding to $\xrightarrow{\text{ws}}$, $\xrightarrow{\text{fr}}$, dependencies and barriers were global and tested this assumption by computing *safe* tests in which the input cycles only include candidate relaxations that we supposed global, *e.g.* $\xrightarrow{\text{SyncdWW}};\xrightarrow{\text{Wse}};\xrightarrow{\text{SyncdWR}};\xrightarrow{\text{Fre}}$.

For each machine, we observed the number of runs required to exhibit the least frequent relaxation (*e.g.* 160M for BCLwSyncsWW on `vargas`), and ran the safe tests at least 20 times this number. The outcomes of the safe tests have not been observed on `vargas` and `squale`, which increases our confidence in the safe set we assumed.

However, `hpcx` does exhibit non-*SC* behaviours for some A-cumulativity tests (albeit rarely), including classical tests [14] such as **iriw** with `sync` instructions on P_0 and P_1. These results are in contradiction with our model. We summarise these contradictions in Fig. 16.

We understand that this is due to a rare erratum in the Power 5 implementation. IBM is providing a software workaround, replacing the `sync` barrier by a short code sequence [Personal Communication], and our testing suggests that this does regain *SC* behaviour for the examples in question (e.g. with 0/4e10 non-*SC* results for **iriw**). We understand also that Power 6 is not subject to the erratum, which is consistent with our testing on `vargas`, and that it should not affect the correctness of code using conventional lock primitives.

6 The CAV 2010 Axiomatic Power Model

In the light of the black-box experimental testing described in Sec. 5, we have instantiated the formalism of Sec. 2 for Power. The resulting model (which we refer to as the *CAV 2010* model) captures several important aspects of the processor's behaviour:

- it describes the lack of *store atomicity* on Power, despite being in the simple global-time style of our framework rather than the per-processor- timeline style implied by the architecture text;

Name	Notation	Comment	Sec.
intra instruction causality order	$m_1 \overset{\text{iico}}{\rightarrow} m_2$	constraints arising from instruction semantics	6.1.1
register read-from map	$w \overset{\text{rf-reg}}{\rightarrow} r$	links a register write to a register read reading its value	6.1.1
data dependency	$m_1 \overset{\text{dd}}{\rightarrow} m_2$	$(m_1, m_2) \in (\overset{\text{rf-reg}}{\rightarrow} \cup \overset{\text{iico}}{\rightarrow})^+$	6.2.1
control dependency	$r \overset{\text{ctrl}}{\rightarrow} w$	read-write pair separated by a branch	6.2.1
isync dependency	$r \overset{\text{isync}}{\rightarrow} m$	read-read or read-write pair separated by a branch+isync sequence	6.2.1
dependency	$m_1 \overset{\text{dp}}{\rightarrow} m_2$	$(m_1, m_2) \in \overset{\text{ctrl}}{\rightarrow} \cup \overset{\text{isync}}{\rightarrow} \cup ((\overset{\text{dd}}{\rightarrow} \cup (\overset{\text{po-loc}}{\rightarrow} \cap (\mathbb{W} \times \mathbb{R})))^+ \cap (\mathbb{R} \times \mathbb{M}))$	6.2.1
sync	$m_1 \overset{\text{sync}}{\rightarrow} m_2$	pairs maintained by `sync`	6.2.3
lwsync	$m_1 \overset{\text{lwsync}}{\rightarrow} m_2$	pairs maintained by `lwsync`	6.2.3

Fig. 17 Table of Power specific relations

- it also models the thread-local *ordering relaxations* we observe; and
- it models *A-cumulative barriers* [25].

The model is sound (modulo the anomalies described in the previous section) with respect to all our experimental results for Power G5, 5, and 6. However, for some of the Power barriers it is weaker than one might like, and for some examples it appears to be stronger than the architectural intent. We discuss this in detail in the next section. Moreover, being primarily based on black-box testing, its relationship to microarchitectural views or the architecture specifications of the processors is less clear than one might like. Accordingly, we do not regard it as definitive, but as a necessary step towards more definitive models.

6.1 Auxiliary Definitions

To define the CAV 2010 model (more specifically, to specify its preserved program order relation), we first need some auxiliary definitions to describe how dependencies and barriers arise from the instruction semantics.

We give in Fig. 18 a table of the fragment of the Power instruction set that we use to describe the forthcoming examples.

6.1.1 Register Events

We first add *register events* to reflect register accesses [26]. Loads and stores now yield additional register events, as depicted in Fig. 19.

For example, consider two registers `r1` and `r2`, such that `r1` initially holds the value 0, `r2` initially holds an address x, and x holds the value 1. In this case, an instruction `lwz r1,0(r2)` creates a read event R`r2`x from register `r2`, with label (b) in Fig. 19. This event reads the address x in `r2`; this leads to a read event $Rx1$ from x labelled (a) in Fig. 19. The event (a) was previously the only event we considered. Finally, the value read from x by the event (a) being 1, the `lwz`

Name	Code	Comment	Doc. [25] (p. in pdf)
load word and zero	`lwz rt,d(ra)`	let x be the address in `ra`; this instructions loads into `rt` from the address $x + d$	p. 76
load word and zero indexed	`lwzx rt,ra,rb`	let x_a be the address in `ra` and x_b the address in `rb`; this instruction loads into `rt` from the address $x_a + x_b$	p. 76
store word	`stw rs,d(ra)`	let x be the address in `ra`; this instruction stores from `rs` into the address $x + d$	p. 81
store word indexed	`stwx rs,ra,rb`	let x_a be the address in `ra` and x_b the address in `rb`; this instruction stores from `rs` into the address $x_a + x_b$	p. 81
branch if not equal	`bne L`	checks if register `cr0` (the same as the one that indicates whether a `stwcx.` has succeeded) holds 0, if not branches to `L`	p. 63
compare word immediate	`cmpwi rx, v`	compares the value in `rx` with `v` and writes 1 if equal (resp. 0 if not) in `cr`	p. 102
xor	`xor rd,ra,rb`	let v_a be the content of `ra` and v_b be the content of `rb`; this instruction writes v_a `xor` v_b into `rd`	p. 107
isync	`isync`	when placed after a `bne`, forms a read-write, read-read non-cumulative barrier	p. 661
lwsync	`lwsync`	read-write, read-read and write-write A- and B-cumulative barrier	p. 700
sync	`sync`	read-write, read-read, write-write and write-read A- and B-cumulative barrier	p. 700

Fig. 18 Table of the Power assembly instructions used in this paper

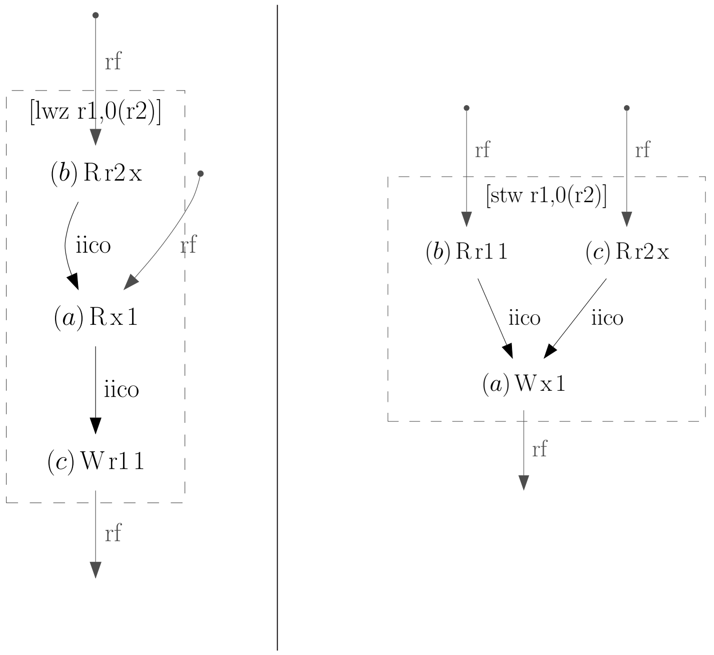


Fig. 19 Semantics of `lwz` and `stw`

`r1,0(r2)` creates a write event W`r1`1 to register `r1` with value 1, labelled (c) in Fig. 19.

Similarly, consider two registers `r1` and `r2`, such that `r1` initially holds the value 1, `r2` initially holds an address x, and x holds the value 0. In this case, an instruction `stw r1,0(r2)` creates a read event R`r2`x from register `r2`, with label (c) in Fig. 19. This event reads the address x in `r2`. In parallel, the store creates a read event R`r1`1 from `r1`, reading 1, labelled (c) in Fig. 19. Finally, the value read from `r1` by the event (b) being 1, the `stw r1,0(r2)` creates a write event $Wx1$ to x with value 1, labelled (a) in Fig. 19. The event (a) previously was the only event we considered.

Intra-Instruction Causality An execution witness now includes an additional *intra-instruction causality* relation $\overset{\text{iico}}{\to}$, as in [26,8].

For example, executing the load `lwz r1, 0(r2)`—which semantics is given in Fig. 19 (`r2` holding the address of a memory location x containing 1)—creates three events (a) R `r2` x, (b) R x 1 and (c) W `r1` 1, such that $(a) \overset{\text{iico}}{\to} (b) \overset{\text{iico}}{\to} (c)$. The $\overset{\text{iico}}{\to}$ relation between (a) and (b) indicates that the load instruction has to perform the read (a) from `r2` before the read (b) from x. Before reading x from `r2` via the event (a), the address x is undetermined. Similarly, (b) and (c) are related by $\overset{\text{iico}}{\to}$, since the write (c) determines the value it has to write into `r1` from the value read by (b).

Stores are less constrained, as depicted in Fig. 19. Indeed the read events (b) and (c) may be performed independently. However, the write event (a) determines its target x and its value 1 from the reads (c) and (b) respectively, hence $(b) \overset{\text{iico}}{\to} (a)$ and $(c) \overset{\text{iico}}{\to} (a)$.

Read-From Map Naturally, $\overset{\text{rf}}{\to}$ now also relates register events: we write $\overset{\text{rf-reg}}{\to}$ the subrelation of $\overset{\text{rf}}{\to}$ relating register stores to register loads that read their values.

6.1.2 Commit Events

We also add *commit events* in order to express branching decisions. We write $\mathbb{C}$ for the set of commits, and c for an element of $\mathbb{C}$.

Consider for example the test given at the left of Fig. 20, written in PowerPC assembly. Suppose that the register `r5` initially holds the address x, and the register `r6` the address y. The `lwz r1,0(r5)` at line (1) and the `stw r2,0(r6)` at line (4) are separated at lines (2)–(3) by a compare and branch sequence, written `cmpwi r1,0` then `bne L0`.

In the execution of this test, given at the right of Fig. 20, the `lwz r1,0(r5)` leads to the read event $Rx0$ from x labelled (a). The `stw r2,0(r6)` leads to the write event $Wy1$ to y labelled (b). At runtime, the compare instruction of line (2) yields equality and writes its result 2 (value 2 encodes equality) in the control register CR0, as depicted by the event (f). The conditional branch instruction reads CR0 (read event (g)) whose value determines the branching decision (commit event (h)): `bne` being "branch not-equal", the branch is not taken. Observe that the execution itself of the store at line (4) depends upon the branching decision. This is depicted in the execution we give here by the events yielded by this store being present and following the commit event (h) in program order. To summarise, we witness a

```
PPC ctrl
{
0:r5=x; 0:r6=y;0:r2=1;
x=0; y=0;
}
P0              ;
(1) lwz r1,0(r5) ;
(2) cmpwi r1,0   ;
(3) bne L0       ;
(4) stw r2,0(r6) ;
L0:              ;
```

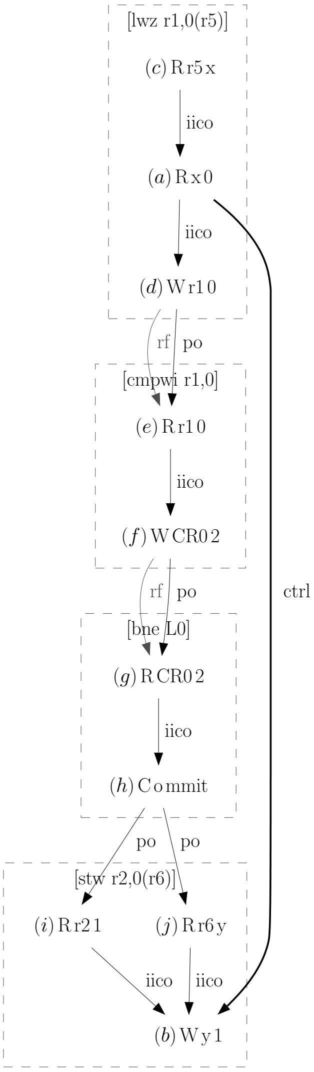


Fig. 20 A Test For Control Dependency in PowerPC Assembly

chain of dependencies through registers from the read (a) to the commit (h) that is $\overset{\text{po}}{\to}$-before the write (b). This situations defines a *control dependency* depicted by the $\overset{\text{ctrl}}{\to}$ arrow at the right of Fig. 20.

6.1.3 Barrier Events

We add *barrier events* in order to indicate the presence of a barrier in the code. We handle three barrier instructions : `isync`, `sync` and `lwsync`. The `sync` barrier is Power's heavyweight barrier, sometimes written `hwsync`. The `lwsync` barrier is

```
PPC isync
{
0:r5=x; 0:r6=y;0:r2=0;
x=0; y=1;
}
P0                ;
(1) lwz r1,0(r5) ;
(2) cmpwi r1,0   ;
(3) bne L0       ;
(4) isync        ;
(5) lwz r2,0(r6) ;
L0:          ;
```

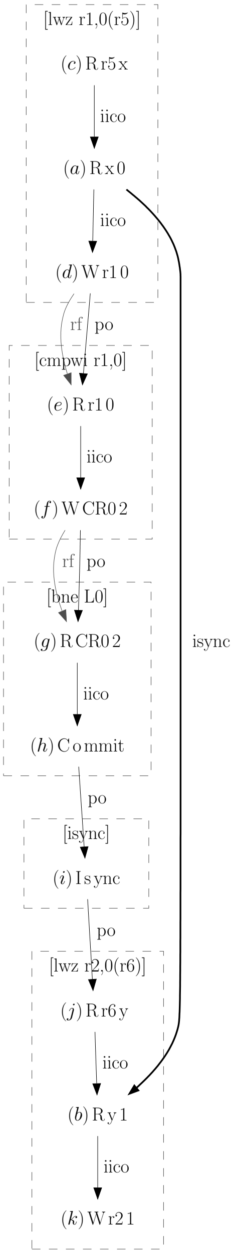


Fig. 21 An Example Use of the `isync` Barrier

a lightweight version of the sync barrier. The isync barrier is even lighter, and mainly used after branching instructions. We distinguish the corresponding events by the eponymous predicates, is-sync, is-lwsync, and is-isync.

Consider for example the test given at the left of Fig. 21, written in PowerPC assembly. Suppose that the register r5 initially holds the address x, and the register r6 the address y.

The lwz r1,0(r5) at line (1) and the lwz r2,0(r6) at line (5) are separated at line (2) by a compare instruction, written cmpwi r1,0, which influences the taking of the following branch at line (3), written bne L0. The branch is followed by an isync barrier at line (4).

In the execution of this test we give at the right of Fig. 21, the lwz r1,0(r5) at line (1) leads to the read event $Rx0$ from x labelled (a). The lwz r2,0(r6) at line (5) leads to the read event $Ry1$ from y labelled (b). These two instructions are separated by a branch followed by an isync. This is depicted in the execution we give here by the commit event labelled (h), in $\overset{\text{po}}{\to}$ with the isync event labelled (i). Hence the read (a) from x and the read (b) from y are globally ordered by the isync barrier, as depicted by the $\overset{\text{isync}}{\to}$ arrow between them. Notice that the $\overset{\text{isync}}{\to}$ notation can be slightly misleading, since an isync barrier alone between two reads does not suffice to enforce global ordering. Instead, the global ordering results from the combination of the the data dependency from the read (a) to the commit (h) and from the presence of the isync barrier between the commit (h) and the second read (b). In Fig. 21, we only depict the ordering between (a) and (b), for this corresponds to the semantics of the $\overset{\text{isync}}{\to}$ relation as defined in Fig. 22.

6.2 Description of the Model

We now describe the preserved program order, global reads-from maps, and barrier relation of the CAV 2010 Power model. The full definition of this model is collected in Fig. 22.

6.2.1 Preserved Program Order

We present in Fig. 22(a) the definition of $\overset{\text{ppo-ppc}}{\to}$, induced by lifting the ordering constraints of a processor to the global level (where $(\overset{\text{r}}{\to})^+$ stands for the transitive closure of a given relation $\overset{\text{r}}{\to}$). This is a formal presentation of the *data dependencies* ($\overset{\text{dd}}{\to}$) and *control dependencies* ($\overset{\text{ctrl}}{\to}$ and $\overset{\text{isync}}{\to}$) of [25, p. 661] which allow loads to be speculated if no isync is added after the branch but prevents stores from being speculated in any case.

Data Dependencies More precisely, we consider that there is a data dependency between two events, written $m_1 \overset{\text{dd}}{\to} m_2$, when:

- they are in $\overset{\text{rf-reg}}{\to}$, *i.e.* m_1 is a register write and m_2 a register read reading from m_1, or
- they are in $\overset{\text{iico}}{\to}$, *i.e.* they both come from the same instruction and the execution of m_2 depends on m_1, *e.g.* when using the value m_1 read, or
- there is a path of $\overset{\text{rf-reg}}{\to} \cup \overset{\text{iico}}{\to}$ between m_1 and m_2.

Control Dependencies We consider that there is a control dependency between two events, written $m_1 \overset{\text{ctrl}}{\to} m_2$, when:

- m_1 is a read,
- m_2 is a write, and
- there exists a commit event c between them in the program order, such that c has a data dependency on m_1.

If there is a control dependency between two events m_1 and m_2, it means that they form a read-write pair separated by a conditional jump, and that the condition of the jump is data-dependent on m_1.

Two events separated by a conditional jump may be reordered if:

- the first one is a write, or
- they are both reads, and there is no `isync` between the branch and the second read (*c.f.* Sec. 6.2.1, semantics of `isync`)

The test and the associated execution of Fig. 20 give an example of control dependency between the read (a) associated to the `lwz r1,0(r5)` at line (1) and the write (b) associated to the `stw r2,0(r6)` at line (3).

Semantics of the `isync` *Barrier* We now give the CAV 2010 model of the `isync` barrier. The ordering induced by `isync` is similar to a control dependency on read-read pairs. We consider that two events m_1 and m_2 are ordered by an `isync` barrier when:

- m_1 is a read, and
- there exists a commit event c in data dependency with m_1, separated from m_2 by an `isync` barrier in the program order.

In particular, this means that two events m_1 and m_2 separated by an `isync` can be reordered if:

- m_1 is a write, or
- there is no commit between m_1 and m_2.

The test and the execution of Fig. 21 give an example of `isync` ordering between the read (a) associated to `lwz r1,0(r5)` at line (1) and the read (b) associated to `lwz r2,0(r6)` at line (5).

All Together Finally, we consider that two events m_1 and m_2 are in Power's preserved program order, written $m_1 \overset{\text{ppo-ppc}}{\to} m_2$ when:

- there is a control dependency between them, *i.e.* $m_1 \overset{\text{ctrl}}{\to} m_2$, or
- they are ordered by an `isync` barrier, *i.e.* $m_1 \overset{\text{isync}}{\to} m_2$, or
- m_1 is a read, and there is a path of $\overset{\text{dd}}{\to} \cup (\overset{\text{po-loc}}{\to} \cap (\mathbb{W} \times \mathbb{R}))$ between m_1 and m_2.

Resemblance to Sparc Relaxed Memory Order's Preserved Program Order The preserved program order that we suggest for Power is similar to the one of Sparc Relaxed Memory Order (RMO) [30, V9, p. 293]. Writing $S(X)$ (resp. $L(X)$) when the memory transaction X is a store (resp. load), Sparc's documentation defines the *dependence order* as follows:

> *Dependence order is a partial order that captures the constraints that hold between instructions that access the same processor register or memory location. [...] Two memory transactions [i.e. accesses] X and Y are dependence ordered, denoted by $X <_d Y$, if and only if they are program ordered, [written] $X <_p Y$, **and** at least one of the following conditions is true:*
>
> *(1) The execution of Y is conditional on X, and $S(Y)$ is true. [i.e. they are in control dependency: in particular, Y is a write.]*
>
> *(2) Y reads a register that is written by X. [This corresponds to our $\overset{\textit{rf-reg}}{\rightarrow}$ relation.]*
>
> *(3) X and Y access the same memory location, and $S(X)$ and $L(Y)$ are both true. [This corresponds to the $\overset{\textit{po-loc}}{\rightarrow} \cap (\mathbb{W} \times \mathbb{R})$ part of Power's $\overset{\textit{ppo}}{\rightarrow}$ definition.]*
>
> *[...] It is important to remember that partial ordering is transitive.*

The items (1) and (2) correspond to our $\overset{\text{ctrl}}{\rightarrow}$ and $\overset{\text{dd}}{\rightarrow}$ relations, while the item (3) corresponds to the $\overset{\text{po-loc}}{\rightarrow} \cap (\mathbb{W} \times \mathbb{R})$ part of Power's $\overset{\text{ppo}}{\rightarrow}$ definition. Moreover, as mentioned in the excerpt above, that order should be transitive. Hence we take the transitive closure of the relation formed by the union of these two relations.

Finally, we take the intersection of $(\overset{\text{dd}}{\rightarrow} \cup (\overset{\text{po-loc}}{\rightarrow} \cap (\mathbb{W} \times \mathbb{R})))^{+}$ and $\mathbb{R} \times \mathbb{W}$, exactly as in RMO [30, V9, p.295]. Writing $M(X,Y)$ when X and Y are separated by a barrier, and X_a when X accesses memory location a, Sparc's documentation defines the legality of a RMO memory order as follows:

> *A memory order [written $<_m$] is legal [in RMO] if and only if:*
>
> *(1) $X <_d Y$ & $L(X) \Rightarrow X <_m Y$ [i.e. X and Y are in dependence order as defined above, and X is a read.]*
>
> *(2) $M(X,Y) \Rightarrow X <_m Y$ [i.e. X and Y are separated by a barrier; we treat this* via *$\overset{ab}{\rightarrow}$, not $\overset{ppo}{\rightarrow}$.]*
>
> *(3) $Xa <_p Ya$ & $S(Y) \Rightarrow X <_m Y$ [Y is a write, and they are both relative to the same memory location a. This corresponds to the fact that we consider $\overset{wsi}{\rightarrow}$ and $\overset{fri}{\rightarrow}$ to be global.]*

The first item indicates that RMO also only considers the dependency chains starting from a load, which explains why we take the intersection with $\mathbb{R} \times \mathbb{M}$.

Discussion of Power's $\overset{ppo}{\rightarrow}$ We include in $\overset{\text{ppo}}{\rightarrow}$ any chain of data dependencies and $\overset{\text{rfi}}{\rightarrow}$ starting from a read, by the $((\overset{\text{dd}}{\rightarrow} \cup (\overset{\text{po-loc}}{\rightarrow} \cap (\mathbb{W} \times \mathbb{R})))^{+} \cap (\mathbb{R} \times \mathbb{M})$ part. However, we do not authorise control dependencies in such a chain. Indeed, consider the example given in Fig. 23, which we observed to be exhibited on a Power 5 for example.

On P_0, the `lwz r1,0(r7)` at line (1) is in control dependency with the `stw r3,0(r9)` at line (5), because of the compare and branch sequence between them (lines (2) to (4)). The `lwz r2,0(r9)` at line (6) is in data dependency with the `lwzx r4,r10,r8` at line (8), because of the `xor r10,r2,r2` between them, at line (7).

Since the `lwz r1,0(r7)` at line (1) on P_0 reads 1 (see the final state, `0:r1=1`), there is a $\overset{\text{rfe}}{\rightarrow}$ between the `stw r2,0(r7)` at line (3) on P_1 and the `lwz r1,0(r7)` at line (1) on P_0. Because of the A-cumulativity of the `sync` barrier on P_1, the `stw r1,0(r8)` at line (1) on P_1 is in $\overset{\text{ab}}{\rightarrow}$ with the `lwz r1,0(r7)` at line (1) on P_0.

$$\xrightarrow{\text{dd}} \triangleq (\xrightarrow{\text{rf-reg}} \cup \xrightarrow{\text{iico}})^+ \qquad\qquad r \xrightarrow{\text{ctrl}} w \triangleq \exists c \in \mathbb{C}.\ r \xrightarrow{\text{dd}} c \xrightarrow{\text{po}} w$$

$$r \xrightarrow{\text{isync}} e \triangleq \exists c \in \mathbb{C}.\ r \xrightarrow{\text{dd}} c \wedge \exists b.\ \text{is-isync}(b) \wedge c \xrightarrow{\text{po}} b \xrightarrow{\text{po}} e$$

$$\xrightarrow{\text{dp}} \triangleq \xrightarrow{\text{ctrl}} \cup \xrightarrow{\text{isync}} \cup \Big(\big(\xrightarrow{\text{dd}} \cup (\xrightarrow{\text{po-loc}} \cap (\mathbb{W} \times \mathbb{R}))\big)^+ \cap (\mathbb{R} \times \mathbb{M})\Big) \qquad\qquad \xrightarrow{\text{ppo-ppc}} \triangleq \xrightarrow{\text{dp}}$$

(a) Preserved program order

$$\begin{aligned} m_1 \xrightarrow{\text{sync}} m_2 &\triangleq \exists b.\ \text{is-sync}(b) \wedge m_1 \xrightarrow{\text{po}} b \xrightarrow{\text{po}} m_2 \\ m_1 \xrightarrow{\text{ab-sync}} m_2 &\triangleq m_1 \xrightarrow{\text{sync}} m_2 \\ &\quad \vee \exists r.\ m_1 \xrightarrow{\text{rf}} r \xrightarrow{\text{ab-sync}} m_2 \\ &\quad \vee \exists w.\ m_1 \xrightarrow{\text{ab-sync}} w \xrightarrow{\text{rf}} m_2 \end{aligned}$$

(b) Barrier sync

$$\begin{aligned} m_1 \xrightarrow{\text{lwsync}} m_2 &\triangleq \exists b.\, \text{is-lwsync}(b) \wedge m_1 \xrightarrow{\text{po}} b \xrightarrow{\text{po}} m_2 \\ m_1 \xrightarrow{\text{ab-lwsync}} m_2 &\triangleq m_1 \xrightarrow{\text{lwsync}} m_2 \cap ((\mathbb{W} \times \mathbb{W}) \cup (\mathbb{R} \times \mathbb{M})) \\ &\quad \vee \exists r.\, m_1 \xrightarrow{\text{rf}} r \xrightarrow{\text{ab-lwsync}} m_2 \wedge m_2 \in \mathbb{W} \\ &\quad \vee \exists w.\, m_1 \xrightarrow{\text{ab-lwsync}} w \xrightarrow{\text{rf}} m_2 \wedge m_1 \in \mathbb{R} \end{aligned}$$

(c) Barrier lwsync

$$\begin{aligned} \xrightarrow{\text{ab-ppc}} &\triangleq \xrightarrow{\text{ab-sync}} \cup \xrightarrow{\text{ab-lwsync}} \\ \mathit{Power} &\triangleq (\xrightarrow{\text{ppo-ppc}}, \emptyset, \xrightarrow{\text{ab-ppc}}) \end{aligned}$$

Fig. 22 The CAV 2010 Power Model

```
{
0:r7=y; 0:r8=z; 0:r9=x; 0:r3=1;
1:r7=y; 1:r8=z; 1:r1=1; 1:r2=1;
}
P0                 | P1           ;
(1) lwz r1,0(r7)   | stw r1,0(r8) ;
(2) cmpwi r1,0     | sync         ;
(3) beq L0         | stw r2,0(r7) ;
(4) L0:            |              ;
(5) stw r3,0(r9)   |              ;
(6) lwz r2,0(r9)   |              ;
(7) xor r10,r2,r2  |              ;
(8) lwzx r4,r10,r8 |              ;
exists (0:r1=1  /\ 0:r4=0)
```

Fig. 23 Contradiction with the View Order Formulation

```
PPC rfi000
"DpdR Fre Rfi DpdR Fre Rfi"
Cycle=DpdR Fre Rfi DpdR Fre Rfi
Relax=Rfi
Safe=Fre DpdR
{
0:r2=x; 0:r6=y;
1:r2=y; 1:r6=x;
}
 P0            | P1            ;
 li r1,1       | li r1,1       ;
 stw r1,0(r2)  | stw r1,0(r2)  ;
 lwz r3,0(r2)  | lwz r3,0(r2)  ;
 xor r4,r3,r3  | xor r4,r3,r3  ;
 lwzx r5,r4,r6 | lwzx r5,r4,r6 ;
exists
(0:r3=1 /\ 0:r5=0 /\ 1:r3=1 /\ 1:r5=0)
```

Fig. 24 A diy PowerPC Test for the Rfi Relaxation

Since the `lwzx r4,r10,r8` at line (8) on P_0 reads 0 (see the final state, `0:r4=0`), there is a $\xrightarrow{\text{fr}}$ between this load and the `stw r1,0(r8)` at line (1) on P_1.

Hence we have a cycle of relations as follows, starting from `lwz r1,0(r7)` at line (1) on P_0 (writing $(i:j)$ for the instruction at line (j) on P_i):

$$(0:1) \xrightarrow{\text{CtrldW}} (0:5) \xrightarrow{\text{PosWR}} (0:6) \xrightarrow{\text{DpdR}} (0:8) \xrightarrow{\text{Fre}} (1:1) \xrightarrow{\text{ACSyncdWW}} (0:1)$$

Since the specified outcome is exhibited, we know that there is a subsequence of this cycle which is not global. The only possible relaxation here is the PosWR one between the `stw r3,0(r9)` at line (5) and the `lwz r2,0(r9)` at line (6). If we added $\xrightarrow{\text{ctrl}}$ to the transitive part of Power's $\xrightarrow{\text{ppo}}$, the outcome would be forbidden in our model. We deduce from this example that $\xrightarrow{\text{ctrl}}$ cannot be included in such chains.

6.2.2 Read-From Maps

Internal Read-From Maps The internal read-from maps are not global, since Power allows store buffering. Running the test given in Fig. 24 confirms this hypothesis. Indeed this test, generated by diy, proceeds from a cycle $\xrightarrow{\text{DpdR}};\xrightarrow{\text{Fre}};\xrightarrow{\text{Rfi}};\xrightarrow{\text{DpdR}};\xrightarrow{\text{Fre}};\xrightarrow{\text{Rfi}}$. We know that $\xrightarrow{\text{dp}}$ is global, since included in $\xrightarrow{\text{ppo}}$, and $\xrightarrow{\text{fr}}$ is global as well. Thus in this test, the only possible relaxation is $\xrightarrow{\text{rfi}}$. Since the outcome is exhibited, as shown in Fig. 15, we know that $\xrightarrow{\text{rfi}}$ is actually relaxed on Power.

Store buffering is a fairly common relaxation. Indeed, the models TSO, PSO, RMO and Alpha also relax their internal read-from maps, *i.e.* allow store buffering.

External Read-From Maps The external read-from maps are not global either, as revealed by running **iriw** with data dependencies (Fig. 11) on a Power machine. This test is associated with the cycle $\xrightarrow{\text{Fre}};\xrightarrow{\text{Rfe}};\xrightarrow{\text{DpdR}};\xrightarrow{\text{Fre}};\xrightarrow{\text{Rfe}};\xrightarrow{\text{DpdR}}$. We know that $\xrightarrow{\text{fre}}$ is always considered global in our framework, an hypothesis which has not been invalidated by the experiment we present in Sec. 5. We also know that $\xrightarrow{\text{dp}}$ is global in Power, since $\xrightarrow{\text{ppo}}$ is equal to $\xrightarrow{\text{dp}}$. Therefore, the only possible relaxation in the test of Fig. 11 is $\xrightarrow{\text{rfe}}$. Since the specified outcome is exhibited as shown in Fig. 15, $\xrightarrow{\text{rfe}}$ is relaxed.

This is probably the main particularity of the Power architecture: the models SC, TSO, PSO, RMO and Alpha do not relax the atomicity of stores. Another model relaxing store atomicity may be Itanium [20]. However, we do not know whether Itanium could be described by the generic framework we presented in Sec. 2.

6.2.3 Barriers

The `sync` *Barrier* The `sync` barrier is defined in Fig. 22 (b) as a full A- and B-cumulative barrier. We saw in Cor. 1 that such a barrier restores *SC* from a weaker model.

The `lwsync` *Barrier* The `lwsync` barrier is defined in Fig. 22 (b). `lwsync` acts as `sync` except on store-load pairs, in both the base and cumulativity cases.

7 Discussion of the Model

The Power model presented here was based largely on black-box testing. It is, to the best of our knowledge and following reasonably thorough experiment, sound with respect to the Power implementations we have tested. We consider it a success: it is the first non-trivial attempt towards the formalisation of the Power architecture with cumulative barriers; is is also notable in being an axiomatic model of a non-multi-copy-atomic architecture in a global-time style. Moreover, our test generation demonstrably generates interesting and useful tests that can be used in the context of other models, and can be reused for other architectures (we have also applied it to x86).

However, as mentioned, we do not claim this model to be definitive. In particular, it gives weaker semantics for the Power `lwsync` barrier than it should, and (we have recently learnt) it gives stronger semantics than the architectural intent in some cases.

For the `lwsync` issue, consider the common programming idiom given at the left of Fig. 25 in C-like syntax: P_0 first stores the value 1 into the location x, and then communicates a pointer to x (by storing it into another location p); P_1 reads that pointer and dereference it. In the *SC* model, if P_1 reads x from p, one can be sure that dereferencing x will always yield 1 (*i.e.* will never yield 0, the initial contents of x). The Power documentation suggests that inserting a `lwsync` barrier between the two stores of P_0 suffices to forbid the non-*SC* behaviour, due to the effective address of P_1 second load depending on the value loaded by P_1 first load. Our model is unable to explain this, because our semantics for the `lwsync` barrier is rather weak.

To see this, consider the PowerPC example we give at the right of Fig. 25, which corresponds to the idiom, where we have replaced the "true" address dependency by a "false" one, whose effect is identical, according to the documentation. That example could, according to our Power model, exhibit the execution given in Fig. 26 since there is no cycle in $\xrightarrow{\text{ghb}}$. More specifically, the B-cumulativity of the `lwsync`

P_0	P_1
(a) `x` $\leftarrow$ `1`	(c) `r1` $\leftarrow$ `p`
(b) `p` $\leftarrow$ `& x`	(d) `r2` $\leftarrow$ `*r1`

Observed? `r1=x; r2=0;`

```
{0:r5=x; 0:r6=y;
 1:r5=x; 1:r6=y;
 x=0; y=0;}

    P0           | P1                ;
    li r3,1      | (c) lwz r1,0(r6)   ;
(a) stw r3,0(r5) |     xor r9,r1,r1   ;
    lwsync       | (d) lwzx r2,r9,r5  ;
    li r4,1      |                    ;
(b) stw r4,0(r6) |                    ;

exists? (1:r1=1 /\ 1:r2=0)
```

Fig. 25 A Common Programming Idiom and a Corresponding Test in PowerPC Assembly

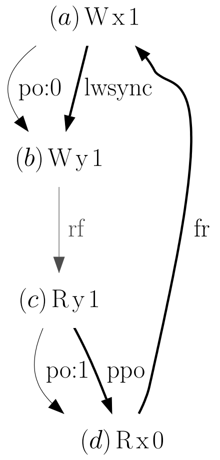


Fig. 26 Weakness of Our `lwsync` Semantics

barrier does not apply in this case, which means that the sequence $(a) \xrightarrow{\text{lwsync}} (b) \xrightarrow{\text{rf}} (c)$ is not global. Indeed, we observed BCLwSyncWW to be relaxed, as shown by the last line of Fig. 15. Hence this behaviour is not forbidden by our model, though it is expected to be, as confirmed by intensive experiments on the test of Fig. 25.

The per-processor view of the memory promoted by the Power documentation helps to explain why the execution of Fig. 26 is guaranteed not to happen. Indeed, the definition of `lwsync` of the documentation specifies (since the pair $((a),(b))$ is a write-write pair, hence an "applicable pair" for `lwsync`) that "the memory barrier ensures that $[(a)]$ will be performed with respect to any processor $[\ldots]$ before $[(b)]$ is performed with respect to that processor $[\ldots]$." Hence the pair $((a),(b))$ has to be seen in the same order by P_0 and P_1, because there is a `lwsync` between (a) and (b). Therefore, if the read (c) on P_1 reads 1 from the write (b) on P_0, then P_1 has also seen the write (a) to x on P_0, hence we cannot have $(d) \xrightarrow{\text{fr}} (a)$ as in Fig. 26.

For the other issue, the model forbids the final result of a test R01 [27], which is apparently intended to be architecturally allowed. We have not observed that result in experimental testing, so this does not contradict the soundness of the model with respect to the implementations we have tested, but in principle might be an issue with respect to future Power implementations, which might permit that final result, so compilers and software verification work should not assume that it is forbidden.

More recently, we have built another Power model [27], based not just on black-box testing but also on extensive discussion with IBM staff about the relevant aspects of the microarchitecture and of the architectural intent. That model resolves both of these issues: as far as we know, it captures the architectural intent precisely, and (as far as we know) is also experimentally sound with respect to Power 6 and Power 7 implementations. However, to do so we had to abandon the attractive simplicity of the global-time axiomatic model presented here — the new model is in an abstract-machine style, with a microarchitectural flavour (e.g., with explicit speculation in the core), though abstracting from implementation detail as much as we can. That is both good and bad: the abstract-machine model may be more intuitive for practicing engineers, but is mathematically more complex to work with. Ideally, we would also have an equivalent axiomatic model.

8 Related Work

Formal memory models roughly fall into two classes: operational models and axiomatic models. Operational models, e.g. [31,19], are abstractions of actual machines composed of idealised hardware components such as queues. They can be appealingly intuitive and offer a relatively direct path to simulation, at least in principle. Axiomatic models focus on segregating allowed and forbidden behaviours, usually by constraining various order relations on memory accesses; they are particularly well adapted for model exploration, as we do here. Several of the more formal vendor specifications have been in this style [10,30,20].

One generic axiomatic model related to ours is Nemos [32]. This covers a broad range of models including Itanium as the most substantial example. Itanium is rather different to Power; we do not know whether our framework could handle such a model or whether a satisfactory Power model could be expressed in Nemos. By contrast, our framework owes much to the concept of *relaxation*, informally presented in [5]. As regards tools, Nemos calculates the behaviour of example programs w.r.t. to a model, but offers no support for generating or running tests on actual hardware.

Previous work on model-building based on experimental testing includes that of Collier [17], Adir et al. [4,3], and Sarkar et al. [26,24,28]. The former is based on hand-coded test programs and Collier's model, in which the cumulativity of the Power barriers does not seem to fit naturally. Adir et al. developed an axiomatic model for a version of Power before cumulative barriers [3]; their testing [4] aims to produce interesting collisions (accesses to related locations) with knowledge of the microarchitecture, using an architecture model as an oracle to determine the legal results of tests rather than (as we do) generating interesting tests from the memory model. Sarkar et al. developed models for the x86 based in part on litmus testing with hand-written tests, using a version of the same tool as we use here for running them on hardware.

9 Conclusion

We present here a general class of axiomatic memory models, extending smoothly from *SC* to very relaxed models. It even extends to a model for Power processors, which does not have store atomicity. This is despite the fact that our models are simple global-time models, without complex structures such as multiple write events per store [20], or a view order per processor [17,3,25,11]. Our principal validity condition is simple, just an acyclicity check of the global happens before relation. This check is already known for *SC* [22], and recent verification tools use it for architectures with store buffer relaxation [18,15].

Our model lends itself well to exploration by automatic and systematic test generation. This is a significant advance over reliance on hand-crafted litmus tests, the current state of the art. Our `diy` tool has discovered several interesting corner cases which would have been easy to miss in a less systematic exploration. We also believe that extensive tests of actual hardware is a crucial component of building axiomatic models, and we provide a tool suite for both generation and running of tests. Such testing can also discover problems of the implementation, and our testing revealed a rare Power 5 implementation erratum for barriers.

Our Power model captures key aspects of the behaviour of cumulative barriers. It can be used as a basis for reasoning, particularly about the placement of heavyweight barriers `hwsync`. However, we do not regard it as definitive: there are known tests for which the model is too weak w.r.t. our perception of the architectural intent (particularly involving the lightweight barrier `lwsync`) and also cases where it is too strong w.r.t. the architectural intent (though sound w.r.t. current implementations). A desirable direction for future work would be to develop an axiomatic model that is equivalent to our more recent abstract-machine model [27].

Acknowledgements We thank Damien Doligez and Xavier Leroy for invaluable discussions and comments, Assia Mahboubi and Vincent Siles for advice on the Coq development, Thomas Braibant, Jules Villard and Boris Yakobowski for comments on a draft, and the anonymous referees for comments on the presentation. We thank the HPCx (UK) and IDRIS(.fr) high-performance computing services. We acknowledge support from EPSRC grants EP/F036345, EP/H005633, and EP/H027351/1, and ANR grant ANR-06-SETI-010-02.

References

1. *AMD64 Architecture Programmer's Manual.* Advanced Micro Devices, September 2007. (3 vols).
2. *Intel 64 and IA-32 Architectures Software Developer's Manual (5 vols).* Intel Corporation, March 2010. rev. 34.
3. A. Adir, H. Attiya, and G. Shurek. Information-Flow Models for Shared Memory with an Application to the PowerPC Architecture. In *TPDS*, 2003.
4. A. Adir and G. Shurek. Generating Concurrent Test-Programs with Collisions for Multi-Processor Verification. In *HLDVT*, 2002.
5. S. V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, 29:66–76, 1995.
6. M. Ahamad, R. A. Bazzi, R.John, P. Kohli, and G. Neiger. The Power of Processor Consistency. In *SPAA*, 1993.
7. J. Alglave. *A Shared Memory Poetics.* PhD thesis, Université Paris 7 and INRIA, 26 November 2010. `http://diy.inria.fr/alglave-thesis.pdf`.

8. J. Alglave, A. Fox, S. Ishtiaq, M. O. Myreen, S. Sarkar, P. Sewell, and F. Zappa Nardelli. The semantics of Power and ARM multiprocessor machine code. In *DAMP*, 2009.
9. J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Fences in Weak Memory Models. In *CAV*, 2010.
10. Alpha Architecture Reference Manual, Fourth Edition, 2002.
11. ARM Architecture Reference Manual (ARMv7-A and ARMv7-R), April 2008.
12. Arvind and J.-W. Maessen. Memory Model = Instruction Reordering + Store Atomicity. In *ISCA*. IEEE Computer Society, 2006.
13. Y. Bertot and P. Casteran. *Coq'Art*. Springer Verlag, EATCS Texts in Theoretical Computer Science, 2004.
14. H.-J. Boehm and S.V. Adve. Foundations of the C++ Concurrency Memory Model. In *PLDI*, 2008.
15. S. Burckhardt and M. Musuvathi. Effective Program Verification for Relaxed Memory Models. In *CAV*, 2008.
16. J. Cantin, M. Lipasti, and J. Smith. The Complexity of Verifying Memory Coherence. In *SPAA*, 2003.
17. W. W. Collier. *Reasoning About Parallel Architectures*. Prentice-Hall, 1992.
18. S. Hangal, D. Vahia, C. Manovit, J.-Y. J. Lu, and S. Narayanan. TSOTool: A Program for Verifying Memory Systems Using the Memory Consistency Model. In *ISCA*, 2004.
19. L. Higham, J. Kawash, and N. Verwaal. Weak memory consistency models part I: Definitions and comparisons. *Technical Report98/612/03, Department of Computer Science, The University of Calgary, January*, 1998.
20. A Formal Specification of Intel Itanium Processor Family Memory Ordering, October 2002. Intel Document 251429-001.
21. L. Lamport. How to Make a Correct Multiprocess Program Execute Correctly on a Multiprocessor. *IEEE Trans. Comput.*, 46(7):779–782, 1979.
22. A. Landin, E. Hagersten, and S. Haridi. Race-free interconnection networks and multiprocessor consistency. *SIGARCH Comput. Archit. News*, 19(3):106–115, 1991.
23. J. Manson, W. Pugh, and S. V. Adve. The Java Memory Model. In *POPL*, 2005.
24. S. Owens, S. Sarkar, and P. Sewell. A Better x86 Memory Model: x86-TSO. In *TPHOL*, 2009.
25. Power ISA version 2.06, January 2009.
26. S. Sarkar, P. Sewell, F. Zappa Nardelli, S. Owens, T. Ridge, T. Braibant, M. Myreen, and J. Alglave. The Semantics of x86-CC Multiprocessor Machine Code. In *POPL*, 2009.
27. Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding Power Multiprocessors. In *PLDI*, 2011.
28. Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 53(7):89–97, July 2010. (Research Highlights).
29. D. Shasha and M. Snir. Efficient and Correct Execution of Parallel Programs that Share Memory. *ACM Trans. Program. Lang. Syst.*, 10(2):282–312, 1988.
30. Sparc Architecture Manual Versions 8 and 9, 1992 and 1994.
31. Y. Yang, G. Gopalakrishnan, and G. Lindstrom. UMM: an Operational Memory Model Specification Framework with Integrated Model Checking Capability. In *CCPE*, 2007.
32. Y. Yang, G. Gopalakrishnan, G. Linstrom, and K. Slind. Nemos: A Framework for Axiomatic and Executable Specifications of Memory Consistency Models. *IPDPS*, 2004.