Fences in Weak Memory Models

Jade Alglave[1], Luc Maranget[1], Susmit Sarkar[2], and Peter Sewell[2]

[1] INRIA [2] University of Cambridge

Abstract. We present a class of relaxed memory models, defined in Coq, parameterised by the chosen permitted local reorderings of reads and writes, and 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 to determine properties of processor implementations. We detail the results of our experiments on Power and the model we base on them. This work identified a rare implementation error in Power 5 memory barriers (for which IBM is providing a workaround); our results also suggest that Power 6 does not suffer from this problem.

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 (*SC*) memory [17]. For some, such as x86 [22, 20] and Power [21], 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 new contributions:

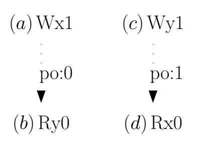
1. We introduce a class of memory models, defined in Coq [8], which we show how to instantiate to produce *SC*, *TSO* [24], and a Power model (3 below).
2. We describe our diy testing tool. Much discussion of memory models has been in terms of *litmus tests* (*e.g.* **iriw** [9]): ad-hoc multiprocessor programs for which particular final states may be allowed on a given architecture. Given a violation of *SC*, diy *systematically* and *automatically* generates litmus tests (including classical ones such as **iriw**) and runs them on the hardware.
3. We model important aspects of Power processors' behaviour, *i.e. ordering relaxations*, the lack of *store atomicity* [3, 7], and *A-cumulative barriers* [21].
4. We use diy to generate about 800 tests, running them up to $1e12$ times on 3 Power machines. Our experimental results confirm that our model captures many important aspects of the processor's behaviour, despite being

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

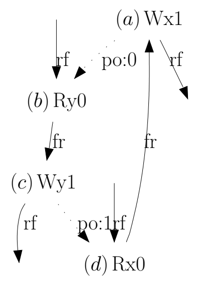
P_0	P_1
(a) `x` $\leftarrow$ `1`	(c) `y` $\leftarrow$ `1`
(b) `r1` $\leftarrow$ `y`	(d) `r2` $\leftarrow$ `x`

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

(a) Program



(b) Events and Program Order



(c) An execution witness

Fig. 1. A program and a candidate execution

in a simple global-time style rather than the per-processor timelines of the architecture text. They also identified a rarely occurring implementation error in Power 5 memory barriers (for which IBM is providing a workaround). They further suggest that Power 6 does not suffer from this.

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].

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 [20].

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) Wx1 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

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

Name	Notation	Comment	Sec.
program order	$m_1 \xrightarrow{\text{po}} m_2$	per-processor total order	2
dependencies	$m_1 \xrightarrow{\text{dp}} m_2$	dependencies	2
po-loc	$m_1 \xrightarrow{\text{po-loc}} m_2$	program order restricted to the same location	2.3
preserved program order	$m_1 \xrightarrow{\text{ppo}} m_2$	pairs maintained in program order	2.2
read-from map	$w \xrightarrow{\text{rf}} r$	links a write to a read reading its value	2.1
external read-from map	$w \xrightarrow{\text{rfe}} r$	$\xrightarrow{\text{rf}}$ between events from distinct processors	2.2
internal read-from map	$w \xrightarrow{\text{rfi}} r$	$\xrightarrow{\text{rf}}$ between events from the same processor	2.2
global read-from map	$w \xrightarrow{\text{grf}} r$	$\xrightarrow{\text{rf}}$ considered global	2.2
write serialisation	$w_1 \xrightarrow{\text{ws}} w_2$	total order on writes to the same location	2.1
from-read map	$r \xrightarrow{\text{fr}} w$	r reads from a write preceding w in $\xrightarrow{\text{ws}}$	2.1
barriers	$m_1 \xrightarrow{\text{ab}} m_2$	ordering induced by barriers	2.2
global happens-before	$m_1 \xrightarrow{\text{ghb}} m_2$	union of global relations	2.2
	$m_1 \xrightarrow{\text{hb-seq}} m_2$	shorthand for m_1 $(\xrightarrow{\text{rf}} \cup \xrightarrow{\text{ws}} \cup \xrightarrow{\text{fr}})$ m_2	2.3

Fig. 2. Table of relations

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.

The models are defined in terms of binary relations over these events, and Fig. 2 has a table of the relations we use.

As usual, the *program order* $\xrightarrow{\text{po}}$ 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, $e_1 \xrightarrow{\text{po}} e_2$ means that a sequential processor would execute i_1 before i_2. When instructions may perform several memory accesses, we take intra-instruction dependencies [22] into account to build a total order.

We postulate a $\xrightarrow{\text{dp}}$ relation to model the dependencies between instructions, such as *data* or *control dependencies* [21, pp. 653-668]. This relation is a subrelation of $\xrightarrow{\text{po}}$, and always has a read as its source.

2.1 Execution witnesses

Although $\xrightarrow{\text{po}}$ conveys important features of program execution, *e.g.* branch resolution, it does not characterise an execution. To do so, we postulate two relations $\xrightarrow{\text{rf}}$ and $\xrightarrow{\text{ws}}$ over memory events.

Reads-from map We write $w \xrightarrow{\text{rf}} r$ 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 $w \xrightarrow{\text{rf}} r$ (w can be an *init* store when r loads from the initial state). Thus, $\xrightarrow{\text{rf}}$ must be well formed following the $\text{wf} - \text{rf}$ predicate:

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

Write serialisation We assume all values written to a given location ℓ to be serialised, following a *coherence order*. This property is widely assumed by modern architectures. We define $\xrightarrow{\text{ws}}$ as the union of the coherence orders for all memory locations, which must be well formed following the $\text{wf} - \text{ws}$ predicate:

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

From-read map We define the following derived relation $\xrightarrow{\text{fr}}$[4] which gathers all pairs of reads r and writes w such that r reads from a write that is before w in $\xrightarrow{\text{ws}}$:

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

We define an *execution witness* X as follows (the well-formedness predicate wf on execution witnesses is the conjunction of those for $\xrightarrow{\text{ws}}$ and $\xrightarrow{\text{rf}}$):

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

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

2.2 Global Happens-Before

An 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. 2.3). This order corresponds roughly to the vendor documentation concept of memory events being *globally performed* [21, 13]: a write in $\xrightarrow{\text{ghb}}$ represents the point in global time when this write becomes visible to all processors; whereas a read in $\xrightarrow{\text{ghb}}$ represents the point in global time when the read takes place.

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 globally performed at once. Thus, $\xrightarrow{\text{rf}}$ is 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* [3]): 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* [3]. 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} for when $\xrightarrow{\text{ab}}$ is empty); *ppo* (resp. *grf*, *ab*, A.ghb) is the function returning the $\xrightarrow{\text{ppo}}$ (resp. $\xrightarrow{\text{grf}}$, $\xrightarrow{\text{ab}}$ and $\xrightarrow{\text{ghb}}$) relation when given an execution witness:

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

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

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

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

We now add two sanity conditions to the above. First, we require each processor to respect memory coherence for each location [11]. 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 $\xrightarrow{\text{hb-seq}}$ for $\xrightarrow{\text{rf}} \cup \xrightarrow{\text{ws}} \cup \xrightarrow{\text{fr}}$):

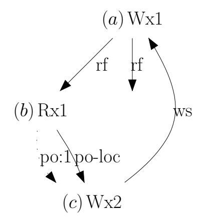
$$m_1 \xrightarrow{\text{po-loc}} m_2 \triangleq m_1 \xrightarrow{\text{po}} m_2 \wedge \text{loc}(m_1) = \text{loc}(m_2)$$
$$\text{uniproc}(X) \triangleq \text{acyclic}(\xrightarrow{\text{hb-seq}} \cup \xrightarrow{\text{po-loc}})$$

P_0	P_1
(a) `x` $\leftarrow$ `1`	(b) `r1` $\leftarrow$ `x`
	(c) `x` $\leftarrow$ `2`

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

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

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



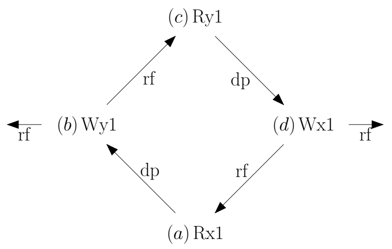


Fig. 3. Invalid executions according to the uniproc and thin criteria

For example, in Fig. 3 (a), we have $(c) \xrightarrow{\text{ws}} (a)$ (by `x` final value) and $(a) \xrightarrow{\text{rf}} (c)$ (by `r1` final value). 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 $\xrightarrow{\text{ws}}$.

Second, we rule out programs where values come *out of thin air* [19] (as in Fig. 3 (b)):

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

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

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

2.4 Comparing architectures *via* validity predicates

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

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

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:

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

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:

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

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.

2.5 Examples

We propose here alternative formulations of *SC* [17] and Sparc's *TSO* [24] in our framework, which we proved equivalent to the original definitions. We omit proofs and the formal details for lack of space, but they can be found online[2]. We write po(X) (resp. rf(X), 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) and makes writes available to all processors as soon as they are issued ($\xrightarrow{\text{rf}}$ are global). Thus, there is no need for barriers, and any architecture is weaker than *SC*: $SC \triangleq (MM, \text{rf}, \lambda X.\emptyset)$. The following criterion characterises, as in Sec. 2.4, valid *SC* executions on any architecture: $A.\text{check}_{SC}(X) = \text{acyclic}(\xrightarrow{\text{hb-seq}} \cup \xrightarrow{\text{po}})$. 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 [3]: *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. 2.4 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

We define 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 wait until w is globally performed, then read again to ensure r is globally performed after w. We provide a less costly requirement: when $w \xrightarrow{\text{rf}} r \xrightarrow{\text{po}} m$, where r may not be globally performed after w is, inserting a barrier instruction between the instructions generating r and m only forces the processor generating r and m to delay m until w is globally performed.

Formally, given $A_1 \leq A_2$, we define the predicate fb (*fully barriered*) on executions X by

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

where $\xrightarrow{\text{r}_2\setminus 1} \triangleq \xrightarrow{\text{r}_2} \setminus \xrightarrow{\text{r}_1}$ is the set difference, and $x \xrightarrow{\text{r}_1}; \xrightarrow{\text{r}_2} y \triangleq \exists z.\, x \xrightarrow{\text{r}_1} z \wedge z \xrightarrow{\text{r}_2} y$.

The fb predicate provides an insight on 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.

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

Theorem 1 (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 [21]: 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. 1 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.

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:

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

Thm. 1 shows that inserting such a barrier between all $\xrightarrow{\text{po}}$ pairs restores SC:

Corollary 1 (Barriers restoring SC).

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

Consider *e.g.* the **iriw** test depicted in Fig. 4. The specified outcome may be the result of a non-SC execution on a weak architecture in the absence of barriers. Our A-cumulative barrier forbids this outcome, as shown in Fig. 4: if placed between each pair of reads on P_0 and P_1, not only does it prevent their reordering, but 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).

iriw

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

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

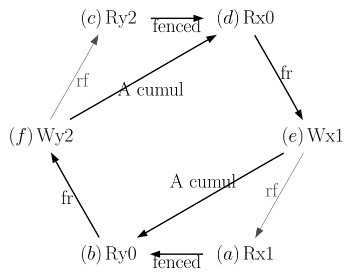


Fig. 4. Study of **iriw** with A-cumulative barriers

Thus, we force a program to have an SC behaviour by fencing all pairs in $\xrightarrow{\text{po}}$. Yet, it would be enough to invalidate non-SC executions, by fencing only the $\xrightarrow{\text{po}}$ pairs in the $\xrightarrow{\text{hb-seq}} \cup \xrightarrow{\text{po}}$ cycles of these executions. We believe the static analysis of [23] (based on compile-time approximation of $\xrightarrow{\text{hb-seq}} \cup \xrightarrow{\text{po}}$ cycles) applies to architectures relaxing store atomicity, if their barriers offer A-cumulativity.

4 diy: a testing tool

We present our diy (*do it yourself*) tool, which computes litmus tests in x86 or Power assembly code by generating violations of SC, *i.e.* cycles in $\xrightarrow{\text{hb-seq}} \cup \xrightarrow{\text{po}}$. A *diy* tutorial is available[1].

4.1 Cycles as specifications of litmus tests

Consider *e.g.* the outcome of Fig. 4 (a): it leads to the $\xrightarrow{\text{hb-seq}} \cup \xrightarrow{\text{po}}$ cycle of Fig. 4 (b): from $r_1 = 1$ on P_0, we know the load (a) reads from the store (e) on P_2, thus $(e) \xrightarrow{\text{rfe}} (a)$. By the `fence` on P_0, we know $(a) \xrightarrow{\text{fenced}} (b)$ and since $r_2 = 0$ on P_0, we know the load (b) read from the initial state, thus $(b) \xrightarrow{\text{fre}} (f)$; *idem* on P_1.

The interesting behaviour of a litmus test can be characterised by a cycle formed of relations: *e.g.* the **iriw** test of Fig. 4 can be built from the cycle $\xrightarrow{\text{rfe}}; \xrightarrow{\text{fenced}}; \xrightarrow{\text{fre}}; \xrightarrow{\text{rfe}}; \xrightarrow{\text{fenced}}; \xrightarrow{\text{fre}}$. The computed outcome ensures the input cycle appears in at least one of the execution witnesses of the test. If the outcome is observed, then at least one subsequence in the cycle is not global, *i.e.* not in $\xrightarrow{\text{ghb}}$: *e.g.* if the `fence` of Fig. 4 orders pairs of loads and since $\xrightarrow{\text{ab}}$ and $\xrightarrow{\text{fr}}$ are global, then $\xrightarrow{\text{rfe}}; \xrightarrow{\text{fenced}} \not\subseteq \xrightarrow{\text{ghb}}$, *i.e.* the `fence` is not A-cumulative.

We call sequences of relations *relaxations* and give them a concrete syntax (see Fig. 7 and 8). Thus Rfe represents a $\xrightarrow{\text{rfe}}$ arrow, Fre a $\xrightarrow{\text{fre}}$ arrow, and DpdR a $\xrightarrow{\text{dp}}$ (Dp) arrow targeting a read (R), with different (d) source and target locations.

diy needs to be specified which relaxations are considered global and which are not. When specified a pool of global relaxations, a single non-global relaxation, and a size n (*i.e.* the number of relaxations arrows in the cycle, *e.g.* 6 for **iriw**), diy generates cycles up to size n that contains at least one occurrence of the non-global relaxation. If no non-global relaxation is specified, diy generates cycles up to size n containing the specified global relaxations. When the cycles generation is done, diy computes litmus tests from these cycles, as detailed in the following.

4.2 Code generation

We show here how we generate a Power litmus test from a given cycle of relaxations by an example below. The complete algorithm for code generation is available online[2]. We write $\_$ for the information not yet set by diy: $\_\_\_$ is an undetermined event, $W\_\_$ a write with 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:

$$(a)\_\_\_ \xrightarrow{\text{Rfe}} (b)\_\_\_ \xrightarrow{\text{DpdR}} (c)\_\_\_ \xrightarrow{\text{Fre}} (d)\_\_\_ \xrightarrow{\text{Rfe}} (e)\_\_\_ \xrightarrow{\text{DpdR}} (f)\_\_\_ \xrightarrow{\text{Fre}} (a)$$

.

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. We pick an event e which is the target of a relaxation specifying a location change. If there is none, generation fails. Otherwise, a linear scan starting from e sets the locations. At the end of the scan, if e and its predecessor have the same location (*e.g.* $\xrightarrow{\text{Rfe}} e \xrightarrow{\text{PodRW}}$), generation fails. As $\xrightarrow{\text{DpdR}}$ specifies a location change (*i.e.* we pick (c)), we 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), changing location between (e) and (f):

$$(c)Rx\_ \xrightarrow{\text{Fre}} (d)Wx\_ \xrightarrow{\text{Rfe}} (e)Rx\_ \xrightarrow{\text{DpdR}} (f)Ry\_ \xrightarrow{\text{Fre}} (a)Wy\_ \xrightarrow{\text{Rfe}} (b)Ry\_ \xrightarrow{\text{DpdR}} (c)$$

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 first write in each sequence is given value 1, the second one 2, *etc.* 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. *Significant reads* are the sources of $\xrightarrow{\text{fr}}$ and the targets of $\xrightarrow{\text{rf}}$ edges. We associate them 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. Non significant reads do not appear in the test condition. All the reads are significant here:

$$(c)Rx0 \xrightarrow{\text{Fre}} (d)Wx1 \xrightarrow{\text{Rfe}} (e)Rx1 \xrightarrow{\text{DpdR}} (f)Ry0 \xrightarrow{\text{Fre}} (a)Wy1 \xrightarrow{\text{Rfe}} (b)Ry1 \xrightarrow{\text{DpdR}} (c)$$

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

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

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

Fig. 5. iriw with dependencies in Power assembly

6. We generate the litmus test given in Fig. 5 for Power. We add *e.g.* a `xor` instruction between the instructions associated with the events (b) and (c) to implement the dependency required by the $\xrightarrow{\text{DpdR}}$ relation between them.

The test in Fig. 5 actually is a Power implementation of **iriw** [9] with dependencies. diy recovers indeed many classical tests, such as **rwc** [9] (see also Fig. 8).

5 Case study: the Power architecture

We now instantiate the formalism of Sec. 2 for Power by adding *register events* to reflect register accesses [22], and *commit events* to express branching decisions. $\mathbb{C}$ is the set of commits, and c is an element of $\mathbb{C}$. We handle three barrier instructions : `isync`, `sync` and `lwsync`. We distinguish the corresponding events by the eponymous predicates, *e.g.* is-isync. An execution witness includes an additional *intra-instruction causality* relation $\xrightarrow{\text{iico}}$: *e.g.* executing the indirect load `lwz r1, 0(r2)` (`r2` holding the address of a memory location `x` containing 1) creates three events $(a)\,\mathrm{R}r_2\mathrm{x}$, $(b)\,\mathrm{Rx1}$ and $(c)\,\mathrm{W}r_1 1$, such that $(a) \xrightarrow{\text{iico}} (b) \xrightarrow{\text{iico}} (c)$. Moreover, $\xrightarrow{\text{rf}}$ now also relates register events: we write $\xrightarrow{\text{rf-reg}}$ the subrelation of $\xrightarrow{\text{rf}}$ relating register stores to register loads that read their values.

Preserved program order We present in Fig. 6(a) the definition of $\xrightarrow{\text{ppo-ppc}}$, induced by lifting the ordering constraints of a processor to the global level (where $+$ is the transitive closure). This is a formal presentation of the *data dependencies* ($\xrightarrow{\text{dd}}$) and *control dependencies* ($\xrightarrow{\text{ctrl}}$ and $\xrightarrow{\text{isync}}$) of [21, p. 661] which allows loads to be speculated if no `isync` is added after the branch but prevents stores from being speculated in any case. This is similar to Sparc RMO [24, V9, p. 265].

Read-from maps Since Power allows store buffering [21, p.661], $\xrightarrow{\text{rfi}}$ is not global. Running **iriw** with data dependencies (Fig. 5) on Power reveals that $\xrightarrow{\text{rfe}}$ is not global either. This is the main particularity of the Power architecture.

Barriers We define in Fig. 6 (b) the `sync` barrier [21, p. 700] as the SC-restoring A-cumulative barrier of Sec. 3 extended to B-cumulativity. Power features another cumulative barrier [21, p. 700], `lwsync`, defined in Fig. 6 (b). `lwsync` acts as `sync` except on store-load pairs, in both the base and cumulativity cases.

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

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

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

(a) Preserved program order

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

(b) Barrier sync

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

(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. 6. A Power model

Experiment diy generated 800 Power tests and ran them up to 1e12 times each on 3 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. The detailed protocol and results are available[1].

Following our model, we assumed $\xrightarrow{\text{ws}}$, $\xrightarrow{\text{fr}}$, $\xrightarrow{\text{ppo-ppc}}$ and $\xrightarrow{\text{ab-ppc}}$ to be global and tested it by computing *safe* tests whose input cycles only include relaxations we suppose global, *e.g.* $\xrightarrow{\text{SyncdWW}};\xrightarrow{\text{Wse}};\xrightarrow{\text{SyncdWR}};\xrightarrow{\text{Fre}}$. We ran the tests supposed to, according to our model, exhibit relaxations. These tests are given in Fig. 7 (where M stands for million). We observed all of them at least on one machine, which corresponds with our model. For each relaxation observed on a given machine, we write the highest number of outcomes. When a relaxation is not observed, we write the total of outcomes: thus we write *e.g.* 0/16725M for PodRR on `vargas`.

For each machine, we observed the number of runs required to exhibit the least frequent relaxation (*e.g.* 32 million for Rfe 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. Yet, `hpcx` exhibits non-*SC* behaviours for some A-cumulativity tests, including classical ones [9] like **iriw** with `sync` instructions on P_0 and P_1(see Fig. 8). We understand that this is due to an erratum in the Power 5 implementation. IBM is providing a workaround, replacing the `sync` barrier by a short code sequence [Personal Communication], and our testing suggests this does regain *SC* behaviour for the examples in question (e.g. with 0/4e10 non-*SC*

Relaxation	Definition[a]	`hpcx`	`squale`	`vargas`
PosRR	$r_\ell \xrightarrow{\text{po}} r'_\ell$	2/40M	3/2M	0/4745M
PodRR	$r_\ell \xrightarrow{\text{po}} r'_{\ell'}$	2275/320M	12/2M	0/16725M
PodRW	$r_\ell \xrightarrow{\text{po}} w'_{\ell'}$	8653/400M	178/2M	0/6305M
PodWW	$w_\ell \xrightarrow{\text{po}} w'_{\ell'}$	2029/4M	2299/2M	2092501/32M
PodWR	$w_\ell \xrightarrow{\text{po}} r'_{\ell'}$	51085/40M	178286/2M	672001/32M
Rfi	$\xrightarrow{\text{rfi}}$	7286/4M	1133/2M	145/32M
Rfe	$\xrightarrow{\text{rfe}}$	177/400M	0/1776M	9/32M
LwSyncsWR	$w_\ell \xrightarrow{\text{lwsync}} r'_\ell$	243423/600M	2/40M	385/32M
LwSyncdWR	$w_\ell \xrightarrow{\text{lwsync}} r'_{\ell'}$	103814/640M	11/2M	117670/32M
ACLwSyncsRR	$w_\ell \xrightarrow{\text{rfe}} r'_\ell \xrightarrow{\text{lwsync}} r''_\ell$	11/320M	0/960M	1/21M
ACLwSyncdRR	$w_\ell \xrightarrow{\text{rfe}} r'_\ell \xrightarrow{\text{lwsync}} r''_{\ell'}$	124/400M	0/7665M	2/21M
BCLwSyncsWW	$w_\ell \xrightarrow{\text{lwsync}} w'_\ell \xrightarrow{\text{rfe}} r''_\ell$	68/400M	0/560M	2/160M
BCLwSyncdWW	$w_\ell \xrightarrow{\text{lwsync}} w'_{\ell'} \xrightarrow{\text{rfe}} r''_{\ell'}$	158/400M	0/11715M	1/21M

[a] Notation: r_ℓ (w_ℓ) is a read (write) event with location ℓ.

Fig. 7. Selected results of the `diy` experiment matching our model

Cycle	`hpcx`	Name in [9]
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. 8. Anomalies observed on Power 5

results for **iriw**). We understand also that the erratum should not be observable for conventional lock-based code and that Power 6 is not subject to it; the latter is consistent with our testing on `vargas`.

6 Related Work

Formal memory models roughly fall into two classes: operational models and axiomatic models. Operational models, e.g. [25, 15], 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 [5, 24, 16].

One generic axiomatic model related to ours is Nemos [26]. 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 [3]. 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 [12] and Adir et al. [2, 1]. 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. The latter developed an axiomatic model for a version of Power before cumulative barriers [1]; their testing [2] 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.

7 Conclusion

We present here a general class of axiomatic memory models, extending smoothly from *SC* to a highly relaxed model for Power processors. We model their relaxation of store atomicity without requiring multiple write events per store [16], or a view order per processor [12, 1, 21, 6]. Our principal validity condition is simple, just an acyclicity check of the global happens before relation. This check is already known for *SC* [18], and recent verification tools use it for architectures with store buffer relaxation [14, 10]. Our Power model captures key aspects of the behaviour of cumulative barriers, though we do not regard it as definitive: on the one hand there are known tests for which the model is too weak w.r.t. our perception of the architect's intent (particularly involving the lightweight barrier `lwsync`); on the other hand, given that we rely heavily on black-box testing, it is hard to establish confidence that there are not tests that would invalidate our model. Despite that, our automatic test generation based on the model succeeds in generating interesting tests, revealing a rare Power 5 implementation erratum for barriers in lock-free code. This is a significant advance over reliance on hand-crafted litmus tests.

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

References

1. A. Adir, H. Attiya, and G. Shurek. Information-Flow Models for Shared Memory with an Application to the PowerPC Architecture. In *TPDS*, 2003.

2. A. Adir and G. Shurek. Generating Concurrent Test-Programs with Collisions for Multi-Processor Verification. In *HLDVT*, 2002.
3. S. V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, 29:66–76, 1995.
4. M. Ahamad, R. A. Bazzi, R.John, P. Kohli, and G. Neiger. The Power of Processor Consistency. In *SPAA*, 1993.
5. Alpha Architecture Reference Manual, Fourth Edition, 2002.
6. ARM Architecture Reference Manual (ARMv7-A and ARMv7-R), April 2008.
7. Arvind and J.-W. Maessen. Memory Model = Instruction Reordering + Store Atomicity. In *ISCA*. IEEE Computer Society, 2006.
8. Y. Bertot and P. Casteran. *Coq'Art*. Springer Verlag, EATCS Texts in Theoretical Computer Science, 2004.
9. H.-J. Boehm and S.V. Adve. Foundations of the C++ Concurrency Memory Model. In *PLDI*, 2008.
10. S. Burckhardt and M. Musuvathi. Effective Program Verification for Relaxed Memory Models. In *CAV*, 2008.
11. J. Cantin, M. Lipasti, and J. Smith. The Complexity of Verifying Memory Coherence. In *SPAA*, 2003.
12. W. W. Collier. *Reasoning About Parallel Architectures.* Prentice-Hall, 1992.
13. M. Dubois and C. Scheurich. Memory Access Dependencies in Shared-Memory Multiprocessors. *IEEE Transactions on Software Engineering*, 16(6), June 1990.
14. 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.
15. 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.
16. A Formal Specification of Intel Itanium Processor Family Memory Ordering, October 2002. Intel Document 251429-001.
17. L. Lamport. How to Make a Correct Multiprocess Program Execute Correctly on a Multiprocessor. *IEEE Trans. Comput.*, 46(7):779–782, 1979.
18. A. Landin, E. Hagersten, and S. Haridi. Race-free interconnection networks and multiprocessor consistency. *SIGARCH Comput. Archit. News*, 19(3):106–115, 1991.
19. J. Manson, W. Pugh, and S. V. Adve. The Java Memory Model. In *POPL*, 2005.
20. S. Owens, S. Sarkar, and P. Sewell. A Better x86 Memory Model: x86-TSO. In *TPHOL*, 2009.
21. Power isa version 2.06, 2009.
22. 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.
23. 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.
24. Sparc Architecture Manual Versions 8 and 9, 1992 and 1994.
25. Y. Yang, G. Gopalakrishnan, and G. Lindstrom. UMM: an Operational Memory Model Specification Framework with Integrated Model Checking Capability. In *CCPE*, 2007.
26. Y. Yang, G. Gopalakrishnan, G. Linstrom, and K. Slind. Nemos: A Framework for Axiomatic and Executable Specifications of Memory Consistency Models. *IPDPS*, 2004.