Form Methods Syst Des (2012) 41:178–210
DOI 10.1007/s10703-012-0161-5

A formal hierarchy of weak memory models

Jade Alglave

Published online: 27 June 2012

Abstract We present in this paper a formal generic framework, implemented in the Coq proof assistant, for defining and reasoning about weak memory models. We first present the three axioms of our framework, with several examples as illustration and justification. Then we show how to implement several existing weak memory models in our framework, and prove formally that our implementation is equivalent to the native definition for each of these models.

Keywords Weak memory models · Semantics · Formal proofs

1 Introduction

When writing a concurrent program, one often expects (or would like) it to behave according to L. Lamport's *Sequential Consistency* (SC) [30], where:

> *[...] the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.*

For example, most of the concurrent verification work supposes SC as the memory model, probably because multiprocessors were not mainstream until recently. Nowadays however, since multiprocessors are widespread, there is a recrudescent interest in such issues. Indeed, as exposed by S. Adve and H.-J. Boehm in [6]:

J. Alglave (✉)
INRIA, Rocquencourt, France
e-mail: jade.alglave@comlab.ox.ac.uk

J. Alglave
Oxford University, Oxford, UK

J. Alglave
Queen Mary University of London, London, UK

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) A program

$(a)(b)(c)(d)$	$\mathrm{r1} = 0 \wedge \mathrm{r2} = 1$
$(c)(d)(a)(b)$	$\mathrm{r1} = 1 \wedge \mathrm{r2} = 0$
$(a)(c)(b)(d)$ $(a)(c)(d)(b)$ $(c)(a)(b)(d)$ $(c)(a)(d)(b)$	$\mathrm{r1} = 1 \wedge \mathrm{r2} = 1$

(b) The three SC outcomes for this program and their associated interleavings

Fig. 1 An example

*The problematic transformations (*e.g.*, reordering accesses to unrelated variables [. . .]) never change the meaning of single-threaded programs, but do affect multi-threaded programs [. . .].*

1.1 Weak memory models

As an illustration of the subtleties induced by modern multiprocessors on concurrent code, consider the program given in Fig. 1(a), written in pseudo code. On P_0, we start with a store of value 1 to the memory location x, labeled (a), followed in program order by a load from memory location y into register `r1`, labeled (b). On P_1, we have a store of value 1 in memory location y, labeled (c), followed in program order by a load from memory location x into register `r2`, labeled (d). The registers are private to a processor, while the memory locations are shared.

We wonder whether the specified outcome—where `r1` on P_0 and `r2` on P_1 hold 0 in the end—can be observed if we assume SC as the execution model, given that the memory locations x and y hold 0 initially. Observe that in any interleaving of the instructions, one of the stores must go first, e.g., the store to x on P_0, which means that the load from x on the other thread cannot read 0. Thus SC authorizes only three final outcomes, depicted in Fig. 1(b), together with the corresponding interleavings.

However, for matters of performance, modern processors may provide features that induce behaviours a machine with a SC model would never exhibit, as L. Lamport already exposed in [30]:

For some applications, achieving sequential consistency may not be worth the price of slowing down the processors. In this case, one must be aware that conventional methods for designing multiprocess algorithms cannot be relied upon to produce correctly executing programs.

Consider for example the test given in Fig. 1(a). Although we expect only three possible outcomes when running this test, an x86 machine may exhibit the one which was specified in Fig. 1(a), because the store-load pairs on each processor may be reordered. Therefore, the load (b) on P_0 may occur before the store (c) on P_1: in that case, the load (b) reads the initial value of y, which is 0. Similarly, the load (c) on P_1 may occur before the store (a) on P_0, in which case (c) reads the initial value of x, which is 0. Thus, we obtain `r1=0` and `r2=0` as the final state.

Hence we cannot assume SC as the execution model of an x86 machine. A program running on a multiprocessor behaves w.r.t. the *memory model* of the architecture. The memory models we studied are said to be *weak*, or *relaxed* w.r.t. to SC, because they allow more behaviours than SC. For example, such models may allow *instruction reordering* [5, 14]: reads and writes may not be preserved in the program order, as we just saw with the example of Fig. 1(a). Some of them [2, 29] also relax the *store atomicity* [5, 14] constraint. A write may not be available to all processors at once: it could be at first initiated by a given processor, then committed to a store buffer or a cache, and finally *globally performed* to memory [22], at which point only it will be available for all processors to see. Hence the value of a given write may be available to certain processors sooner than to others.

Therefore one needs to understand precisely the definition and consequences of a given memory model in order to predict the possible outcomes of a running program. But some public documentation [2, 27] lack formal definitions of these models. The effort of writing correct concurrent programs is increased by the absence of precise, if not formal, definitions.

1.2 Modelling

The goal of the present paper is to gather formal specifications of a handful of architectures into a single document, along with a uniform presentation that allows for straightforward comparison amongst them.

To do so, we tried to identify as few concepts as possible to describe a whole family of architectures, because we wanted to be able to describe precisely the reason why a given execution is allowed on a given architecture, and not on another one. From our reading of the existing documentations (assuming that they are sound, which should not be taken for granted, as explained for example in [40]) these reasons are often unclear, and sometimes several reasons are entangled, due to the inherent lack of rigour of writing a specification in natural language. Thus, explaining in clear and rigorous terms why an execution is allowed becomes difficult because we do not have the appropriate language, or concepts, to describe it.

This work is an attempt at providing clear, general concepts to describe and reason about memory models. We provide three axioms (namely the consensus, uniproc and thin checks, as described in Sects. 4 and 5) that are enough to describe a whole family of architectures. We show that store atomic architectures, such as SC or TSO, belong to this family. Perhaps surprisingly, we are also able to model the store atomicity relaxation as exhibited by very relaxed architectures such as Power and ARM.

Modern architectures such as x86, Power and ARM, do not have a documentation that is rigorous enough to lend themselves to immediate formalisation. Itanium [29] is a notable exception. Yet, the style of formalisation of [29] is rather different from the one that we adopt here. Hence we do not discuss this architecture here, leaving the comparison with this framework for future work.

Other work by ourselves and colleagues address the validation of models w.r.t. actual architectures, either via testing [11, 12], or discussion with processor vendors [33, 35]. At the time of writing, x86 has been established to correspond to the TSO model, as exposed in [33]. Since, as we show in Sect. 7.2.2.2, our framework embraces TSO, the results presented here apply to x86 as well. The formalisation of Power and ARM is still an ongoing work [10, 11, 35]. We have proposed a Power model which is an instance of the framework presented here, but we will refer the interested reader to the corresponding papers for more details [10, 11].

1.3 Contribution

We present here a generic framework designed to describe weak memory models. Although memory models issues arise at all levels of the software implementation stack, we focus here exclusively on the hardware level. Though some public documentation, e.g. Intel [27] and Power [2], lack formal definitions of these models, others—such as Alpha [13] and Sparc [1]—provide a precise definition of the model that their processors exhibit. Our generic framework is widely inspired of the common style of Alpha and Sparc's documentations, in that we use a *global time axiomatic* model. However, Alpha and Sparc consider the stores to be *atomic*. We adapted the style of their model to allow the store atomicity relaxation, as does e.g. Power and ARM. In addition, we took care to minimise the number and the complexity of our axioms, so that they are easier to understand.

We start with a presentation of some related work in Sect. 2. We present in Sects. 3, 4 and 6 the objects, terms and axioms of our framework. We illustrate in Sect. 7 how to instantiate its parameters to produce several well known models, namely *Sequential Consistency* [30], the Sparc hierarchy (i.e. TSO, PSO and RMO) [1], and Alpha [13].

This is an extended version of Sects. 2 and 3 of [11], which appeared as Part I (Chaps. 3, 4 and 5) of the author's PhD thesis [8]. All our definitions and results are formalised in the Coq proof assistant [15]. The associated development can be found at the following address: http://moscova.inria.fr/~alglave/wmm.

2 Related work

In describing memory models, several styles of mathematical description coexist. We first present several generic weak memory models. Then we examine related work according to the view of memory they use, either a unique *global-time* view as in the present work, or using one *view order* per processor. Finally, we distinguish models according to their style, either *axiomatic* like our model, *operational*, or specifications of weak memory models as *program transformations*.

Generic models The work that is the closest to ours is probably W. Collier's [21]. He presents several abstract models in terms of the relaxations (w.r.t. SC) they allow. However he does not address the store atomicity relaxation.

S. Adve and K. Gharachorloo's tutorial gives a categorisation of memory models, in which they give intuition about the relaxations in terms of the actual hardware, i.e. store buffers and cache lines. By contrast, we choose in the present work to abstract from hardware implementation details.

S. Adve [4] and K. Gharachorloo [24] both present in their theses a generic framework. S. Adve's work focuses on the notion of *data race freeness*, and defines and studies models which enforce the *data race freeness guarantee*. We do not address this issue in the present work. In other work, e.g., [8, 10], we chose to examine this property on top of our framework, and see which conditions enforce this guarantee for its instances, instead of building a model with a hard-wired data race free guarantee. K. Gharachorloo's work focuses on the implementation and performance of several weak memory models. We choose to give an abstract view of the memory, because we want to provide a model in which the programmer does not have to care about the minute details of the implementation—which are often secret—to write correct programs.

Finally, the Nemos [39] framework covers a broad range of models including Itanium as the most substantial example. Itanium [29] is rather different from the models that we consider here, and from the Power model that we present in [11]; we do not know whether the present work could handle such a model. Indeed, Itanium uses several events per instruction, whereas we represent here instructions by only one memory event. Moreover, Itanium's model specifies the semantics not only of stores, loads and fences, but also of *load-acquire* and *store-release* instructions. By contrast, we chose to specify the semantics of more atomic constructions, and build the semantics of derived constructions on top of them, as developed in [10].

Global-time vs. view orders We can distinguish memory models w.r.t. the view of memory they present. Such models are either in terms of a global time line in which the memory events are embedded, or provide one view order per processor.

Most of the documentations that provide a formal model, e.g. Alpha [13] and Sun [1], are in terms of a global time line. We believe this provides a usable model to the programmer, because it abstracts from the implementation's details. Moreover such a model allows the vendor to provide a formal and usable model without revealing the secrets of the implementation.

Some memory models are in terms of view orders, e.g. [3] and the Power documentation [2]. A. Adir et al.'s work focuses on the PowerPC model, and presents numerous axioms describing a pre-cumulativity (pre Power 4) version of Power.

Axiomatic vs. operational Formal models roughly fall into two classes: operational models and axiomatic models. Operational models, e.g., [17, 26, 38], are abstractions of actual machines composed of idealised hardware components such as queues. They seem appealingly intuitive and offer a relatively direct path to simulation, at least in principle.

Axiomatic models focus on the segregation of allowed and forbidden behaviours, usually by constraining various order relations on memory accesses; they are well adapted for model exploration, as we do in [11]. Several of the most formal vendor specifications have been in this style [1, 13, 29].

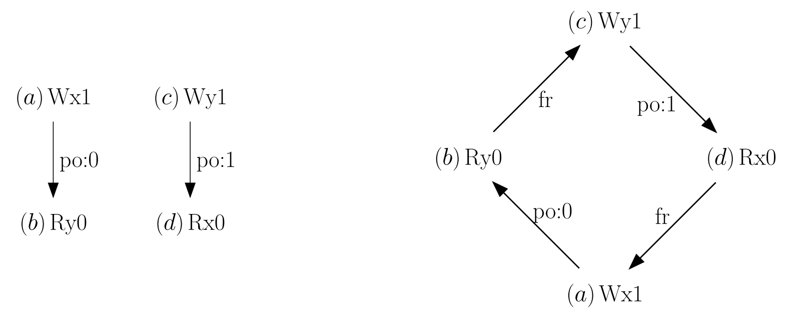
Memory models as program transformations Another style of weak memory models' specification has recently emerged, e.g. in S. Burckhardt et al.'s work [19] or R. Ferreira et al.'s [23]. This line of research specifies weak memory models as program transformations. Instead of specifying a transition system as in an operational style, rewriting rules apply to the program as a whole, to represent the effect of the memory model on this program's behaviour. This approach addresses only a limited store atomicity relaxation.

3 From events to execution witnesses

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

3.1 Informal overview of our approach

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` initially. In this paper, we write the instructions in pseudo-code; for example $\texttt{x} \leftarrow \texttt{1}$ is a store of value `1` into memory location `x`,



(a) Events and program order (b) An Execution Witness

Fig. 2 An event structure and an execution witness for the program of Fig. 1

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 labeled with letters—e.g., in Fig. 1(a), the first line, which holds `x` ← `1`, is labeled (a).

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. 2(b).

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. 2(a) the memory events of one candidate execution of the program of Fig. 1(a): the write event (a) Wx1 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 po relation between the corresponding events in Fig. 2(a). Other relations represent the interaction with memory: the reads-from relation rf indicates which write the value of each read event comes from; and the write serialisation 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 consider an execution of a given program to be valid when the read and write memory events associated with the instructions of the program follow a single global *consensus*, i.e. can be embedded in a single partial order. Thus, we define the validity of a given candidate execution as acyclicity checks of certain unions of these relations.

This consensus represents the order in which these events are *globally performed*, which means that we embed them in the order when we reach the point in time where all processors involved have to take these events into account. In this paper, we consider reads to be globally performed at once, whereas writes may not become visible to all processors at once. This allows us to model both store buffering and the store atomicity relaxation, as we expose below.

However, 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. 2(b). Typically, this cycle will exhibit the fact that the execution that we choose to depict is invalid in the Sequential Consistency (SC) model [30]. The execution in Fig. 2(b) is allowed in TSO and in Power, but not in SC, where at least one of the reads would have to read `1`.

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. 2(b), 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 po in Fig. 2(a)) 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. 2(b) does not forbid the execution on some x86 machines, and furthermore that the x86 model allows the reordering of write-read pairs.

In the present paper, we focus exclusively on the Allowed/Forbidden case. For the Observed/Not observed case, we refer the interested reader to other work by ourselves and colleagues, which address the validation of theoretical models, either via testing [11, 12], or discussion with processor vendors [33, 35].

3.2 Events and program order

As sketched above, rather than dealing directly with programs, our models are expressed in terms of the *events* $\mathbb{E}$ occurring in a candidate execution. A *memory event* m represents a memory access, specified by its direction (write or read), its location $\mathrm{loc}(m)$, its value $\mathrm{val}(m)$, its processor $\mathrm{proc}(m)$, and a unique label. For example, the store to x marked (a) in Fig. 1(a) generates the event (a) Wx1 in Fig. 2. Henceforth, we write r (resp. w) for a read (resp. write) event. We write $\mathbb{M}_\ell$ (resp. $\mathbb{R}_\ell$, $\mathbb{W}_\ell$) for the set of memory events (resp. reads, writes) to a location ℓ (we omit ℓ when quantifying over all of them). We give a table of notations for these sets of events, and the corresponding Cartesian products in Appendix.

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

The *program order* po is a *linear order*[1] 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, $(e_1, e_2) \in \mathsf{po}$ (or $e_1 \xrightarrow{\mathsf{po}} e_2$) means that a sequential processor would execute i_1 before i_2. When instructions may perform several memory accesses, we take intra-instruction dependencies [34] into account to build a more precise order.

Hence we describe a program by an *event structure*,[2] which collects the memory events issued by the instructions of this program, and the program order relation, which lifts the program order between instructions to the events' level:

Definition 1 (Event structure)

$$E \triangleq (\mathbb{E}, \mathsf{po})$$

Consider for example the test given in Fig. 1(a). We give in Fig. 2(a) an associated event structure. For example, to the store instruction marked (a) on P_0, we associate the write event (a) Wx in Fig. 2. To the load instruction marked (b) on P_0, we associate the read event (b) Ry in Fig. 2. Since these two instructions are in program order on P_0, the associated events are related by the po relation. The reasoning is similar on P_1.

3.3 Execution witnesses

Although po conveys important features of a program execution, e.g., branch resolution, it does not characterise an execution. Indeed, on a weak memory model, the events in program order may be reordered in an execution. Moreover, we need to describe the communication between distinct processors during the execution of a program. Hence, in order to describe an execution, we postulate two relations rf and ws over memory events.

3.3.1 Read-from map

We write $(w, r) \in \mathsf{rf}$ (or $w \xrightarrow{\mathsf{rf}} r$) to mean that r loads the value stored by w (so w and r must share the same location). In any execution, given a read r there exists a unique write w such that $(w, r) \in \mathsf{rf}$ (w can be an *init* store when r loads from the initial state). Thus, rf must be well formed following the wf-rf predicate:

Definition 2 (Well-formed read-from map)

$$\text{wf-rf}(\mathsf{rf}) \triangleq \left(\mathsf{rf} \subseteq \bigcup_{\ell,v} \mathbb{WR}_{\ell,v}\right) \wedge \left(\forall r.\exists! w.(w, r) \in \mathsf{rf}\right)$$

Consider the example given in Fig. 3(a). In the associated execution given in Fig. 3(b), the read (c) from x on P_1 reads its value from the write (b) to x on P_1. Hence we have a rf relation between them, depicted in the execution: $(b, c) \in \mathsf{rf}$.

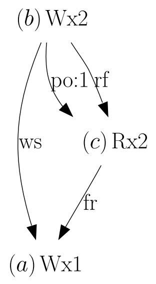
[1]By linear order, we mean a relation r that is *irreflexive* (i.e. $\forall x.\neg((x, x) \in r)$), *transitive* (i.e. $\forall xyz.(x, y) \in r \wedge (y, z) \in r \Rightarrow (x, z) \in r$) and *total* (i.e. $\forall xy.(x, y) \in r \vee (y, x) \in r$).

[2]Note that these are not G. Winskel's event structures.

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

Allowed: `r1=2; x=1`

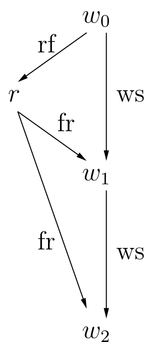
(a) A program



(b) An execution witness

Fig. 3 A program and a candidate execution

Fig. 4 fr proceeds from rf and ws



3.3.2 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 ws as the union of the coherence orders for all memory locations, which must be well formed following the wf-ws predicate, where linear(r, S) means that the relation r is a linear order over the set S:

Definition 3 (Well-formed write serialisation)

$$\text{wf-ws}(\mathsf{ws}) \triangleq \left(\mathsf{ws} \subseteq \bigcup_{\ell} \mathbb{WW}_{\ell}\right) \wedge \left(\forall \ell.\, \text{linear}(\mathsf{ws}.\mathbb{WW}_{\ell})\right)$$

Consider the example given in Fig. 3(a). In the associated execution given in Fig. 3(b), the write (b) to x on P_1 hits the memory before the write (a) to x on P_0. Hence we have a ws relation between them, depicted in the execution: $(b, a) \in \mathsf{ws}$.

As we shall see in Sect. 4, we will embed the write events in our global consensus according to the write serialisation.

3.3.3 From-read map

We define the derived relation fr [7] which gathers all pairs of reads r and writes w such that r reads from a write that is before w in ws, as depicted in Fig. 4. Intuitively, a read r is in fr with a write w when r reads from a write that hit the memory before w did:

Definition 4 (From-read map)

$$(r, w) \in \mathsf{fr} \triangleq \exists w'.\big(w', r\big) \in \mathsf{rf} \wedge \big(w', w\big) \in \mathsf{ws}$$

Consider the example given in Fig. 3(a). In the associated execution given in Fig. 3(b), the write (b) to x on P_1 hits the memory before the write (a) to x on P_0, i.e. $(b, a) \in \mathsf{ws}$. Moreover, the read (c) from x on P_1 reads its value from the write (b) to x on P_1, i.e. $(b, c) \in \mathsf{rf}$. Hence we have a fr relation between (c) and (a) (i.e. $(c, a) \in \mathsf{fr}$) because (c) reads from a write which is older than (a) in the write serialisation.

As we shall see in Sect. 4, we will use the fr relation to include the read events in our global consensus.

3.3.4 All together

Given a certain event structure E, we call the rf, ws and fr relations the *communication* relations, and we write com for their union:

Definition 5 (Communication)

$$\mathsf{com} \triangleq \mathsf{rf} \cup \mathsf{ws} \cup \mathsf{fr}$$

We define an *execution witness* X associated with an event structure E as:

Definition 6 (Execution witness)

$$X \triangleq (\mathsf{rf}, \mathsf{ws})$$

For example, we give in Fig. 2 an execution witness associated with the program of Fig. 1(a). The set of events is $\{(a), (b), (c), (d)\}$, the program order is $(a, b) \in \mathsf{po}$, $(c, d) \in \mathsf{po}$. Since the initial state is implicitly a write preceding any other write to the same location in the write serialisation, the only communication arrows we have between the events of this execution are $(b, c) \in \mathsf{fr}$ and $(d, a) \in \mathsf{fr}$.

The well-formedness predicate wf on execution witnesses is the conjunction of those for ws and rf. We write $\mathsf{rf}(X)$ (resp. $\mathsf{ws}(X)$, $\mathrm{po}(X)$) to extract the rf (resp. ws, po) relation from a given execution witness X. When X is clear from the context, we may write rf instead of $\mathsf{rf}(X)$ for example.

Definition 7 (Well-formed execution witness)

$$\mathrm{wf}(X) \triangleq \text{wf-rf}\big(\mathsf{rf}(X)\big) \wedge \text{wf-ws}\big(\mathsf{ws}(X)\big)$$

4 Global happens-before

We consider an execution to be valid when we can embed the memory events of this execution is a single global consensus. By global we mean that the memory events are the events relative to memory actions, in a way that every processor involved has to take them into account. Therefore, we do not consider the events relative to store buffers or caches, but rather we wait until these events hit the main memory. Thus, we focus on the history of the system from the main memory's point of view.

Hence, an execution witness is valid if the memory events can be embedded in an acyclic *global happens-before* relation ghb (together with two auxiliary conditions detailed in Sect. 6). This order corresponds roughly to the vendor documentation concept of memory events being *globally performed* [2, 22]: a write in ghb represents the point in global time when this write becomes visible to all processors; whereas a read in ghb represents the point in global time when the read takes place. We will formalise this notion later on, at Sect. 4.3.1.

In order to do so, we present first the choices as to which relations we include in ghb (i.e. which we consider to be in global time). Thereby we define a class of models. In the following, we will call a relation *global* when it is included in ghb. Intuitively, a relation is considered global if the participants of the system have to take it into account to build a valid execution.

In our class of models, ws is always included in ghb. Indeed, the write serialisation for a given location ℓ is by definition the order in which writes to ℓ are globally performed. The relation fr is also always included in ghb. Indeed, as $(r, w) \in \mathsf{fr}$ means that the write w' from which r reads is globally performed before w, it forces the read r to be globally performed (since as we expose in the preamble of the present section a read is globally performed as soon as it is performed) before w is globally performed.

Yet, rf is not necessarily global, as we explain below. In the following, we write grf for the subrelation of rf included in ghb. We write rfi (resp. rfe) for the internal and external rf, when the events in rf are on the same (resp. distinct) processor(s):

Definition 8 (Internal and external read-from map)

$$(w, r) \in \mathsf{rfi} \triangleq (w, r) \in \mathsf{rf} \wedge \mathrm{proc}(w) = \mathrm{proc}(r)$$
$$(w, r) \in \mathsf{rfe} \triangleq (w, r) \in \mathsf{rf} \wedge \mathrm{proc}(w) \neq \mathrm{proc}(r)$$

4.1 Preserved program order

In any given architecture, certain pairs of events in the program order are guaranteed to occur in this order. We postulate a global relation ppo (for *preserved program order*) gathering all such pairs. For example, the execution witness in Fig. 2 is only valid if the writes and reads relative 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 ghb: $(a) \xrightarrow{\mathsf{ppo}} (b) \xrightarrow{\mathsf{fr}} (c) \xrightarrow{\mathsf{ppo}} (d) \xrightarrow{\mathsf{fr}} (a)$. Such a cycle contradicts the validity of the execution, hence the execution depicted in Fig. 2 is not valid on an architecture such as SC, which maintains the write-read pairs in program order.

An example of load-load and store-store reordering Consider the example given in Fig. 5, which appears in the Intel documentation [28, §8.2.3.2, pp. 8–13]. On P_0, we write the value 1 into x, then write 1 into y. On P_1, we load from y into register `r3` and finally load from x into register `r4`. On an architecture that allows the reordering of either write-write or read-read pairs, e.g., RMO, Power and ARM, we can observe the specified outcome, where the value of y written by P_0 is seen by P_1, as witnessed by the result `r3=1`, but not the value of x, as witnessed by the result `r4=0`. This could happen if the two writes on P_0 are reordered, or if the two reads on P_1 are reordered (i.e. not in ppo). This could also happen if the store atomicity is relaxed, but we explain this later (see Sect. 4.2.2).

On the execution given in Fig. 5, we can see the cycle $(a) \xrightarrow{\mathsf{po}} (b) \xrightarrow{\mathsf{rfe}} (c) \xrightarrow{\mathsf{po}} (d) \xrightarrow{\mathsf{fr}} (a)$. We know that fr is always global by hypothesis. Let us now assume that we are on an architecture

Fig. 5 An example of load-load and store-store reordering

P_0	P_1
(a) `x ← 1`	(c) `r3 ← y`
(b) `y ← 1`	(d) `r4 ← x`

Observed? `r3=1; r4=0`

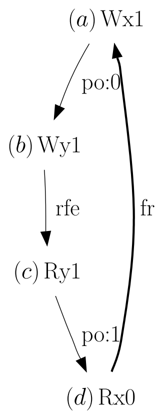
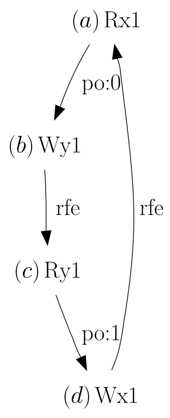


Fig. 6 An example of load-store reordering

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

Observed? `r1=r2=1;`



where the store atomicity is not relaxed (which we model by some fragment of rf being global—see below, Sect. 4.2.2). Then, the only possibility for this execution to be allowed (or in other terms for this cycle to be non-global), is for one of the po relations (a, b) or (c, d) to be non-global. This corresponds to the reordering scenarios exposed above.

An example of load-store reordering Consider the example given in Fig. 6, which appears in the Intel documentation [28, §8.2.3.3, pp. 8–13]. On P_0, we load from x into register `r1`, then write 1 into y. On P_1, we load from y into register `r2` and finally write 1 to x. On an architecture that allows the reordering of either read-write pairs, e.g., RMO, Power and ARM, we can observe the specified outcome, where the value of y written by P_0 is seen by P_1, as witnessed by the result `r2=1`, and the value of x written by P_1 is seen by P_0, as witnessed by the result `r1=1`. This could happen if the read-write pair on P_0 is reordered, or symmetrically on P_1. This could also happen if the store atomicity is relaxed, but we explain this later (see Sect. 4.2.2).

On the execution given in Fig. 6, we can see the cycle $(a) \xrightarrow{\text{po}} (b) \xrightarrow{\text{rfe}} (c) \xrightarrow{\text{po}} (d) \xrightarrow{\text{rfe}} (a)$. Let us now assume that we are on an architecture where the store atomicity is not relaxed (which we model by the some fragment of rf being global—see below, Sect. 4.2.2). Then, the only possibility for this execution to be allowed (or in other terms for this cycle to be non-global), is for one of the po relations (a, b) or (c, d) to be non-global. This corresponds to the reordering scenarios exposed above.

store-fwd

P_0	P_1
(a) `x ← 1`	(d) `y ← 1`
(b) `r1 ← x`	(e) `r3 ← y`
(c) `r2 ← y`	(f) `r4 ← x`

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

$$\begin{tikzcd}
 & (d)\,\mathrm{Wy1} \arrow[r, "\mathrm{rfi}"] & (e)\,\mathrm{Ry1} \arrow[dr, "\mathrm{po}"] & \\
(c)\,\mathrm{Ry0} \arrow[ur, "\mathrm{fr}"] & & & (f)\,\mathrm{Rx0} \arrow[dl, "\mathrm{fr}"] \\
 & (b)\,\mathrm{Rx1} \arrow[ul, "\mathrm{po}"] & (a)\,\mathrm{Wx1} \arrow[l, "\mathrm{rfi}"] &
\end{tikzcd}$$

Fig. 7 An example of store buffering

4.2 Read-from maps

Writes are not necessarily globally performed at once. Some architectures allow *store buffering* (or *read own writes early* [5]): the processor issuing a given write can read its value before any other participant has access to it. 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 case of *store atomicity relaxation*, or *read others' writes early* [5]).

We said above that ws and fr are always global, i.e. included in ghb, but that rf might not be global. We now explain when rf is global or not.

4.2.1 Store buffering

We model the store buffering by rfi being not included in ghb. Indeed, the communication between a write to a given processor's store buffer and a read from this buffer does not influence the execution of another processor, because the write has not hit the main memory yet. Therefore, this communication, modelled by rfi, is private to the processor issuing the write, and we do not embed it in our global consensus.

Consider the example given in Fig. 7, which appears in the Intel documentation [28, §8.2.3.4, pp. 8–15]. On P_0, we write the value 1 into x, then load it from x into register `r1`. Then, we load the value from y into register `r2`. On P_1, we write to y, then load its value into register `r3` and finally load the value from x into register `r4`. On an architecture that allows store buffering, e.g., x86, Power and ARM, we can observe the specified outcome, where the value of x written by P_0 is immediately accessible to P_0, as witnessed by the result `r1=1`, but not yet to P_1, as witnessed by the result `r4=0`. This could happen if P_0 reads from x via its store buffer, which has not been flushed to commit the new value of x to memory. The situation is symmetric with P_1 and y.

On the execution given in Fig. 7, we can see the cycle $(a) \xrightarrow{\mathsf{rfi}} (b) \xrightarrow{\mathsf{po}} (c) \xrightarrow{\mathsf{fr}} (d) \xrightarrow{\mathsf{rfi}} (e) \xrightarrow{\mathsf{po}} (f) \xrightarrow{\mathsf{fr}} (a)$. We know that fr is always global by hypothesis. Let us now assume that we are on an architecture where the read-read pairs (b, c) and (e, f) cannot be reordered (which we model by the corresponding fragment of the program order po being global—see above Sect. 4.1). Then, the only possibility for this execution to be allowed (or in other terms for this cycle to be non-global), is for the rfi relation to be non-global. This corresponds to the store buffering scenario exposed above.

iriw

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

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

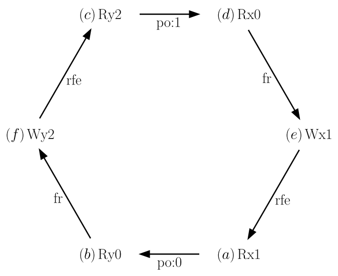


Fig. 8 An example of atomicity relaxation

4.2.2 *Store atomicity relaxation*

Similarly, we model the store atomicity relaxation by rfe being not global. Indeed, the communication between two processors via a shared cache may not influence the execution of another processor, because the write has not hit the main memory yet. Therefore, this communication, modelled by rfe is private to the two communicating processors, and we do not embed it in our global consensus.

Consider the example given in Fig. 8, which appears in [16]. On P_0, we read from x then from y. On P_1, we read the same locations but in the converse order. On P_2 we write 1 to x, and on P_3 we write 2 to y. On an architecture that relaxes store atomicity, e.g., Power and ARM, we can observe the specified outcome, where the value of x written by P_2 is already accessible to P_0, as witnessed by the result `r1=1`, but not yet to P_1, as witnessed by the result `r4=0`. This could happen if P_0 reads from x via a cache that is shared by P_0 and P_2 only, so that the new value of x is not yet visible to the other pair of processors, namely P_1 and P_3. The situation is symmetric with P_1 and P_3 communicating via a value of y written in a cache shared by them only.

On the execution given in Fig. 8, we can see the cycle $(a) \xrightarrow{\text{po}} (b) \xrightarrow{\text{fr}} (f) \xrightarrow{\text{rfe}} (c) \xrightarrow{\text{po}} (d) \xrightarrow{\text{fr}} (e) \xrightarrow{\text{rfe}} (a)$. We know that fr is always global by hypothesis. Let us now assume that we are on an architecture where the read-read pairs (a, b) and (c, d) cannot be reordered (which we model by the corresponding fragment of the program order po being global—see above Sect. 4.1). Then, the only possibility for this execution to be allowed (or in other terms for this cycle to be non-global), is for the rfe relation to be non-global. This corresponds to the communication via caches scenario exposed above.

Let us now revisit the example given in Fig. 6. In Fig. 6, suppose that we have means to maintain the read-read pairs on P_0 and P_1, either because they are natively maintained by the architecture, like in TSO, or because we put an arithmetic operation in between them to create a dependency, as we could on Power or ARM. In this case, the only possibility for this execution to be allowed (or in other terms for the cycle to be non-global), is for the rfe relation to be non-global.

4.3 Architectures

4.3.1 *Definition*

We call a particular model of our class an *architecture*, written A. We model an architecture by a tuple of functions over executions. Hence we consider an architecture as a filter over executions, which determines which executions are valid and which are not. By abuse of notation, we write ppo (resp. grf) for the function returning the ppo (resp. grf) relation w.r.t. A when given an event structure and execution witness:

Definition 9 (Architecture)

$$A \triangleq (\mathsf{ppo}, \mathsf{grf})$$

We use in the following the notation f_A for a function f over execution witnesses w.r.t. the architecture A. For example, given an event structure E, an associated execution witness X and two architectures A_1 and A_2, we write $\mathsf{ghb}_{A_1}(E, X)$ for the ghb of the execution (E, X) relative to A_1, while $\mathsf{ppo}_{A_2}(E, X)$ returns the ppo of the execution (E, X) relative to A_2. We omit the architecture when it is clear from the context. Finally, we define ghb as the union of the global relations:

Definition 10 (Global happens-before)

$$\mathsf{ghb} \triangleq \mathsf{ppo} \cup \mathsf{ws} \cup \mathsf{fr} \cup \mathsf{grf}$$

4.3.2 Examples of architectures

Sequential Consistency (SC) (see also Sect. 7.2.1) allows no reordering of events (ppo equals po on memory events) and makes writes available to all processors as soon as they are issued (rf is global, i.e. $\mathsf{grf} = \mathsf{rf}$). Thus, the outcome of Fig. 1 cannot be the result of a SC execution. Indeed, the associated execution exhibits the cycle: $(a) \xrightarrow{\mathsf{po}} (b) \xrightarrow{\mathsf{fr}} (c) \xrightarrow{\mathsf{po}} (d) \xrightarrow{\mathsf{fr}} (a)$. Since fr is always in ghb, and since the program order po is included in SC's preserved program order, this cycle is a cycle in $\mathsf{ghb}_{\mathrm{SC}}$, hence we contradict the validity of this execution on SC.

Sun's Total Store Ordering (TSO) (see also Sect. 7.2.2.2) allows two relaxations [5]: *write to read program order*, and *read own write early*. The *write to read program order* relaxation means that TSO's preserved program order includes all pairs but the store-load ones. The *read own write early* relaxation means that TSO's internal read-from maps are not global, i.e. $\mathsf{rfi} \not\subseteq \mathsf{ghb}_{\mathrm{TSO}}$. Moreover, TSO does not relax the atomicity of stores, i.e. $\mathsf{rfe} \subseteq \mathsf{ghb}_{\mathrm{TSO}}$. Thus, the outcome of Fig. 1 can be the result of a TSO execution. Even if the associated execution (E, X) exhibits the cycle $(a) \xrightarrow{\mathsf{po}} (b) \xrightarrow{\mathsf{fr}} (c) \xrightarrow{\mathsf{po}} (d) \xrightarrow{\mathsf{fr}} (a)$, this does not form a cycle in $\mathsf{ghb}_{\mathrm{TSO}}(E, X)$. Indeed, the write-read pairs in $(a, b) \in \mathsf{po}$ on P_0 and $(c, d) \in \mathsf{po}$ on P_1 do not have to be maintained on TSO.

5 Healthiness conditions

We now describe two healthiness conditions (independent of the one presented in the previous section) that *every execution*, on *any architecture of our framework*, should satisfy. In conjunction with the condition exposed in Sect. 4, they form the criterion to decide whether an execution is valid on an architecture of our framework.

5.1 Uniprocessor behaviour

First, we require each processor to respect memory coherence for each location [20] (i.e. the per-location write serialisation): if a processor writes e.g., v then v' to the same location ℓ, then the associated writes w and w' should be in this order in the write serialisation, i.e. w' should not precede w in the write serialisation. We formalise this notion as follows.

Fig. 9 Invalid execution according to the uniproc criterion

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

Forbidden: `x=1; r1=1;`

$$\begin{tikzcd}
& (a)\,\mathrm{Wx1} \arrow[dl, "\mathrm{rf}"'] \\
(b)\,\mathrm{Rx1} \arrow[dr, bend right, "\mathrm{po{:}1}"'] \arrow[dr, "\mathrm{po\text{-}loc}"] & \\
& (c)\,\mathrm{Wx2} \arrow[uu, bend right, "\mathrm{ws}"']
\end{tikzcd}$$

P_0	P_1
(a) `x` $\leftarrow$ `1`	(d) `x` $\leftarrow$ `2`
(b) `r1` $\leftarrow$ `x`	
(c) `r2` $\leftarrow$ `x`	

Observed? `r1=2; r2=1; x=2`

(a) Program

$$\begin{tikzcd}
& (c)\,\mathrm{Rx1} \arrow[dr, "\mathrm{fr}"] & \\
(b)\,\mathrm{Rx2} \arrow[ur, "\mathrm{po{:}0}"] & & (d)\,\mathrm{Wx2} \arrow[ll, "\mathrm{rf}"] \\
& (a)\,\mathrm{Wx1} \arrow[ul, "\mathrm{po{:}0}"] \arrow[ur, "\mathrm{ws}"'] &
\end{tikzcd}$$

(b) An Execution Witness

Fig. 10 Load-load hazard example

5.1.1 *Definition*

We define the relation $\mathsf{po\text{-}loc}$ over accesses to the same location in program order:

$$(m_1, m_2) \in \mathsf{po\text{-}loc} \triangleq (m_1, m_2) \in \mathsf{po} \wedge \mathrm{loc}(m_1) = \mathrm{loc}(m_2)$$

We require $\mathsf{po\text{-}loc}$ to be compatible with com (i.e. $\mathsf{rf} \cup \mathsf{ws} \cup \mathsf{fr}$):

Definition 11 (Uniprocessor check)

$$\mathrm{uniproc}(E, X) \triangleq \mathrm{acyclic}(\mathsf{com} \cup \mathsf{po\text{-}loc})$$

For example, in Fig. 9, we have $(c, a) \in \mathsf{ws}$ (by x final value) and $(a, b) \in \mathsf{rf}$ (by `r1` final value). The cycle $(a) \xrightarrow{\mathsf{rf}} (b) \xrightarrow{\mathsf{po\text{-}loc}} (c) \xrightarrow{\mathsf{ws}} (a)$ invalidates this execution: (b) cannot read from (a) as it is a future value of x in ws.

Note that uniproc corresponds, as we shall see in Sect. 7.2.1, to checking that SC holds per location. This observation is of crucial importance when proving that non-relational data-flow analyses are sound on weak memory models, as we do in [9]. Indeed, this axiom basically guarantees that if one examines a given program from the point of view of a sole memory location, everything appears to be SC. Since non-relational analyses deal with programs on a per-location basis, they are sound on any given model that enjoys this axiom.

5.1.2 *Load-load hazard*

Certain architectures such as RMO allow *load-load hazard*, i.e. two reads from the same location in program order may be reordered. We give in Fig. 10 an example program of load-load hazard: the read (b) from x and the read (c) from x on P_0 have been reordered. The read (c) reads from the write (a) on P_0, hence $(a, c) \in \mathsf{rf}$, which is not depicted in Fig. 10 to ease the reading. The read (b) reads from the write (d) on P_1, hence $(d, b) \in \mathsf{rf}$. Suppose $(a, d) \in \mathsf{ws}$, then, since $(a, c) \in \mathsf{rf}$, we have $(c, d) \in \mathsf{fr}$. Since we have $(b, c) \in \mathsf{po}$, we exhibit a cycle which is a contradiction to uniproc: $(b) \xrightarrow{\mathsf{po}} (c) \xrightarrow{\mathsf{fr}} (d) \xrightarrow{\mathsf{rf}} (b)$. The program order between the two reads (b) and (c) on P_0 is not respected, even though they access the same location x.

To allow load-load hazard, we define the relation $\mathsf{po\text{-}loc_{llh}}$ over accesses to the same location in the program order as $\mathsf{po\text{-}loc}$ except for read-read pairs:

$$(m_1, m_2) \in \mathsf{po\text{-}loc_{llh}} \triangleq (m_1, m_2) \in \mathsf{po} \wedge \mathrm{loc}(m_1) = \mathrm{loc}(m_2) \wedge \neg(m_1 \in \mathbb{R} \wedge m_2 \in \mathbb{R})$$

Then we slightly alter the definition of uniproc to:

Definition 12 (Load-load hazard uniproc)

$$\mathrm{uniproc_{\mathsf{llh}}}(E, X) \triangleq \mathrm{acyclic}(\mathsf{com} \cup \mathsf{po\text{-}loc_{llh}})$$

In the following, we will use the notation uniproc for $\mathrm{uniproc_{\mathsf{llh}}}$. Thus, all our results hold with this weak variant of the uniproc check. Note however that for all the architectures presented here, as well as for Power [11, 35], except RMO, the full version of uniproc holds, on any valid execution.

5.1.3 Discussion

Note that the uniproc check can spare the cost of including certain pairs of events in program order in the preserved program order of an architecture. Consider for example two writes to the same location in program order. They are necessarily (by uniproc) included in ghb. Indeed, such a pair (x, y) is in $\mathsf{po\text{-}loc}$, thus (by uniproc) in $(\mathsf{com})^+$. Moreover, observe that $(\mathsf{com})^+$ is equal to $(\mathsf{com} \cup (\mathsf{ws}; \mathsf{rf}) \cup (\mathsf{fr}; \mathsf{rf}))$ (because $\mathsf{ws}; \mathsf{ws} = \mathsf{ws}$ and $\mathsf{fr}; \mathsf{ws} = \mathsf{fr}$). Hence, a write-write pair to the same location is, by uniproc, in $(\mathsf{com} \cup (\mathsf{ws}; \mathsf{rf}) \cup (\mathsf{fr}; \mathsf{rf}))$. The cases $(\mathsf{ws}; \mathsf{rf})$ and $(\mathsf{fr}; \mathsf{rf})$ do not apply here because of the directions of the events. Hence a write-write pair to the same location is in com, i.e. in $\mathsf{ws} \cup \mathsf{rf} \cup \mathsf{fr}$. The cases rf and fr do not apply because of the directions of the events, hence such a pair is in ws. We know, by hypothesis of our framework, that ws is always global. Hence, there is no need to specify write-write pairs to the same location in the preserved program order, since we know that they are in ghb by the uniproc check.

The same reasoning applies for read-write pairs to the same location: such pairs are necessarily in fr, thus in ghb.

Hence, the uniproc check can be viewed as a minimal condition imposed by a machine: the write-write and read-write pairs to the same location in the program order are necessarily preserved globally in the order specified by the program.

5.2 Thin air

Second, we rule out programs where values come *out of thin air* [32]. This means that we forbid the *causal loops*, as illustrated in Fig. 11. In this example, the write (b) to y on P_0 depends 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

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`

Forbidden: `r1=1; r4=1;`

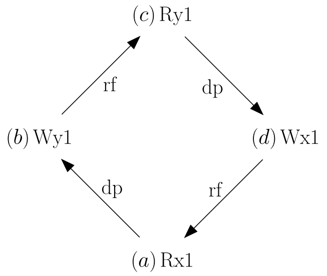


Fig. 11 Invalid execution according to the thin criterion

(b). Similarly on P_1, (d) depends on (c). 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. 11. In this case, the values read by (a) and (c) seem to come out of thin air, because they cannot be determined.

We model the dependencies between instructions with the dp relation. This relation is a subrelation of po, and always has a read at its source. Note that we suppose here that the definition of dp is independent of the architecture that we are studying. We express the absence of causal loop in a valid execution via the following check, directly inspired by Alpha's documentation [13, (I) 5–15, p. 245]:

Definition 13 (Thin air check)

$$\mathrm{thin}(E, X) \triangleq \mathrm{acyclic}\big(\mathsf{rf}(X) \cup \mathsf{dp}(E)\big)$$

6 Validity of an execution

We can now define what it means for an execution (E, X) to be valid on an architecture A of our framework. Then we state a notion of comparison between architectures, and two simple theorems relating two architectures.

6.1 Definition

We define the validity of an execution w.r.t. an architecture A as the conjunction of four checks. The first three, namely $\mathrm{wf}(X)$, $\mathrm{uniproc}(E, X)$ and $\mathrm{thin}(E, X)$ are independent of the architecture. The last one, i.e. the acyclicity of $\mathsf{ghb}_A(E, X)$, characterises the architecture. We write $\mathrm{valid}_A(E, X)$ when the execution (E, X) is valid on the architecture A:

Definition 14 (Validity)

$$\mathrm{valid}_A(E, X) \triangleq \mathrm{wf}(X) \wedge \mathrm{uniproc}(E, X) \wedge \mathrm{thin}(E, X) \wedge \mathrm{acyclic}\big(\mathsf{ghb}_A(E, X)\big)$$

For example, the execution of Fig. 2 is invalid on SC. Indeed the $\mathsf{ghb}_{\mathrm{SC}}(E, X)$ of this execution contains po and fr, therefore has a cycle: $(a) \overset{\mathsf{po}}{\to} (b) \overset{\mathsf{fr}}{\to} (c) \overset{\mathsf{po}}{\to} (d) \overset{\mathsf{fr}}{\to} (a)$. On the contrary, the $\mathsf{ghb}_{\mathrm{TSO}}(E, X)$ of this execution does not contain any po arrow whose source is a write and target a read, hence does not contain $(a, b) \in \mathsf{po}$ and $(c, d) \in \mathsf{po}$. Thus, there is no cycle in $\mathsf{ghb}_{\mathrm{TSO}}(E, X)$, which means that this execution is not forbidden on TSO.

6.2 Comparing architectures

From our definition of architecture arises a simple notion of comparison amongst them. $A_1 \leq A_2$ means that A_1 is *weaker* than A_2:

Definition 15 (Weaker)

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

As an example, TSO is weaker than SC. In the following two theorems, we suppose $A_1 \leq A_2$.

6.2.1 Validity is decreasing

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)

$$\forall EX. \operatorname{valid}_{A_2}(E, X) \quad \Rightarrow \quad \operatorname{valid}_{A_1}(E, X)$$

Proof From $A_1 \leq A_2$, we immediately have $\mathsf{ghb}_{A_1} \subseteq \mathsf{ghb}_{A_2}$, thus if ghb_{A_2} is acyclic, so is ghb_{A_1}. □

For example, since TSO is weaker than SC, all the executions valid on SC are valid on TSO.

6.2.2 Monotonicity of validity

The converse is not always true. However, some programs running on an architecture A_1 exhibit executions that would be valid on a stronger architecture A_2. To characterise all such executions, we first define $\operatorname{check}_{A_2}(E, X)$ as follows:

Definition 16 (Strong execution on weak architecture)

$$\operatorname{check}_{A_2}(E, X) \triangleq \operatorname{acyclic}(\mathsf{grf}_2 \cup \mathsf{ws} \cup \mathsf{fr} \cup \mathsf{ppo}_2)$$

We show that executions satisfying this criterion are valid on A_1 if and only if they are valid on A_2:

Theorem 2 (Characterisation)

$$\forall EX. \big(\operatorname{valid}_{A_1}(E, X) \wedge \operatorname{check}_{A_2}(E, X)\big) \quad \Leftrightarrow \quad \operatorname{valid}_{A_2}(E, X)$$

Proof

⇒ (E, X) being valid on A_1, we have all requirements—well-formedness, uniproc and thin—to guarantee that (E, X) is valid on A_2, except $\operatorname{valid}_{A_2}(E, X)$, which holds by the hypothesis $\operatorname{check}_{A_2}$.

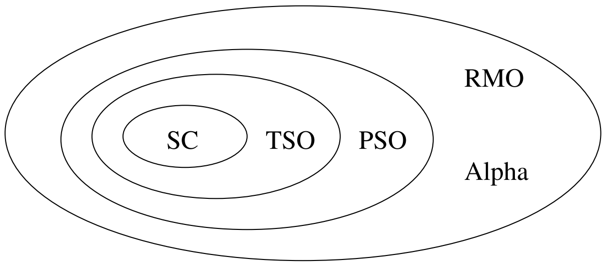


Fig. 12 Inclusion of some architectures

$\Leftarrow$ (E, X) being valid on A_2 gives us all requirements—well-formedness, uniproc and thin—to guarantee its validity on A_1 except the last one $\text{valid}_{A_1}(E, X)$. As $A_1 \leq A_2$, we know that $\mathsf{ghb}_{A_1} \subseteq \mathsf{ghb}_{A_2}$, thus the acyclicity requirement for ghb_{A_1} holds if ghb_{A_2} is acyclic. $\square$

For example, consider the execution of the test of Fig. 2 where P_0 executes its instructions before P_1 does: $(a) \xrightarrow{\mathsf{po}} (b) \xrightarrow{\mathsf{fr}} (c) \xrightarrow{\mathsf{po}} (d)$ and $(a, d) \in \mathsf{rf}$. It is valid on TSO since there is no cycle in $\mathsf{ghb}_{\text{TSO}}(E, X)$. It also satisfies $\text{check}_{\text{SC}}(E, X)$ since there is no cycle in $\mathsf{ghb}_{\text{SC}}(E, X)$. Hence it is valid on SC as well.

These theorems, though fairly simple, are useful to compare two models and to restore a strong model from a weaker one, as we do in [8, 10].

7 Classical models

We expose here how we implement several classical models in our framework, namely Sequential Consistency (SC) [30], the Sparc hierarchy (i.e. TSO, PSO and RMO [1]) and Alpha [13]. We prove our implementations equivalent to the original definitions. We present the models from the stronger (w.r.t. the order $\leq$), namely SC, to the weaker, namely RMO and Alpha. We show in Fig. 12 the inclusion of these models w.r.t. each other. The inclusion is here in terms of the behaviours that each model authorizes, therefore is in the converse order that the order $\leq$ induces, as expressed by Theorem 1.

7.1 Implementing an architecture

The native definitions of the models presented here roughly follow the same generic form. In these definitions, an execution ex is valid on an architecture A if it is an order on events which contains a certain subrelation r_A of the program order. Intuitively, the order ex corresponds to our global happens-before relation, and r_A to our preserved program order.

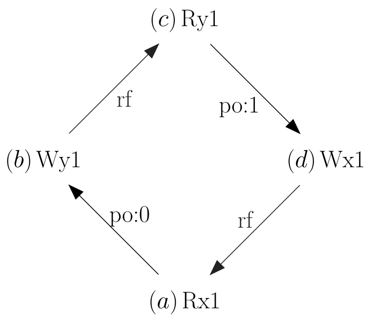
To show that the original specifications of the architectures that we study here correspond to their implementations in our framework, we need to express the definitions of our framework within the same shape as the original specifications. This means that from an execution witness (E, X) valid on A (as defined in our framework) we have to build an execution order ex which contains r_A (as given by the documentation), and conversely that from an execution order ex containing r_A (as given by the documentation), we need to extract an execution witness (E, X) valid on A (as defined in our framework).

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

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

Observed? `r1=1; r2=1;`

(a) A program



(b) A non-SC execution

Fig. 13 A program and a non-SC execution

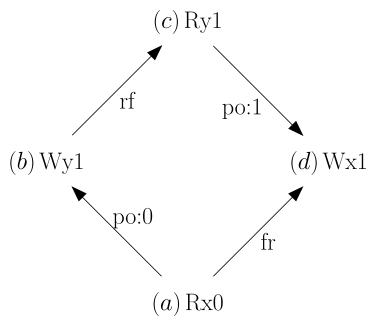


Fig. 14 A SC execution for the test of Fig. 13(a)

7.1.1 Building an execution witness from an order

Consider e.g. the event structure $(\{(a), (b), (c), (d)\}, \{(a, b) \in \mathsf{po}, (c, d) \in \mathsf{po}\})$ associated with the program of Fig. 13(a). On SC we have $(a, b) \in \mathsf{ppo}$ and $(c, d) \in \mathsf{ppo}$. Hence we can build a valid SC execution from the order $(a) \overset{\mathsf{ex}}{\to} (b) \overset{\mathsf{ex}}{\to} (c) \overset{\mathsf{ex}}{\to} (d)$, which is the one we give in Fig. 14. The first write in the order ex is (b), a write to y, which is immediately followed by the read (c) to y, hence we have $(b, c) \in \mathsf{rf}$. There is no write preceding the read (a) from x, hence (a) reads from the initial state. Moreover, this initial write to x precedes the write (d) in ws, hence $(a, d) \in \mathsf{fr}$.

We need to build an execution witness from a given order ex. In order to do so, we need to extract rf and ws from an order ex.

We write $\mathsf{ws}(\mathsf{ex})$ (resp. $\mathsf{rf}(\mathsf{ex})$) for the ws (resp. rf) extracted from ex. We have $(x, y) \in \mathsf{ws}(\mathsf{ex})$ when x and y are writes to the same location and $(x, y) \in \mathsf{ex}$. We have $(x, y) \in \mathsf{rf}(\mathsf{ex})$ when x is a write and y a read, both to the same location, such that x is a maximal previous write to this location before y in ex. Formally, writing $\mathrm{pw}(\mathsf{ex}, r)$ for the set of writes to the same location that precede the read event r in an order ex, we extract our rf and ws relations from ex as follows:

Definition 17 (Extraction of ws and fr from an order ex)

$$\mathsf{ws}(\mathsf{ex}) \triangleq \left(\bigcup_{\ell} \mathbb{WW}_{\ell}\right) \cap \mathsf{ex}$$

$$\mathsf{rf}(\mathsf{ex}) \triangleq \left\{(w, r) \in \mathsf{ex} \mid \mathrm{loc}(w) = \mathrm{loc}(r) \wedge \neg\left(\exists w'.\left(w, w'\right) \in \mathsf{ex} \wedge \left(w', r\right) \in \mathsf{ex} \cup \mathsf{po}\right)\right\}$$

We derive the from-read map as in Sect. 3.3.3:

Definition 18 (Extracted fr)

$$(r, w) \in \mathsf{fr}(\mathsf{ex}) \triangleq \exists w'.\left(w', r\right) \in \mathsf{rf}(\mathsf{ex}) \wedge \left(w', w\right) \in \mathsf{ws}(\mathsf{ex})$$

We show that the extracted read-from maps $\mathsf{rf}(\mathsf{ex})$, write serialisation $\mathsf{ws}(\mathsf{ex})$ and from-read maps $\mathsf{fr}(\mathsf{ex})$ are included in ex:

Lemma 1 (Inclusion of extracted communication in ex)

$$\forall \mathsf{ex}.\, \mathsf{rf}(\mathsf{ex}) \subseteq \mathsf{ex} \wedge \mathsf{ws}(\mathsf{ex}) \subseteq \mathsf{ex} \wedge \mathsf{fr}(\mathsf{ex}) \subseteq \mathsf{ex}$$

Proof

- The read-from maps and write serialisation extracted from ex are included in ex by definition.
- Inclusion of from-read maps: Consider two events x and y such that $(x, y) \in \mathsf{fr}(\mathsf{ex})$. We want to show that $(x, y) \in \mathsf{ex}$. Since ex is a linear order, we know that either $(x, y) \in \mathsf{ex}$, in which case we have the result, or $(y, x) \in \mathsf{ex}$. Suppose this last possibility, i.e. $(y, x) \in \mathsf{ex}$. We know that y is a write to x's location, since it is the target of a fr which source is x. Therefore, if $(y, x) \in \mathsf{ex}$, we know that y is a previous write to x in ex. Hence we have $y \in \mathrm{pw}(\mathsf{ex}, x)$. Moreover, since $(x, y) \in \mathsf{fr}(\mathsf{ex})$, we know by definition that there exists w_x such that $(w_x, x) \in \mathsf{rf}(\mathsf{ex})$ and $(w_x, y) \in \mathsf{ws}(\mathsf{ex})$. Since $\mathsf{ws}(\mathsf{ex})$ is included in ex, we have $(w_x, y) \in \mathsf{ex}$. But by definition of $\mathsf{rf}(\mathsf{ex})$, and since $(w_x, x) \in \mathsf{rf}(\mathsf{ex})$, w_x is the maximal previous write to x in ex. Since $(w_x, y) \in \mathsf{ex}$ and y is also a previous write, this contradicts the maximality of w_x. □

7.1.2 Lifting the constraints of an architecture to an order

Now that we know how to extract an execution witness from an order, we would like to express the constraints of an architecture A over this order, and prove the equivalence of the two notions of validity (the one over execution witnesses and the one over orders). Formally, writing $\mathrm{wit}(\mathsf{ex})$ for the execution witness built from ex, and $\mathrm{native}_A(E, \mathsf{ex})$ when ex is valid on A in the sense of the documentation (a notion formalised below), we would like to show that:

$$\forall E \mathsf{ex}.\, \mathrm{valid}_A\left(E, \mathrm{wit}(\mathsf{ex})\right) \quad \Leftrightarrow \quad \mathrm{native}_A(E, \mathsf{ex})$$

To do so, in the following, we interpret the relation r_A from the documentation as our preserved program order ppo_A, and the order ex as our global happens-before ghb_A. The order ex is defined as partial in Alpha's documentation [13], or early versions of the Sparc's [36]. In the current version of Sparc documentation [37], it is defined as a linear order. We suppose that we are in a setting where a partial order can be extended into a linear order, and define the native versions of the models in terms of a linear order.

Yet, all the criteria that we presented so far to decide the validity of an execution are in terms of execution witnesses. Thus, to ease the proofs, it helps to find an intermediate formulation of our framework that is closer to the native definitions as given by the documentations. Hence, for a given architecture A of our framework, we write $\text{linearised}_A(E, \mathsf{ex})$ when an order ex satisfies the conditions imposed by A. This merely corresponds to adapting the uniproc, thin and acyclicity of ghb_A checks to an order instead of an execution witness. Thus we use the same notations as for an execution witness, e.g. $\mathsf{com}(\mathsf{ex})$ for the communication relations extracted from the order ex. Formally, we have:

Definition 19 (Linearised definition of an architecture)

$$\begin{aligned}\text{linearised}_A(E, \mathsf{ex}) \triangleq\ & \text{acyclic}\big(\mathsf{com}(\mathsf{ex}) \cup \mathsf{po\text{-}loc}(E)\big)\\ & \wedge \text{acyclic}\big(\mathsf{rf}(\mathsf{ex}) \cup \mathsf{dp}(E)\big)\\ & \wedge \text{acyclic}\big(\mathsf{ws}(\mathsf{ex}) \cup \mathsf{fr}(\mathsf{ex}) \cup \mathsf{grf}_A(\mathsf{ex}) \cup \mathsf{ppo}_{\mathsf{A}}(E)\big)\end{aligned}$$

We want to show that the validity of an execution on A corresponds to the definition above. This means that whenever an execution ex is valid on A according to the definition above, we can build an execution witness (E, X) which is valid on A, such that ex and (E, X) have the same events and the same communication relations. Conversely, from an execution (E, X) valid on A, we are able to build an execution ex valid on A with the same events and communication relations.

7.1.3 From the linearised definition to ours

Let us consider the first part of this equivalence. Consider an order ex satisfying the linearised definition. The extracted rf and ws are well formed in a finite execution, hence an execution witness built from these relations is well formed. Let us now show that the execution witness built out of ex is valid on a given architecture A. This means that we have to show that the extracted execution witness respects the uniproc, thin and acyclicity of ghb_A checks.

Let us first show that an extracted execution witness respects uniproc:

Lemma 2 (Extracted execution witness respects uniproc)

$$\forall E \mathsf{ex}.\text{acyclic}\big(\mathsf{com}(\mathsf{ex}) \cup \mathsf{po\text{-}loc}(E)\big) \quad\Rightarrow\quad \text{uniproc}\big(E, \text{wit}(\mathsf{ex})\big)$$

Proof Let us write X for wit(ex). The uniproc check implies that $\forall xy, (x, y) \in \text{pio}(E) \Rightarrow \neg((y, x) \in (\mathsf{com})^+(X))$. Let us suppose as a contradiction two events x and y such that $(x, y) \in \mathsf{po\text{-}loc}(E)$ and $(y, x) \in (\mathsf{com})^+(X)$. We know that $(\mathsf{com})^+ = \mathsf{rf} \cup \mathsf{ws} \cup \mathsf{fr} \cup (\mathsf{ws};\mathsf{rf}) \cup (\mathsf{fr};\mathsf{rf})$. Let us do a case disjunction on $(y, x) \in (\mathsf{com})^+$:

- if $(y, x) \in \mathsf{rf}(X)$, we have $(y, x) \in \mathsf{rf}(\mathsf{ex})$ by hypothesis. Therefore, since $(x, y) \in \mathsf{po\text{-}loc}$, we have a cycle in $\mathsf{com}(\mathsf{ex}) \cup \mathsf{po\text{-}loc}$, a contradiction.
- if $(y, x) \in \mathsf{ws}(X)$, we have $(y, x) \in \mathsf{ws}(\mathsf{ex})$ by hypothesis. Since $(x, y) \in \mathsf{po\text{-}loc}$, we have a cycle in $\mathsf{com}(\mathsf{ex}) \cup \mathsf{po\text{-}loc}$, a contradiction.
- if $(y, x) \in \mathsf{fr}(E, X)$, we have $(y, x) \in \mathsf{fr}(\mathsf{ex})$ by definition. Since $(x, y) \in \mathsf{po\text{-}loc}$, we have a cycle in $\mathsf{com}(\mathsf{ex}) \cup \mathsf{po\text{-}loc}$, a contradiction.

- if $(y, x) \in \mathsf{ws}(X); \mathsf{rf}(X)$, there exists w_x such that $(y, w_x) \in \mathsf{ws}(X)$ and $(w_x, x) \in \mathsf{rf}(X)$. Since $(w_x, x) \in \mathsf{rf}(X)$, we have $(w_x, x) \in \mathsf{rf}(\mathsf{ex})$ by hypothesis. Similarly, we have $(y, w_x) \in \mathsf{ws}(\mathsf{ex})$. Hence we have a cycle in $\mathsf{com}(\mathsf{ex}) \cup \mathsf{po\text{-}loc}$.
- if $(y, x) \in \mathsf{fr}(E, X); \mathsf{rf}(X)$, there exists w_x such that $(y, w_x) \in \mathsf{fr}(E, X)$ and $(w_x, x) \in \mathsf{rf}(X)$. Since $(w_x, x) \in \mathsf{rf}(X)$, we have $(w_x, x) \in \mathsf{rf}(\mathsf{ex})$ by hypothesis. By definition we have $(y, w_x) \in \mathsf{fr}(\mathsf{ex})$, hence a cycle in $\mathsf{com}(\mathsf{ex}) \cup \mathsf{po\text{-}loc}$. □

Let us now show that the extracted execution witness respects the thin check:

Lemma 3 (Extracted execution witness respects thin)

$$\forall E\,\mathsf{ex}.\mathrm{acyclic}\big(\mathsf{rf}(\mathsf{ex}) \cup \mathsf{dp}(E)\big) \quad \Rightarrow \quad \mathrm{thin}\big(E, \mathrm{wit}(\mathsf{ex})\big)$$

Proof We have $\mathsf{rf}(\mathrm{wit}(\mathsf{ex})) = \mathsf{rf}(\mathsf{ex})$, hence the result. □

Using these two lemmas, we show the validity of the extracted witness:

Lemma 4 (Validity of extracted execution witness)

$$\forall E\,\mathsf{ex}.\mathrm{linearised}_A(E, \mathsf{ex}) \quad \Rightarrow \quad \mathrm{valid}_A\big(E, \mathrm{wit}(\mathsf{ex})\big)$$

Proof Let us write $X = \mathrm{wit}(\mathsf{ex})$. By Sect. 6.1, X is valid on A if X is well formed, respects the uniproc and thin checks, and $\mathsf{ghb}_A(X)$ is acyclic.

- Well-formedness: $\mathsf{rf}(\mathsf{ex})$ and $\mathsf{ws}(\mathsf{ex})$ are trivially well formed, hence $(\mathsf{rf}(\mathsf{ex}), \mathsf{ws}(\mathsf{ex}))$ is well formed as well.
- Uniproc: we want to show that X respects the uniproc check. Since we know by hypothesis that acyclic$(\mathsf{com}(\mathsf{ex}) \cup \mathsf{po\text{-}loc})$, Lemma 2 applies directly.
- Thin: we want to show that X respects the thin check. Since we know by hypothesis that acyclic$(\mathsf{rf}(\mathsf{ex}) \cup \mathsf{dp})$, Lemma 3 applies directly.
- Acyclicity of ghb: we want to show that $\mathsf{ghb}_A(X)$ is acyclic. Since $X = \mathrm{wit}(\mathsf{ex})$, we know that $\mathsf{ws}(\mathsf{ex}) = \mathsf{ws}(X)$ and $\mathsf{rf}(\mathsf{ex}) = \mathsf{rf}(X)$. By definition, we have $\mathsf{fr}(\mathsf{ex}) = \mathsf{fr}(E, X)$. Thus, $\mathsf{ghb}(X) = (\mathsf{grf}(\mathsf{ex}) \cup \mathsf{ws}(\mathsf{ex}) \cup \mathsf{fr}(\mathsf{ex}) \cup \mathsf{ppo}_\mathsf{A})$, which is acyclic by hypothesis. This entails the acyclicity of ghb_A. □

This is enough to show that the linearised notion of validity entails our definition of validity, for a given architecture A.

7.1.4 From our implementation to the linearised one

Conversely, to show that one of our execution witnesses corresponds to a linearised execution, we need to build an order from an execution witness. This order will typically be the ghb of our execution witness, or more precisely a linear extension of it, so as to build a linear order. Formally, we want to show:

Lemma 5 From A to its linearised definition

$$\forall E\,\mathsf{ex}.\ \mathrm{valid}_A\big(E, \mathrm{wit}(\mathsf{ex})\big) \quad \Rightarrow \quad \mathrm{linearised}_A(E, \mathsf{ex})$$

Proof From $(E, \mathrm{wit}(\mathsf{ex}))$ being valid on A, we know that:

Name	Arch	Section
SC	$(\mathrm{MM}, \mathsf{rf})$	7.2.1
TSO	$(\lambda(E, X).(\mathrm{RM}(E, X) \cup \mathrm{WW}(E, X)), \mathsf{rfe})$	7.2.2.2
PSO	$(\lambda(E, X) \cdot \mathrm{RM}(E, X), \mathsf{rfe})$	7.2.2.3
RMO	$(\lambda(E, X) \cdot \mathsf{dp}(E, X), \mathsf{rfe})$	7.2.2.4
Alpha	$(\lambda(E, X) \cdot (\bigcup_\ell \mathrm{RR}_\ell(E, X)), \mathsf{rfe})$	7.2.3

Fig. 15 Summary of models

– $(E, \mathrm{wit}(\mathsf{ex}))$ passes the uniproc check, thus $\mathrm{acyclic}(\mathsf{com}(\mathsf{ex}) \cup \mathsf{po\text{-}loc}(E))$;
– $(E, \mathrm{wit}(\mathsf{ex}))$ passes the thin check, from which we have $\mathrm{acyclic}(\mathsf{rf}(\mathsf{ex}) \cup \mathsf{dp}(E))$;
– $\mathsf{ghb}_A(E, \mathrm{wit}(\mathsf{ex}))$ is acyclic, from which we have the last requirement. □

This is enough to conclude that our notion of validity entails the linearised one. Thus, the two notions of validity are equivalent. In the following, we will use the linearised notion to relate more easily to the architectures' definitions given by documentations.

7.2 A hierarchy of classical models

We now describe how we implement several classical models, namely SC, the Sparc hierarchy, and Alpha. We give in Fig. 15 a table summarising the implementation of these models in our framework. Note that all of these models consider the stores to be atomic. The reader will find an instance of our framework that relaxes store atomicity in [11], where we present a model of the Power architecture.

We define notations to extract pairs of memory events from the program order in Appendix. For example, WW represents the function which extracts the write-write pairs in the program order of an execution. We write $\mathbb{WW}_\ell$ when the writes have the same location ℓ.

7.2.1 Sequential Consistency (SC)

SC has been defined in [30] as follows:

[...] the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

SC allows no reordering of events (ppo equals po on memory events) and makes writes available to all processors as soon as they are issued (rf is global). Note that any architecture definable in our framework is weaker than SC:

Definition 20 (Sequential Consistency)

$$\mathrm{SC} \triangleq (\mathsf{po}, \mathsf{rf})$$

The following criterion characterises, as shown in Sect. 6.2, valid SC executions on any architecture:

Definition 21 (SC check)

$$\text{check}_{\text{SC}}(E, X) = \text{acyclic}(\mathsf{com} \cup \mathsf{po})$$

In [30], a SC execution is an order ex which includes the program order:

$$\text{native}_{\text{SC}}(E, \mathsf{ex}) \triangleq \mathsf{po} \subseteq \mathsf{ex}$$

The implicit execution model of [30] states that a read r takes its value from the most recent write that precedes it in ex. Hence we extract rf and ws from ex following Sect. 7.1, and build one of our execution witnesses from ex.

Finally, we show, following the proof given in Sect. 7.1.3, that each execution witness built as above corresponds to a valid execution in our SC model. Indeed, since ex contains all of po, it contains in particular the read-write and write-write pairs to the same location, which is enough to ensure uniproc. Similarly, since it contains the dp relation, it ensures the thin check:

Theorem 3 (SC is SC)

$$\forall E \mathsf{ex}.\, \text{valid}_{\text{SC}}(E, \mathsf{ex}) \quad \Leftrightarrow \quad \text{native}_{\text{SC}}(E, \mathsf{ex})$$

7.2.2 The Sparc hierarchy

We present here the definitions of Sun's TSO, PSO and RMO.

7.2.2.1 The Value *axiom* The execution model of the Sparc architectures is provided by the *Value* axiom of [1, V8; App. K; p. 283], which states that a read (L_a for Sparc) reads from the most recent write (S_a) before L_a in the global ordering relation (which they note $\leq$) or in the program order (which they note ;):

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

The fact that the store from which a load reads is specified to come *either* from the global ordering relation *or* the program order means that the program order is not included in the global ordering. This means that an rf relation occurs in the global order if and only if it is an rf between two events from distinct processors. Therefore, we deduce that for each of the Sparc architecture, the external rf are global, and the internal rf are not.

7.2.2.2 Total Store Order (TSO) TSO allows two relaxations [5]: *write to read program order*, and *read own write early*. The *write to read program order* relaxation means that TSO's preserved program order includes all pairs but the store-load ones. The *read own write early* relaxation means TSO's internal read-from maps are not global, which is also expressed by the *Value* axiom.

Definition 22 (TSO)

$$\mathsf{ppo}_{\text{TSO}} \triangleq \big(\lambda(E, X).\big(\text{RM}(E, X) \cup \text{WW}(E, X)\big)\big)$$

$$\text{TSO} \triangleq (\mathsf{ppo}_{\text{TSO}}, \mathsf{rfe})$$

Section 6.2 shows that the following criterion characterises valid executions (w.r.t. any $A \leq \text{TSO}$) that would be valid on TSO, e.g., in Fig. 2:

Definition 23 (TSO check)

$$\text{check}_{\text{TSO}}(E, X) = \text{acyclic}(\mathsf{ws} \cup \mathsf{fr} \cup \mathsf{rfe} \cup \mathsf{ppo}_{\text{TSO}})$$

Sparc [1, V. 8, Appendix K] defines a TSO execution as an order ex on memory events constrained by some axioms. We formulate those axioms as follows:[3]

$$\text{native}_{\text{TSO}}(E, \mathsf{ex}) \triangleq (\mathbb{RM} \cup \mathbb{WW}) \subseteq \mathsf{ex}$$

Finally, we show, following the proof given in Sect. 7.1.3, that an order ex satisfying the axioms of Sun's TSO's specification corresponds to a valid execution in our TSO model. Indeed, since ex contains the read-write and write-write pairs to the same location, this is enough to ensure uniproc. Similarly, since it contains the dp relation, it ensures the thin check:

Theorem 4 (TSO is TSO)

$$\forall E \mathsf{ex}.\ \text{valid}_{\text{TSO}}(E, \mathsf{ex}) \quad \Leftrightarrow \quad \text{native}_{\text{TSO}}(E, \mathsf{ex})$$

7.2.2.3 Partial Store Ordering (PSO) PSO maintains only the write-write pairs to the same location and all read-read and read-write pairs [1]. However, there is no need to specify the write-write pairs to the same location in PSO's preserved program order. Indeed, according to Sect. 5.1.1, we know that two writes in program order to the same location are in ws. We know, by hypothesis of our framework, that ws is always global. Hence, there is no need to specify write-write pairs to the same location in PSO's preserved program order, since we know that they are preserved globally (i.e. in ghb) by the uniproc check. As the *Value* axiom holds for PSO as well, PSO's external rf are global whereas its internal rf are not:

Definition 24 (PSO)

$$\begin{aligned} &\mathsf{ppo}_{\text{PSO}} \triangleq \lambda(E, X).\text{RM}(E, X) \\ &\text{PSO} \triangleq (\mathsf{ppo}_{\text{PSO}}, \mathsf{rfe}) \end{aligned}$$

Section 6.2 shows that the following criterion characterises valid executions (w.r.t. any $A \leq \text{PSO}$) that would be valid on PSO, e.g., in Fig. 2:

Definition 25 (PSO check)

$$\text{check}_{\text{PSO}}(E, X) = \text{acyclic}(\mathsf{ws} \cup \mathsf{fr} \cup \mathsf{rfe} \cup \mathsf{ppo}_{\text{PSO}})$$

Sparc [1, V. 8, Appendix K] defines a PSO execution as an order ex on memory events constrained by some axioms. We formulate those as follows:

$$\text{native}_{\text{PSO}}(E, \mathsf{ex}) \triangleq \left(\mathbb{RM} \cup \bigcup_{\ell} \mathbb{WW}_{\ell}\right) \subseteq \mathsf{ex}$$

[3] We omit the axioms *Atomicity* and *Termination*.

We show, following the proof given in Sect. 7.1.3, that an order ex satisfying Sun PSO's specification corresponds to a valid execution of our PSO model. Indeed, since ex contains the read-write and write-write pairs to the same location, this is enough to ensure uniproc. Similarly, since it contains the dp relation, it ensures the thin check:

Theorem 5 (PSO is PSO)

$$\forall E \mathsf{ex}.\ \mathrm{valid}_{\mathrm{PSO}}\big(E, \mathrm{wit}(\mathsf{ex})\big) \quad \Leftrightarrow \quad \mathrm{native}_{\mathrm{PSO}}(E, \mathsf{ex})$$

7.2.2.4 Relaxed Memory Order (RMO) RMO preserves the program order between the write-write and read-write pairs to the same location, and the read-read and read-write pairs in the dependency relation [1]. However, there is no need to specify the write-write and read-write pairs to the same location in RMO's preserved program order, as exposed in Sect. 5.1.1. Indeed, the write-write pairs to the same location are in ws, hence in ghb. Moreover, by the same reasoning, the read-write pairs to the same location are in fr, hence in ghb.

The *Value* axiom holds for RMO as well. Hence we have (writing $\mathsf{dp}(E, X)$ for the pairs in dependency in an execution (E, X)):

Definition 26 (RMO)

$$\mathsf{ppo}_{\mathrm{RMO}} \triangleq \lambda(E, X).\mathsf{dp}(E, X)$$
$$\mathrm{RMO} \triangleq (\mathsf{ppo}_{\mathrm{RMO}}, \mathsf{rfe})$$

Section 6.2 shows that the following criterion characterises valid executions (w.r.t. any $A \le \mathrm{RMO}$) that would be valid on RMO, e.g., in Fig. 2:

Definition 27 (RMO check)

$$\mathrm{check}_{\mathrm{RMO}}(E, X) = \mathrm{acyclic}(\mathsf{ws} \cup \mathsf{fr} \cup \mathsf{rfe} \cup \mathsf{ppo}_{\mathrm{RMO}})$$

Sparc [1, V. 8, Appendix K] defines a RMO execution as an order ex on memory events constrained by some axioms:

$$\mathrm{native}_{\mathrm{RMO}}(E, \mathsf{ex}) \triangleq \left(\mathsf{dp} \cup \bigcup_{\ell} \mathbb{MW}_{\ell}\right) \subseteq \mathsf{ex}$$

We show, following the proof given in Sect. 7.1.3, that an order ex satisfying Sun RMO's specification corresponds to a valid execution of our RMO model. Indeed, since ex contains the read-write and write-write pairs to the same location, this is enough to ensure uniproc. Similarly, since it contains the dp relation, it ensures the thin check:

Theorem 6 (RMO is RMO)

$$\forall E \mathsf{ex}.\ \mathrm{valid}_{\mathrm{RMO}}(E, \mathsf{ex}) \quad \Leftrightarrow \quad \mathrm{native}_{\mathrm{RMO}}(E, \mathsf{ex})$$

7.2.3 Alpha

Alpha maintains the write-write, read-read and read-write pairs to the same location [13]. We exposed in Sect. 5.1.1 why there is no need to include the write-write pairs to the same location in ppo (they are in ws thus in ghb by uniproc), and why the read-write pairs to the same location are exempted as well (they are in fr thus in ghb by uniproc). However, we do need to specify the read-read pairs to the same location in Alpha's preserved program order, because such a relation may not be global, if not specified in the ppo. Moreover, Alpha specifies, for every read, the write from which it reads as the last one either:

- in *processor issue sequence*, i.e. our program order, or
- in the *BEFORE* order, which corresponds to our global happens-before order.

Thus, similarly to the Sun models, external rf are global whereas internal are not:

Definition 28 (Alpha)

$$\mathsf{ppo}_{\text{Alpha}} \triangleq \lambda(E, X).\left(\bigcup_{\ell} \text{RR}_{\ell}(E, X)\right)$$

$$\text{Alpha} \triangleq (\mathsf{ppo}_{\text{Alpha}}, \mathsf{rfe})$$

Section 6.2 shows the following criterion characterises valid executions (w.r.t. any $A \leq$ Alpha) that would be valid on Alpha, e.g., in Fig. 2:

Definition 29 (Alpha check)

$$\text{check}_{\text{Alpha}}(E, X) = \text{acyclic}(\mathsf{ws} \cup \mathsf{fr} \cup \mathsf{rfe} \cup \mathsf{ppo}_{\text{Alpha}})$$

Alpha [13] formally defines an Alpha execution as an order ex on memory events constrained by some axioms. We formulate those axioms as follows:

$$\text{native}_{\text{Alpha}}(E, \mathsf{ex}) \triangleq \left(\bigcup_{\ell} \mathbb{RM}_{\ell} \cup \mathbb{WW}_{\ell}\right) \subseteq \mathsf{ex} \wedge \text{acyclic}\big(\mathsf{rf}(\mathsf{ex}) \cup \mathsf{dp}(E)\big)$$

Finally, we show, following the proof given in Sect. 7.1.3, that any order ex satisfying Alpha's axioms corresponds to a valid execution in our Alpha model. Indeed, since ex contains the read-write and write-write pairs to the same location, this is enough to ensure uniproc. Moreover, the thin check is literally given:

Theorem 7 (Alpha is Alpha)

$$\forall E \mathsf{ex}.\ \text{valid}_{\text{Alpha}}(E, \mathsf{ex}) \quad \Leftrightarrow \quad \text{native}_{\text{Alpha}}(E, \mathsf{ex})$$

7.2.4 RMO and Alpha are incomparable

We saw in Sect. 5.1.2 that RMO authorizes load-load hazard, where two reads on the same processor from the same location can be reordered. We illustrated this by explaining why the test of Fig. 10(a) can exhibit its outcome on a RMO machine.

However, Alpha preserves read-read pairs to the same location in program order, as exposed in Sect. 7.2.3. Therefore the read-read pair $(b, c) \in \mathsf{po}$ to the same location on P_0 is included in Alpha's preserved program order. Moreover, we know that the external read-from maps are global on Alpha, hence the relation $(d, b) \in \mathsf{rf}$ is global as well. Hence the execution (E, X) depicted in Fig. 10(b) exhibits a cycle in $\mathsf{ghb}_{\text{Alpha}}(E, X)$: $(b) \xrightarrow{\mathsf{ppo}} (c) \xrightarrow{\mathsf{fr}} (d) \xrightarrow{\mathsf{rfe}} (b)$, which forbids this execution.

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`
(b) `r3` $\leftarrow$ `y`	(d) `r4` $\leftarrow$ `x`		

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

Fig. 16 The **iriw** example

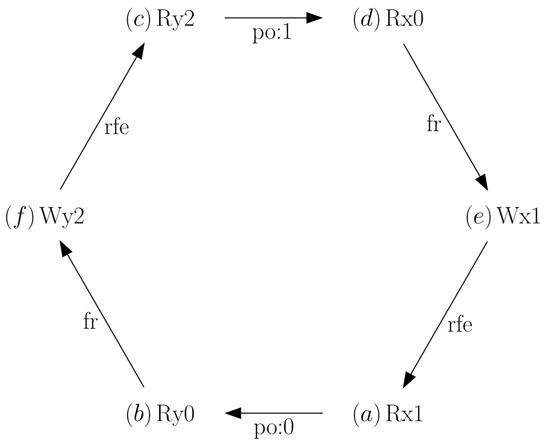


Fig. 17 A non-SC execution of **iriw**

Hence, RMO authorizes load-load hazard whereas Alpha does not.

Consider now the **iriw** (for Independent Reads of Independent Writes) example given in Fig. 16, and suppose that there is a dependency between the pairs of reads on P_0 and P_1, i.e. $(a, b) \in \mathsf{dp}$ and $(c, d) \in \mathsf{dp}$. We can enforce these pairs to be in dependency by adding for example a logical operation between them, such as a `xor` operating on the registers of the load instructions associated to the read events.

The specified outcome may be revealed by an execution such as the one we depict in Fig. 17. Suppose that each location and register initially hold 0. If `r1` holds 1 on P_0 in the end, the read (a) has read its value from the write (e) on P_2, hence $(e, a) \in \mathsf{rf}$. On the contrary, if `r2` holds 0 in the end, the read (b) has read its value from the initial state, thus before the write (f) on P_3, hence $(b, f) \in \mathsf{fr}$. Similarly, we have $(f, c) \in \mathsf{rf}$ from `r2` holding 1 on P_1, and $(d, e) \in \mathsf{fr}$ from `r1` holding 0 on P_1. Hence, if the specified outcome is observed, it ensures that at least one execution of the test contains the cycle depicted in Fig. 17: $(a) \xrightarrow{\mathsf{dp}} (b) \xrightarrow{\mathsf{fr}} (f) \xrightarrow{\mathsf{rfe}} (c) \xrightarrow{\mathsf{dp}} (d) \xrightarrow{\mathsf{fr}} (e) \xrightarrow{\mathsf{rf}} (a)$.

This cycle is not global on Alpha whereas it is on RMO. This means that the associated execution is authorised on Alpha whereas it is forbidden on RMO. Indeed on RMO, the pairs in dependency are included in the preserved program order, hence the pairs $(a, b) \in \mathsf{dp}$ and $(c, d) \in \mathsf{dp}$ are included in ppo, hence in ghb. Moreover, the external read-from maps are global on RMO (see Sect. 7.2.2.4), and fr is always global. However on Alpha, these pairs are not preserved globally, therefore this execution is authorised.

Hence, Alpha authorizes **iriw** with dependencies whereas RMO does not.

8 Conclusion

We presented here a formal generic framework for defining weak memory models. We minimised the number of our axioms to make any specification of a model in our framework concise. We demonstrated the strength and semantical scope of our axioms by equivalences proofs of several existing weak memory models and their native definition, amongst which Sequential Consistency and Sun's TSO, which is also known to be the memory model exhibited by x86 processors [33].

We hope that we have highlighted by these proofs, in a precise and formal way, the ingredients (i.e. our axioms) necessary to the definition and study of weak memory models in the same generic terms.

The characterisation we propose in Theorem 2 is simple, since it is merely an acyclicity check of the global happens-before relation. This check is already known for SC [31], and recent verification tools use it for architectures with store buffer relaxation [18, 25]. Our work extends the scope of these methodologies to models relaxing store atomicity.

Finally, even though we do not expose these results in the present paper, we have demonstrated the generality and applicability of our framework in an experimental way. As presented in [11], we have indeed tested several Power machines, which allowed us to design a model of the Power architecture, which is an instance of the present framework. Given the complexity and subtlety of the Power architecture, this demonstrates (in conjunction with the formal proofs presented here) the generality, the expressivity, as well as the preciseness of our framework.

In addition, we also prove formally (in [8, 10]) results about where to place synchronisation primitives in a piece of code, and how to optimise this placement. Our formal proofs demonstrate the useability of our framework, for example as a basis for defining programming disciplines for concurrent programs.

Acknowledgements We thank the anonymous reviewers of several versions of this paper for their helpful and insightful reviews. We thank Gérard Boudol, Damien Doligez, Matthew Hague, Maurice Herlihy, Xavier Leroy, Luc Maranget, Susmit Sarkar and Peter Sewell for invaluable discussions and comments, Assia Mahboubi and Vincent Siles for advice on the Coq development, and Thomas Braibant, Matt Lewis, Jules Villard and Boris Yakobowski for comments on a draft.

Appendix: Tables of notations

Table 1 Table of notations for events and pairs of events

Name	Notation	Comment
Memory events	$\mathbb{M}$	All memory events
Memory events to the same location	$\mathbb{M}_\ell$	Memory events relative w/location ℓ
Read events, reads	$\mathbb{R}$	Memory events that are reads
Reads from the same location	$\mathbb{R}_\ell$	Reads from the location ℓ
Write events, writes	$\mathbb{W}$	Memory events that are writes
Writes to the same location	$\mathbb{W}_\ell$	Writes to the location ℓ
Memory pairs	$\mathbb{MM}$	Pairs of any memory events in program order
Memory pairs to the same location	$\mathbb{MM}_\ell$	Pairs of any memory events to the same location in program order

Table 1 (*Continued*)

Name	Notation	Comment
Read-read pairs	$\mathbb{RR}$	Pairs of reads in program order
Read-read pairs to the same location	$\mathbb{RR}_\ell$	Pairs of reads from the same location in program order
Read-write pairs	$\mathbb{RW}$	Read followed by write in program order
Read-write pairs to the same location	$\mathbb{RW}_\ell$	Read followed by write to the same location in program order
Write-write pairs	$\mathbb{WW}$	Pairs of writes in program order
Write-write pairs to the same location	$\mathbb{WW}_\ell$	Pairs of writes to the same location in program order
Write-read pairs	$\mathbb{WR}$	Write followed by read in program order
Write-read pairs to the same location	$\mathbb{WR}_\ell$	Write followed by read from the same location in program order

Table 2 Table of relations

Name	Notation	Comment	Section
Program order	$(m_1, m_2) \in \mathsf{po}$	Per-processor linear order	3.2
Dependencies	$(m_1, m_2) \in \mathsf{dp}$	Included in po, source is a read	5.2
Po-loc	$(m_1, m_2) \in \mathsf{po\text{-}loc}$	po restricted to the same location	5.1.1
Preserved program order	$(m_1, m_2) \in \mathsf{ppo}$	Pairs maintained in program order ($\subset$ po)	4.1
Read-from map	$(w, r) \in \mathsf{rf}$	Links a write to a read reading its value	3.3.1
External read-from map	$(w, r) \in \mathsf{rfe}$	rf between events from distinct processors	4
Internal read-from map	$(w, r) \in \mathsf{rfi}$	rf between events from the same processor	4
Global read-from map	$(w, r) \in \mathsf{grf}$	rf considered global	4
Write serialisation	$(w_1, w_2) \in \mathsf{ws}$	Linear order on writes to the same location	3.3.2
From-read map	$r \,\mathsf{fr}\, w$	r reads from a write preceding w in ws	3.3.3
Global happens-before	$(m_1, m_2) \in \mathsf{ghb}$	Union of global relations	4.3.1
Communication	$(m_1, m_2) \in \mathsf{com}$	Shorthand for $(m_1, m_2) \in (\mathsf{rf} \cup \mathsf{ws} \cup \mathsf{fr})$	3.3.4

Table 3 Notations to extract pairs from po

Function	Comment
$\mathrm{MM} \triangleq \lambda(E, X).((\mathbb{MM}) \cap \mathrm{po}(X))$	Two memory events in program order
$\mathrm{RM} \triangleq \lambda(E, X).((\mathbb{RM}) \cap \mathrm{po}(X))$	A read followed by a memory event in program order
$\mathrm{WW} \triangleq \lambda(E, X).((\mathbb{WW}) \cap \mathrm{po}(X))$	Two writes in program order
$\mathrm{MW} \triangleq \lambda(E, X).((\mathbb{MW}) \cap \mathrm{po}(X))$	A memory event followed by a write in program order

References

1. Sparc Architecture Manual (1992 and 1994) Versions 8 and 9
2. Power ISA (2009) Version 2.06
3. Adir A, Attiya H, Shurek G (2003) Information-flow models for shared memory with an application to the powerPC architecture. In: TPDS

4. Adve SV (1993) Designing memory consistency models for shared-memory multiprocessors. PhD thesis, 1993
5. Adve SV, Gharachorloo K (1995) Shared memory consistency models: a tutorial. IEEE Comput 29:66–76
6. Adve SV, Boehm H-J (2012) Memory models: a case for rethinking parallel languages and hardware. Commun ACM. doi:10.1145/1787234.1787255
7. Ahamad M, Bazzi RA, John R, Kohli P, Neiger G (1993) The power of processor consistency. In: SPAA
8. Alglave J (2010) A shared memory poetics. PhD thesis, Université Paris 7 and INRIA. http://moscova.inria.fr/~alglave/these
9. Alglave J, Kroening D, Lugton J, Nimal V, Tautschnig M (2011) Soundness of data flow analyses on weak memory models In: APLAS 11
10. Alglave J, Maranget L (2011) Stability in weak memory models. In: CAV
11. Alglave J, Maranget L, Sarkar S, Sewell P (2010) Fences in weak memory models. In: CAV
12. Alglave J, Maranget L, Sarkar S, Sewell P (2011) Litmus: running tests against hardware. In: TACAS
13. Alpha Architecture Reference Manual, 4th edn (2002)
14. Arvind, Maessen J-W (2006) Memory model = instruction reordering + store atomicity. In: ISCA
15. Bertot Y, Casteran P (2004) Coq'Art, EATCS texts in theoretical computer science. Springer, Berlin
16. Boehm H-J, Adve SV (2008) Foundations of the C++ concurrency memory model. In: PLDI
17. Boudol G, Petri G (2009) Relaxed memory models: an operational approach. In: POPL
18. Burckhardt S, Musuvathi M (2008) Effective program verification for relaxed memory models. In: CAV
19. Burckhardt S, Musuvathi M, Singh V (2010) Verifying local transformations of concurrent programs. In: CC
20. Cantin J, Lipasti M, Smith J (2003) The complexity of verifying memory coherence. In: SPAA
21. Collier WW (1992) Reasoning about parallel architectures. Prentice Hall, New York
22. Dubois M, Scheurich C (1990) Memory access dependencies in shared-memory multiprocessors. IEEE Trans Softw Eng 16(6). doi:10.1109/32.55094
23. Ferreira R, Feng X, Shao Z (2010) Parameterized memory models and concurrent separation logic. In: ESOP
24. Gharachorloo K (1995) Memory consistency models for shared-memory multiprocessors. WRL Res Rep 95(9). doi:10.1.1.37.3026
25. Hangal S, Vahia D, Manovit C, Lu J-YJ, Narayanan S (2004) TSOTool: a program for verifying memory systems using the memory consistency model. In: ISCA
26. Higham L, Kawash J, Verwaal N (1998) Weak memory consistency models part I: definitions and comparisons. Technical report 98/612/03, Department of Computer Science, The University of Calgary
27. Intel 64 Architecture Memory Ordering White Paper, August 2007
28. Intel 64 and IA-32 Architectures Software Developer's Manual, vol 3A, October 2011
29. A Formal Specification of Intel Itanium Processor Family Memory Ordering, October 2002. Intel Document 251429-001
30. Lamport L (1979) How to make a correct multiprocess program execute correctly on a multiprocessor. IEEE Trans Comput 46(7):779–782
31. Landin A, Hagersten E, Haridi S (1991) Race-free interconnection networks and multiprocessor consistency. Comput Archit News 19(3):106–115
32. Manson J, Pugh W, Adve SV (2005) The Java memory model. In: POPL
33. Owens S, Sarkar S, Sewell P (2009) A better x86 memory model: x86-TSO. In: TPHOL
34. Sarkar S, Sewell P, Zappa Nardelli F, Owens S, Ridge T, Braibant T, Myreen M, Alglave J (2009) The semantics of x86-CC multiprocessor machine code. In: POPL
35. Sarkar S, Sewell P, Alglave J, Maranget L, Williams D (2011) Understanding power multiprocessors. In: PLDI 11
36. Sparc Architecture Manual Version 8 (1992)
37. Sparc Architecture Manual Version 9 (1994)
38. Yang Y, Gopalakrishnan G, Lindstrom G (2007) UMM: an operational memory model specification framework with integrated model checking capability. In: CCPE
39. Yang Y, Gopalakrishnan G, Linstrom G, Slind K (2004) Nemos: a framework for axiomatic and executable specifications of memory consistency models. In: IPDPS
40. Zappa Nardelli F, Sewell P, Sevcik J, Sarkar S, Owens S, Maranget L, Batty M, Alglave J (2009) Relaxed memory models must be rigorous. In: EC^2 09