

# Relaxed Operational Semantics of Concurrent Programming Languages

G rard Boudol

INRIA – Sophia Antipolis, France

Gustavo Petri

Purdue University, USA

Bernard Serpette

INRIA – Sophia Antipolis, France

We propose a novel, operational framework to formally describe the semantics of concurrent programs running within the context of a relaxed memory model. Our framework features a “temporary store” where the memory operations issued by the threads are recorded, in program order. A memory model then specifies the conditions under which a pending operation from this sequence is allowed to be globally performed, possibly out of order. The memory model also involves a “write grain,” accounting for architectures where a thread may read a write that is not yet globally visible. Our formal model is supported by a software simulator, allowing us to run litmus tests in our semantics.

## 1 Introduction

The hardware evolution towards multicore architectures means that the most significant future performance gains will rely on using concurrent programming techniques at the application level. This is currently supported by some general purpose programming languages, such as JAVA or C/C++. The semantics that is assumed by the application programmer using such a concurrent language is the standard *interleaving* semantics, also known as *sequential consistency* (SC, [11]). This is also the semantics assumed by most verification methods. However, it is well-known [2] that this semantics is *not* the one we observe when running concurrent programs in optimizing execution environments, i.e. compilers and hardware architectures, which are designed to run sequential programs as fast as possible. For instance, let us consider the program

$$\begin{array}{l} p := tt; \quad \parallel \quad q := tt; \\ r_0 := !q \quad \parallel \quad r_1 := !p \end{array} \quad (1)$$

where we use ML’s notation  $!p$  for dereferencing the pointer – or reference, in ML’s jargon –  $p$ . If the initial state is such that the values of  $p$  and  $q$  are both  $ff$ , we cannot get, by the standard interleaving semantics, a final state where the value of both  $r_0$  and  $r_1$  is  $ff$ . Still, running this program may, on most multiprocessor architectures, produce this outcome. This is the case for instance on a TSO machine [2] where the writes  $p := tt$  and  $q := tt$  are put in (distinct) buffers attached with the processors, and thus delayed with respect to the reads  $!q$  and  $!p$  respectively, which get their value from the (not yet updated) main memory. In effect, the reads are *reordered* with respect to the writes. Other reordering optimizations, which may also be introduced by compilers, yield similar failures of sequential consistency (see the survey [2]), yet sequential consistency is generally considered as a suitable abstraction at the application programming level.

Then a question is: how to ensure that concurrent programs running in a given optimized execution environment appear, from the programmer’s point of view, to be sequentially consistent, behaving as in the interleaving semantics? A classical answer is: the program should not give rise to data races in its sequentially consistent behavior, keeping apart some specific synchronization variables, like locks. This is known as the “DRF (Data Race Free) guarantee,” that was first stated in [3, 10], and has been widely advocated since then (see [1, 6, 12]). An attractive feature of the DRF guarantee is that it allows the

programmer to reason in terms of the standard interleaving semantics alone. However, there are still some issues with this property. First, one would sometimes like to know what racy programs do, for safety reasons as in JAVA for instance, or for debugging purposes, or else for the purpose of establishing the validity of program transformations in a relaxed memory model. Second, the DRF guarantee is more an axiom, or a contract, than a guarantee: once stated that racy programs have undefined semantics, how do we indeed guarantee that a particular implementation provides sequentially consistent semantics for race free programs?

Clearly, to address such a question, there is a preliminary problem to solve, namely: how do we describe the actual behavior of concurrent programs running in a relaxed execution environment? This is known to be a difficult problem. For instance, to the best of our knowledge, the JAVA Memory Model (JMM) [12] is still not sound. Moreover, its current formal description is fairly complex. To our view, this is true also regarding the formalization of the C++ primitives for concurrent programming [5, 6], or the formalization of the PowerPC memory model [14]. Our intention here is *not* to describe a specific memory model, be it a hardware, low-level one, or the memory model for a high-level concurrent programming language, like JAVA or C++. Our aim is rather to design a semantical framework that would be

- flexible enough to allow for the description of a wide range of memory models;
- simple enough to support the intuition of the programmer and the implementer;
- precise enough to support formal analysis of programs.

(Since we are talking about programs, there will be a programming language, but the particular choice we make is not essential to our work.)

To address the problem stated above, we adopt the *operational* style advocated in [7, 15], which, besides being “*widely accessible to working programmers*” [15], allows us to use standard techniques to analyse and verify programs, proving properties such as the DRF guarantee [7] for instance. In [7, 15], write buffers are explicitly introduced in the semantic framework, and their behavior accounts for some of the reorderings mentioned above. The model we propose goes beyond the simple operational model for write buffering, by introducing into the semantic framework a different intermediate structure, between the shared memory and the threads. The idea is to record in this structure the memory operations – reads and write, or loads and stores, in low level terminology – that are issued by the threads, in program order. We call the sequence of pending operations issued by the threads a *temporary store*. Then these operations may be delayed, and finally performed, with regard to the global shared memory, out of order. To be globally performed, an operation from the temporary store must be allowed to overtake the operations that were previously issued, that is, the operations that precede it in the temporary store. Then a key ingredient in our model is the *commutability predicate*, that characterizes, for a given memory model, the conditions under which an operation from the temporary store may be performed early. This accounts in particular for the usual relaxations of the program order, and also for the semantics of synchronization constructs, like barriers.

In some relaxed memory models, some fairly complex behaviors arise that cannot be fully explained by relaxations of the program order. These behaviors are caused by the failure of write atomicity [2]. To deal with this feature, we introduce another key ingredient to characterize a memory model. In our framework, with each pending write is associated a *visibility*, that is the set of threads that can see it, and can therefore read the written value. Depending on the memory model, and more specifically on the (abstract) communicating network topology between threads (or processors), not any set of threads is allowed to be a legitimate visibility. For instance, the Sequential Consistency model [11] only allows the empty set, and the singletons to be visibility sets, meaning that only the thread issuing a write can

see it before it is globally performed. Then the definition of a memory model involves, besides the commutability predicate, a “*write grain*,” which specifies which visibility a write is allowed to acquire. This accounts for the fact that some threads can read others’ writes early [2]. Our model then easily explains, in operational terms, the behavior of a series of “litmus tests,” such as IRIW, WRC, RWC and CC discussed in [6] for instance, and the tests from [14], designed to investigate the PowerPC architecture. Regarding this particular memory model, we found only three cases where our formalization of the main PowerPC barriers is more strict than the one of [14]. However, these are cases where the behavior that our model forbids was never observed during the extensive experiments on real machines done by Sarkar & al. (and reported in files available on the web as a supplement to their paper). On the other hand, for all the litmus tests that can be expressed in our language, the behaviors that are observed in Sarkar’s experiments on real machines are accounted for in our model, which therefore is not invalidated by these experimental results. Needless to say, the experimental test suite provided by Sarkar & al. was invaluable for us to see which behaviors the model should explain. These litmus tests were, among others, run in a software simulator that we have built to experiment with our semantics.

Compared to other formalizations of relaxed semantics, our model is truly operational. By this we mean that it consists in a set of rules that specify what can be the next step to perform, to go from one configuration to another. This contrasts with [8] for instance, where a whole sequence of steps is only deemed a valid behavior if it can be shown equivalent to a computation in normal order. We notice that, again contrasting [8], our model preserves a notion of *causality*: a read can only return a value that is present in the shared memory, or that is previously written by some thread. Our notion of a temporary store is quite similar to the “reorder box” of [13], but formulated in the standard framework of programming language semantics. In some approaches, including [4] and [9], the various relaxations of the order of memory operations are described by means of rewrite rules on traces of memory operations (which again are similar to our temporary store). Notice that permuting operations in a trace is, in general, a cyclic process. Regarding the relaxation of write atomicity, and more specifically the read-others’-write-early capability (as illustrated by the IRIW litmus test in subsection 4.2 below, where no relaxation of the program order is involved), the only work we know that proposes a formal operational formulation of this capability is [14] which, to our view, provides a quite complicated semantics of this feature. We think that our formalization, by means of write visibility, is much simpler than the one of [14]. Moreover, by relying on a concrete notion of state, our model should be more amenable to standard programming languages proof techniques, like for establishing that programs only exhibit sequentially consistent behavior [7], or more generally to achieve mathematical analysis and verification of programs.

**Note.** The web page <http://www-sop.inria.fr/index/MemoryModels/> contains a full version of the paper. The additional contents are explained in the text.

## 2 The Core Language

Our language is a higher-order, imperative and concurrent language à la ML, that is a call-by-value  $\lambda$ -calculus extended with constructs to deal with a mutable store. (This choice of a functional core language is largely a matter of taste.) In order to simplify some technical developments, the syntax is given in administrative normal form. In this way, only one construction, namely the application of a function to an argument, is responsible for introducing an evaluation order (the program order). Assuming given a

set  $\mathcal{Var}$  of variables, ranged over by  $x, y, z, \dots$ , the syntax is as follows:

$$\begin{array}{ll}
v ::= x \mid \lambda x e \mid tt \mid ff \mid () & \text{values} \\
b \in \mathcal{Bar} & \text{barriers} \\
e \in \mathcal{L} ::= v \mid (ve) \mid (\text{if } v \text{ then } e_0 \text{ else } e_1) & \text{expressions} \\
\quad \mid (\text{ref } v) \mid (!v) \mid (v_0 := v_1) \mid b &
\end{array}$$

As usual, the variable  $x$  is bound in an expression  $\lambda x e$ , and we consider expressions up to  $\alpha$ -conversion, that is up to the renaming of bound variables. The capture-avoiding substitution of a value  $v$  for the free occurrences of  $x$  in  $e$  is denoted  $\{x \mapsto v\}e$ . We shall use some standard abbreviations like  $(\text{let } x = e_0 \text{ in } e_1)$  for  $(\lambda x e_1 e_0)$ , which is also denoted  $e_0 ; e_1$  whenever  $x$  does not occur free in  $e_1$ . We shall sometimes (in the examples) write expressions in standard syntax, which is easily converted to administrative form, like for instance converting  $(e_0 e_1)$  into  $(\text{let } f = e_0 \text{ in } (f e_1))$ , or  $(v := e)$  into  $(\text{let } x = e \text{ in } (v := x))$ .

The barrier constructs are “no-ops” in the abstract (interleaving) semantics of the language. Such synchronization constructs are often considered low-level. However, we believe they can also be useful in a high-level concurrent programming language, for “relaxed memory aware” programming (see [6]). We do not focus on a particular set  $\mathcal{Bar}$  here, so the language should actually be  $\mathcal{L}(\mathcal{Bar})$ , but in the following we shall give some examples of useful barriers, and see how to formalize their semantics. In the full version of this paper we also consider constructs for spawning and joining threads, and for locking references.

As usual, to formalize the operational semantics of the language, we have to extend it, introducing some run-time values. Namely, we assume given a set  $\mathcal{Ref}$  of *references*, ranged over by  $p, q, \dots$ . These are the values returned by reference creation. In the examples we shall examine, the names  $r_i$  suggest that such a reference should actually be regarded as a register, which is not shared with other threads. We still use  $e$  to range not only over expressions of the source language  $\mathcal{L}$ , but also over expressions built with run-time values, that is, possibly involving references.

A step in the semantics consists in evaluating a redex inside an *evaluation context*. The syntax of the latter is as follows:

$$\mathbf{E} ::= [] \mid (v \mathbf{E}) \quad \text{evaluation contexts}$$

As usual, we denote by  $\mathbf{E}[e]$  the run-time expression obtained by filling the hole in  $\mathbf{E}$  by  $e$ . The semantics is specified as small step transitions  $C \rightarrow C'$  between configurations  $C, C'$  of the form  $(S, T)$  where  $S$  and  $T$  are respectively the *store* and the *thread system*. To define the latter, we assume given a set  $\mathcal{Tid}$  of *thread identifiers*, ranged over by  $t$ . The store  $S$ , also called here the *memory*, is a mapping from a finite set  $\text{dom}(S)$  of references to values. The thread system  $T$  is a mapping from a finite set  $\text{dom}(T)$  of thread identifiers, subset of  $\mathcal{Tid}$ , to run-time expressions. If  $\text{dom}(T) = \{t_1, \dots, t_n\}$  and  $T(t_i) = e_i$  we also write  $T$  as

$$(t_1, e_1) \parallel \dots \parallel (t_n, e_n)$$

The reference operational semantics, that is the standard interleaving semantics, is given in Figure 1.

$$\begin{aligned}
(S, (t, \mathbf{E}[(\lambda x ev)]) \parallel T) &\rightarrow (S, (t, \mathbf{E}[\{x \mapsto v\}e]) \parallel T) \\
(S, (t, \mathbf{E}[(\text{if } tt \text{ then } e_0 \text{ else } e_1)]) \parallel T) &\rightarrow (S, (t, \mathbf{E}[e_0]) \parallel T) \\
(S, (t, \mathbf{E}[(\text{if } ff \text{ then } e_0 \text{ else } e_1)]) \parallel T) &\rightarrow (S, (t, \mathbf{E}[e_1]) \parallel T) \\
(S, (t, \mathbf{E}[(\text{ref } v)]) \parallel T) &\rightarrow (S \cup \{p \mapsto v\}, (t, \mathbf{E}[p]) \parallel T) && \text{if } p \notin \text{dom}(S) \\
(S, (t, \mathbf{E}[(!p)]) \parallel T) &\rightarrow (S, (t, \mathbf{E}[v]) \parallel T) && \text{if } S(p) = v \\
(S, (t, \mathbf{E}[(p := v)]) \parallel T) &\rightarrow (S[p := v], (t, \mathbf{E}[]) \parallel T) \\
(S, (t, \mathbf{E}[b]) \parallel T) &\rightarrow (S, (t, \mathbf{E}[]) \parallel T)
\end{aligned}$$

---

**Figure 1: Reference Operational Semantics**

---

### 3 Relaxed Computations

#### 3.1 Preliminary Definitions

The relaxed operational semantics is formalized by means of small steps transitions

$$RC \xrightarrow{\mathcal{M}} RC'$$

between relaxed configurations  $RC$  and  $RC'$ . The  $\mathcal{M}$  parameter is the *memory model*. Let us first describe the relaxed configurations. For this we need to introduce some technical ingredients. In the relaxed semantics a read can be issued by a thread, evaluating a subexpression  $(!p)$ , while not immediately returning a value. In this way the read can be overtaken by a subsequent operation. To model this, we shall dynamically assign to each read operation a unique identifier, returned as the value read. That is, we extend the language with names, or *identifiers*, to point to future values. The set  $\mathcal{Ident}$  of identifiers is assumed to be disjoint from  $\mathcal{Var} \cup \mathcal{Ref}$ , and is ranged over by  $\iota$ . We shall use  $\varrho$  to range over  $\mathcal{Ref} \cup \mathcal{Ident}$ . The identifiers  $\iota \in \mathcal{Ident}$  are *values* in the extended language, still denoted by  $v$ , but notice that  $\mathcal{Val}$  denotes the set of (not relaxed) values, that do not contain any identifier  $\iota$ . We shall require that only true values, not relaxed ones, can be stored. It should be clear that substituting a relaxed value  $v$  for an identifier  $\iota$  in an expression  $e$  results in a valid expression, denoted  $\{\iota \mapsto v\}e$ .

Our next technical ingredient is the set  $\mathcal{Mop}(\mathcal{L})$  of *memory operations* in the language  $\mathcal{L}$ . These represent the instructions that are issued by the threads, but are not necessarily immediately performed. The set  $\mathcal{Mop}(\mathcal{L})$  of memory operations comprises the barriers  $b \in \mathcal{Bar}$  and the *read* and *write* operations, respectively denoted  $\text{rd}_{\varrho, \iota}$  and  $\text{wr}_{\varrho, v}^{W, I}$  where  $\varrho \in \mathcal{Ref} \cup \mathcal{Ident}$ ,  $\iota \in \mathcal{Ident}$ ,  $W \subseteq \mathcal{Tid}$  is a set of thread names, and  $I \subseteq \mathcal{Ident}$  is a set of identifiers. We call the set  $W$  in  $\text{wr}_{\varrho, v}^{W, I}$  the *visibility* of the write (we comment on this, and on the  $I$  component, below). Finally, we introduce operations of the form  $\overline{\text{rd}}_{\iota}$  that we call a *read mark*, meaning that a read has occurred, where  $\iota$  serves as identifying the corresponding write. That is, the syntax of memory operations is as follows:

$$\xi \in \mathcal{Mop}(\mathcal{L}) ::= \text{rd}_{\varrho, \iota} \mid \overline{\text{rd}}_{\iota} \mid \text{wr}_{\varrho, v}^{W, I} \mid b$$

We can now define a *relaxed configuration*  $RC$  as a triple  $RC = (S, \sigma, T)$  where  $S$  and  $T$  are as above, and  $\sigma$  is a sequence of pairs  $(t, \xi)$ , where  $t \in \mathcal{Tid}$  is a thread name and  $\xi \in \mathcal{Mop}(\mathcal{L})$  a memory operation. The meaning of  $(t, \xi)$  in a sequence  $\sigma$  is that  $\xi$  is a memory operation issued by thread  $t$ . The sequence  $\sigma$  then records the pending memory operations issued by the threads, which will not necessarily be performed (on the shared memory) in the order in which they appear in  $\sigma$ . We shall call such a  $\sigma$  a *temporary store*. We denote by  $\Sigma_{\mathcal{L}}$  the set  $\mathcal{Tid} \times \mathcal{Mop}(\mathcal{L})$ , so that the set of temporary stores is  $\Sigma_{\mathcal{L}}^*$ , the set of finite sequences over  $\Sigma_{\mathcal{L}}$ . We denote by  $\varepsilon$  the empty sequence, and we write  $\sigma \cdot \sigma'$  for the

$$\begin{aligned}
(S, \sigma, (t, \mathbf{E}[(\lambda x ev)]) \parallel T) &\hookrightarrow (S, \sigma, (t, \mathbf{E}[\{x \mapsto v\}e]) \parallel T) \\
(S, \sigma, (t, \mathbf{E}[(\text{if } tt \text{ then } e_0 \text{ else } e_1)]) \parallel T) &\hookrightarrow (S, \sigma, (t, \mathbf{E}[e_0]) \parallel T) \\
(S, \sigma, (t, \mathbf{E}[(\text{if } ff \text{ then } e_0 \text{ else } e_1)]) \parallel T) &\hookrightarrow (S, \sigma, (t, \mathbf{E}[e_1]) \parallel T) \\
(S, \sigma, (t, \mathbf{E}[(\text{ref } v)]) \parallel T) &\hookrightarrow (S, \sigma \cdot (t, \text{wr}_{p,v}^{\emptyset, \emptyset}), (t, \mathbf{E}[p]) \parallel T) \quad p \text{ fresh} \\
(S, \sigma, (t, \mathbf{E}[(! \varrho)]) \parallel T) &\hookrightarrow (S, \sigma \cdot (t, \text{rd}_{\varrho, \iota}), (t, \mathbf{E}[\iota]) \parallel T) \quad \iota \text{ fresh} \\
(S, \sigma, (t, \mathbf{E}[(\varrho := v)]) \parallel T) &\hookrightarrow (S, \sigma \cdot (t, \text{wr}_{\varrho, v}^{\emptyset, \emptyset}), (t, \mathbf{E}[\emptyset]) \parallel T) \\
(S, \sigma, (t, \mathbf{E}[b]) \parallel T) &\hookrightarrow (S, \sigma \cdot (t, b), (t, \mathbf{E}[\emptyset]) \parallel T)
\end{aligned}$$

---

**Figure 2:  $\mathcal{M}$ -Relaxed Operational Semantics (Threads)**

---

concatenation of the two sequences  $\sigma$  and  $\sigma'$ . We say that a relaxed configuration  $(S, \sigma, T)$  is *normal* whenever  $\sigma = \varepsilon$ , and no expression occurring in the configuration (that is, in the store  $S$  or the thread pool  $T$ ) contains an identifier.

### 3.2 The Relaxed Semantics

We present the relaxed semantics in two parts: the first one describes the evaluation of the threads, that is, the contribution of the  $T$  component in the semantics, and the second one explains how the memory operations from the temporary store  $\sigma$  are performed. One could say that the instructions executed by the threads are “locally performed,” while the operations executed from the temporary store will be “globally performed,” as their effect is made visible to the other threads. The particular memory model  $\mathcal{M}$  is irrelevant to the local evaluation of threads, and therefore in Figure 2, which presents this evaluation, we simplify  $\xrightarrow{\mathcal{M}}$  into  $\hookrightarrow$ . In the rules for reducing  $(\text{ref } v)$  and  $(! \varrho)$ , “ $p$  fresh” and “ $\iota$  fresh” mean that  $p$  and  $\iota$  do not occur in the configuration.

The relaxed semantics differs from the reference semantics in several ways. The main difference is that the effect on the memory – if any – of evaluating the code is delayed. Namely, instead of updating the memory, the effect of evaluating  $(p := v)$ , or more generally  $(\varrho := v)$  where the exact reference to update may still be undetermined, consists in recording the write operation, with a default empty visibility, at the end of the sequence of pending memory operations. Creating a reference, reducing  $(\text{ref } v)$ , has the same effect, once a new reference name is obtained. Reducing a dereferencing operation  $(! \varrho)$  does not immediately return a proper value, but creates and returns a fresh identifier  $\iota \in \mathcal{Ident}$ , to be later bound to a definite value, while appending a corresponding read operation to the temporary store. A barrier just appends itself at the end of the temporary store. Notice that the rules of Figure 2 are not concerned with the store  $S$ . As an example, considering the thread system  $T$  of Example (1) given in the Introduction, assigning the thread names  $t_0$  to the thread on the left and  $t_1$  to the one on the right, assuming  $\iota_0$  and  $\iota_1$  to be fresh identifiers, and executing  $t_0$  followed by  $t_1$  we can reach the following temporal store:

$$\sigma = (t_0, \text{wr}_{p, tt}^{\emptyset, \emptyset}) \cdot (t_0, \text{rd}_{q, \iota_0}) \cdot (t_1, \text{wr}_{q, tt}^{\emptyset, \emptyset}) \cdot (t_1, \text{rd}_{p, \iota_1})$$

A relaxed configuration  $(S, \sigma, T)$  can also perform actions that originate from the temporary store  $\sigma$ . These steps are performed independently from the evaluation of threads, in an asynchronous way. To define these transitions, we need to say a bit more about the memory model  $\mathcal{M}$ . We shall not focus here on a particular memory model, since our purpose is to design a general framework for describing the semantics of concurrent programs in a relaxed setting. However, we shall make some minimal

hypotheses about the  $\mathcal{M}$  parameter. But let us first say what  $\mathcal{M}$  consists of. We assume that this is a pair  $\mathcal{M} = (\uparrow, \mathcal{W})$  made of a *commutability* predicate  $\uparrow$  and a “*write grain*”  $\mathcal{W}$ . These two components provide a formalization of the approach of Adve and Gharachroloo in [2], who distinguish these two key features as the basis for categorizing memory models.

The commutability predicate delineates the relaxations of the program order that are allowed in the weak semantics under consideration, and in particular it provides semantics for barriers. This first component  $\uparrow$  of a memory model is a subset of  $\Sigma_{\mathcal{L}}^* \times \Sigma_{\mathcal{L}}$ , that is a binary predicate relating temporary stores  $\sigma \in \Sigma_{\mathcal{L}}^*$  with issued operations  $(t, \xi) \in \Sigma_{\mathcal{L}}$ . This predicate is expressing which operations issued by some thread are allowed to be performed early, that is, out of order in the relaxed semantics. Indeed, if the temporary store is  $\sigma \cdot (t, \xi) \cdot \sigma'$  with  $\sigma \uparrow (t, \xi)$ , then the operation  $\xi$  from thread  $t$  may, in general, be globally performed, as if it were the first one, and removed from the temporary store. We read  $\sigma \uparrow (t, \xi)$  as:  $(t, \xi)$  may overtake  $\sigma$ , or:  $\sigma$  allows  $(t, \xi)$  to be performed. We assume, as an axiom satisfied by any memory model, that the first operation in the temporary store is always allowed to execute, that is, for any  $\xi$  and  $t$ :

$$\varepsilon \uparrow (t, \xi) \tag{E}$$

The  $\mathcal{W}$  component of a memory model is a set of subsets of  $\mathcal{Tid}$ , comprising the set of the allowed write visibilities. In the relaxed semantics, with each write operation  $wr_{\rho, v}^{W, I}$  is associated a visibility  $W$ , which is a (possibly empty) set of thread identifiers. (We delay the discussion of the set  $I$  to the subsection 3.3.) The default visibility of a write when it is issued, as prescribed in Figure 2, is  $\emptyset$ , so we assume that for any memory model this is an allowed visibility, that is  $\emptyset \in \mathcal{W}$ . The visibility of a write may dynamically evolve (within  $\mathcal{W}$ ), but we shall assume that it can only grow. The threads in  $W$  see the write, while in the temporary store, and these threads can therefore read the corresponding value, possibly before it is globally visible (in that case the  $I$  component of the write is extended). The  $\mathcal{W}$  component allows us to deal with *write atomicity*, or, more generally, with the extent to which the threads are allowed to read each others writes. In a hardware architecture, this is determined by a particular topology and behavior of the interconnection network. Thus, for example, assuming three different threads  $t, t'$  and  $t''$ , a write  $wr_{\rho, v}^{\{t, t'\}, I}$  in the temporal store can be prematurely read by thread  $t$  and  $t'$  but not from thread  $t''$ .

We can now formulate the rules for the  $\xrightarrow{\mathcal{M}}$  transitions as regards the memory. These are given in Figure 3, with  $(\uparrow, \mathcal{W}) = \mathcal{M}$ . In the rule  $R2$  we use a restricted commutability predicate  $\sigma \uparrow^{\mathcal{B}ar} (t, \xi)$ , ignoring the operations from  $\sigma$  that are not synchronization operations, that is:

$$\sigma \uparrow^{\mathcal{B}ar} (t, \xi) \Leftrightarrow_{\text{def}} \sigma \upharpoonright \mathcal{B}ar \uparrow (t, \xi)$$

where  $\sigma \upharpoonright \mathcal{B}ar$  is the restriction of the sequence  $\sigma$  to the set  $\mathcal{B}ar$ , that is the subsequence of  $\sigma$  containing only the issued barriers.

We now comment the rules. In all cases but the early ones ( $R2$  and  $R5$ ), performing an operation from the temporary store  $\sigma$  consists in checking that the operation can be moved, up to  $\uparrow$ , at the head of  $\sigma$ , and then in removing the operation from  $\sigma$  while possibly performing some effect. Namely, such an effect is produced when the performed operation is a read or a write. The reference that is concerned by the effect must be known in these cases. A read may also return a value if it can be moved to a corresponding, visible write ( $R2$ ). In this case, the read operation should not be blocked by barriers previously issued but not yet globally performed. This is expressed as  $\sigma_0 \uparrow^{\mathcal{B}ar} (t, rd_{p, \iota})$ . The read operation does not completely vanish, but is transformed in a read mark  $\overline{rd}_{\iota}$ , where  $\iota$  identifies the matching write. When read is resolved using rule  $R2$ , the identifier of the read is added to the  $I$  set of the write used to serve the read. The purpose of this set is to maintain the ordering of some memory operations, as explained below

$$\begin{array}{l}
(S, \sigma, T) \xrightarrow{\uparrow, \mathcal{W}} (S, \{\iota \mapsto v\}(\sigma_0 \cdot \sigma_1, T)) \quad R1 \text{ (read)} \\
\text{if } \sigma = \sigma_0 \cdot (t, \text{rd}_{p,\iota}) \cdot \sigma_1 \ \& \ \sigma_0 \uparrow (t, \text{rd}_{p,\iota}) \ \& \ S(p) = v \\
(S, \sigma, T) \xrightarrow{\uparrow, \mathcal{W}} (S, \{\iota \mapsto v\}(\sigma_0 \cdot (t', \text{wr}_{p,v}^{W, I \cup \{\iota\}}) \cdot \sigma_1 \cdot (t, \overline{\text{rd}}_\iota) \cdot \sigma_2, T)) \quad R2 \text{ (read early)} \\
\text{if } \sigma = \sigma_0 \cdot (t', \text{wr}_{p,v}^{W, I}) \cdot \sigma_1 \cdot (t, \text{rd}_{p,\iota}) \cdot \sigma_2 \ \& \ t \in W \ \& \\
\sigma_1 \uparrow (t, \text{rd}_{p,\iota}) \ \& \ \sigma_0 \uparrow^{\text{Bar}} (t, \text{rd}_{p,\iota}) \\
(S, \sigma, T) \xrightarrow{\uparrow, \mathcal{W}} (S, \sigma_0 \cdot \sigma_1, T) \quad R3 \text{ (read)} \\
\text{if } \sigma = \sigma_0 \cdot (t, \overline{\text{rd}}_\iota) \cdot \sigma_1 \ \& \ \sigma_0 \uparrow (t, \overline{\text{rd}}_\iota) \ \text{or} \\
\sigma_0 = \delta_0 \cdot (t', \text{wr}_{p,v}^{\text{tid}, I \cup \{\iota\}}) \cdot \delta_1 \ \& \ \delta_0 \uparrow (t', \text{wr}_{p,v}^{\text{tid}, I \cup \{\iota\}}) \\
(S, \sigma, T) \xrightarrow{\uparrow, \mathcal{W}} (S[p := v], \sigma_0 \cdot \sigma_1, T) \quad R4 \text{ (write)} \\
\text{if } \sigma = \sigma_0 \cdot (t, \text{wr}_{p,v}^{W, I}) \cdot \sigma_1 \ \& \ \sigma_0 \uparrow (t, \text{wr}_{p,v}^{W, I}) \ \& \ v \in \text{Val} \\
(S, \sigma, T) \xrightarrow{\uparrow, \mathcal{W}} (S, \sigma_0 \cdot (t, \text{wr}_{\varrho,v}^{W', I}) \cdot \sigma_1, T) \quad R5 \text{ (write early)} \\
\text{if } \sigma = \sigma_0 \cdot (t, \text{wr}_{\varrho,v}^{W, I}) \cdot \sigma_1 \ \& \ t \in W' \ \& \ W \subset W' \in \mathcal{W} \\
(S, \sigma, T) \xrightarrow{\uparrow, \mathcal{W}} (S, \sigma_0 \cdot \sigma_1, T) \quad R6 \text{ (barrier)} \\
\text{if } \sigma = \sigma_0 \cdot (t, b) \cdot \sigma_1 \ \& \ \sigma_0 \uparrow (t, b)
\end{array}$$

---

**Figure 3:  $\mathcal{M}$ -Relaxed Operational Semantics (Memory)**

---

in the subsection 3.3. As we shall see in Section 4.3, a read mark is only useful in relation with barriers and can be eliminated from the temporary store as specified by *R3*. Notice that when we say that the read  $(t, \text{rd}_{p,\iota})$  can be “moved,” this is only an image: there is no transformation of the temporary store, but only a condition on it, namely, in *R1*,  $\sigma_0 \uparrow (t, \text{rd}_{p,\iota})$ . In the rules *R1* and *R2* for read operations, there is a global replacement of the identifier  $\iota$  associated with the read by the actual value  $v$  that is read: in these rules  $\{\iota \mapsto v\}(\sigma, T)$  stands for such a replacement, which does not affect the  $I$  component in the writes. (Recall that we required that an identifier such as  $\iota$  cannot appear in the store.) Similarly, a write operation  $\text{wr}_{\varrho,v}^{W, I}$  from the temporary store  $\sigma_0 \cdot (t, \text{wr}_{\varrho,v}^{W, I}) \cdot \sigma_1$  may update (rule *R4*) the memory when  $\varrho$  is a reference  $p$ ,  $v$  is in  $\text{Val}$  and the write is allowed to commute with the preceding operations, that is  $\sigma_0 \uparrow (t, \text{wr}_{p,v}^W)$ . An early write action in *R5* has only the effect of modifying the temporary store, by extending the visibility of the write to more threads.

An obvious remark about the relaxed semantics is that it contains in a sense the interleaving semantics, with temporary stores containing at most one operation: one can mimick a transition of the latter either by one local step, or by a local step immediately followed by a global action. One can also immediately see that if  $\mathcal{W} = \{\emptyset\}$ , then the rule *R5* cannot be used, and consequently no early read can take place. If, in addition to  $\emptyset$ ,  $\mathcal{W}$  only contains the singletons  $\{t\}$  for  $t \in \text{tid}$ , the read early rule *R2* is restricted to the “read-own-write-early” capability [2]. In the write early rule *R5*, the requirement  $t \in W'$  means that we do not consider memory models where the “read-others’-write-early” capability would be enabled, but not the “read-own-write-early” one (again, see [2]).

In the full version of the paper we also provide a formalization of speculative computations in our framework.

### 3.3 Memory Models: Requirements

In the next section we briefly illustrate the expressive power of our framework for relaxed computations, by showing some programs exhibiting behaviors that are *not* allowed by the reference semantics. (Many more examples are given in the full version of this paper.) Most of these examples are standard “litmus tests” found in the literature about memory models, that reveal in particular the consequences of relaxing in various ways the normal order of evaluation. In most cases, the relaxations of program order can be specified by a binary relation on  $\Sigma_{\mathcal{L}}$ . It is actually more convenient to use the converse relation, which can usually be more concisely described. We call this a *precedence* relation. Given such a binary relation  $\mathcal{P}$  on pairs  $(t, \xi) \in \Sigma_{\mathcal{L}}$ , the commutability relation is supposed to satisfy

$$(\omega, \xi) \mathcal{P} (\omega', \xi') \Rightarrow \forall \sigma, \sigma'. \neg(\sigma \cdot (\omega, \xi) \cdot \sigma' \uparrow (\omega', \xi'))$$

That is, an operation in a temporary store is prevented from being globally performed by another, previously issued one, that has precedence over it. A more positive formulation of this property is:

$$\sigma \cdot (\omega, \xi) \cdot \sigma' \uparrow (\omega', \xi') \Rightarrow \neg((\omega, \xi) \mathcal{P} (\omega', \xi')) \quad (\text{A}_{\mathcal{P}})$$

Before examining various relaxations of the program order, by way of examples, we discuss some precedence pairs that are most often assumed in memory models. For instance, if we do not assume any constraint as regards the commutability of writes, from the program

$$(p := tt); (p := ff)$$

we could get as a possible outcome a state where the value of  $p$  in the memory is  $tt$ , by commuting the second write before the first. This is clearly unacceptable, because this violates the semantics of sequential programs. Then we should assume that two writes on the same reference issued by the same thread cannot be permuted. Similarly, a write should not be overtaken by a read on the same reference issued by the same thread, and conversely, otherwise the semantics of the sequential programs

$$\begin{aligned} (p := tt); (r := !p) \\ (r := !p); (p := tt) \end{aligned}$$

would be violated. We shall then require that any memory model satisfies axiom  $(\text{A}_{\blacktriangleleft})$  where  $\blacktriangleleft$  is the minimal precedence relation enjoying the following properties, where the free symbols are implicitly universally quantified:

$$\left. \begin{array}{l} \varrho \in \{\varrho'\} \cup \mathcal{I}dent \ \& \\ t' \in W \cup \{t\} \text{ or } I \neq \emptyset \neq I' \end{array} \right\} \Rightarrow \left\{ \begin{array}{l} (t, \text{wr}_{\varrho, v}^{W, I}) \blacktriangleleft (t', \text{rd}_{\varrho', \iota}) \ \& \\ (t, \text{wr}_{\varrho, v}^{W, I}) \blacktriangleleft (t', \text{wr}_{\varrho', v'}^{W', I'}) \end{array} \right.$$

$$\iota \in I \Rightarrow (t, \text{wr}_{p, v}^{W, I}) \blacktriangleleft (t', \overline{\text{rd}}_{\iota})$$

$$\varrho \in \{\varrho'\} \cup \mathcal{I}dent \Rightarrow (t, \text{rd}_{\varrho, \iota}) \blacktriangleleft (t, \text{wr}_{\varrho', v}^{W, I})$$

These properties ensure in particular that the precedence relations discussed above are enforced: among the operations of a given thread, one cannot commute for instance a read and a write on the same reference. Notice however that it is not required that the program order is maintained as regards two reads on the same reference. Therefore, from the program

$$p := tt \parallel \begin{array}{l} r_0 := !p; \\ r_1 := !p \end{array}$$

if initially  $S(p) = ff$ , we could end up in a state where the value of  $r_1$  is  $ff$ , while the one for  $r_0$  is  $tt$ . If one wishes to preclude such a behavior, one can simply add

$$\varrho \in \{p\} \cup \overline{Ident} \Rightarrow (t, rd_{\varrho, \iota}) \mathcal{P} (t, rd_{p, \iota'})$$

to the precedence relation.

There are three cases where the  $\blacktriangleleft$  precedence relates two distinct threads. The first one, that is  $(t, wr_{\varrho, v}^{W, I}) \blacktriangleleft (t', rd_{p, \iota})$  where  $t' \in W$ , means that a thread  $t'$  “sees” the writes, previously issued by other threads, that include  $t'$  in their scope – the same holds with  $(t, wr_{\varrho, v}^{W, I}) \blacktriangleleft (t', wr_{\varrho', v'}^{W', I'})$  where  $t' \in W$ . The precedence  $(t, wr_{\varrho, v}^{W, I}) \blacktriangleleft (t', wr_{\varrho', v'}^{W', I'})$  where  $I \neq \emptyset \neq I'$  means that the order of writes on a given reference must be respected if these writes have been read by some threads (this is similar to the “coherence order” of [14]). Finally,  $(t, wr_{p, v}^{W, I}) \blacktriangleleft (t', rd_{\iota})$  where  $\iota \in I$  means that an early read cannot vanish from the temporary store before the corresponding write. These two properties explains the role of the  $I$  component in our model. One should notice that *no* specific precedence assumption is made at this point regarding the barriers. Then our definition of the notion of a memory model is as follows:

**DEFINITION (MEMORY MODELS) 3.1.** A memory model  $\mathcal{M}$  for  $\mathcal{L}$  is a pair  $(\blacktriangleright, \mathcal{W})$  where  $\emptyset \in \mathcal{W}$ , and the commutability predicate  $\blacktriangleright \subseteq \Sigma_{\mathcal{L}}^* \times \Sigma_{\mathcal{L}}$  satisfies the axioms (E) and (A $\blacktriangleleft$ ).

As an example memory model, one can define  $SC$ , for Sequential Consistency, as

$$SC = (\{\varepsilon\} \times \Sigma_{\mathcal{L}}, \{\emptyset\} \cup \{\{t\} \mid t \in \mathcal{T}id\})$$

which obviously satisfies Definition 3.1 (the axiom (A $\blacktriangleleft$ ) is vacuously true). All the examples discussed in the following section hold in the minimal, or *most relaxed*, memory model  $\mathcal{M}_{\blacktriangleleft}(\mathcal{L}) = (\blacktriangleright_{\blacktriangleleft}, 2^{\mathcal{T}id})$ , where  $\blacktriangleright_{\blacktriangleleft}$  is the largest commutability predicate satisfying (A $\blacktriangleleft$ ),  $2^{\mathcal{T}id}$  is the set of all subsets of  $\mathcal{T}id$ .

In our work we mainly use commutability properties that are generated by precedence relations, in the sense of axiom (A $\mathcal{P}$ ). Then one could think of defining a memory model as a pair  $(\mathcal{P}, \mathcal{W})$ , instead of  $(\blacktriangleright, \mathcal{W})$ . However, we shall see in Section 4.3 a case where this is not general enough. More precisely, we shall see a case where we have to say that  $\neg(\sigma \blacktriangleright (t, \xi))$ , not on the basis that  $\sigma$  contains an operation that has precedence over  $(t, \xi)$ , but because there is a subsequence of  $\sigma$  which, as a whole, has precedence over it.

## 4 Examples

Now we examine a few examples of programs exhibiting relaxed behaviors that are not allowed by the reference semantics. (As mentioned above, in the full version of this paper we examine many more examples.) In all the examples we assume that the initial values of the references are  $ff$ . We shall omit the superscript  $W$  in  $(t, wr_{\varrho, v}^{W, I})$  whenever  $W = \emptyset$ , and similarly for  $I$ .

### 4.1 Simple Relaxations

Let us start with the most common relaxation, the one of the  $\mathbf{W} \rightarrow \mathbf{R}$  order [2], supported by simple write buffering as in TSO machines. That is, we are assuming that a write  $(t, wr_{p, v}^{W, I})$  does not have precedence over a read  $(t, rd_{q, \iota})$  if  $p \neq q$ . The litmus test here is the thread system  $T$  of the example (1) given in the Introduction (with an obvious thread names assignment). If we let

$$\sigma = (t_0, wr_{p, tt}) \cdot (t_0, rd_{q, \iota_0}) \cdot (t_1, wr_{q, tt}) \cdot (t_1, rd_{p, \iota_1})$$

we have

$$(S, \varepsilon, T) \xrightarrow[\tau, \mathcal{W}]{*} (S, \sigma, (t_0, r_0 := \iota_0) \parallel (t_1, r_1 := \iota_1))$$

It is then easy to see that, given that the order  $\mathbf{W} \rightarrow \mathbf{R}$  may be relaxed, we have

$$(S, \varepsilon, T) \xrightarrow[\tau, \mathcal{W}]{*} (S, \sigma', (t_0, r_0 := ff) \parallel (t_1, r_1 := ff))$$

where  $\sigma' = (t_0, wr_{p,tt}) \cdot (t_1, wr_{q,tt})$ . These write operations can now be executed, and we reach a final state  $(S', \varepsilon, T')$  where  $S'(p) = tt = S'(q)$  and  $S'(r_0) = ff = S'(r_1)$ .

To restore *SC* behavior in a relaxed memory model, the language must offer appropriate synchronization means. Most often these are *barriers*, that disallow some relaxations, when inserted between memory operations. For instance, to forbid the  $\mathbf{W} \rightarrow \mathbf{R}$  relaxation, a natural barrier to use is  $\langle wr \rangle$  (write/read), which cannot overtake a write, and cannot be overtaken by a read from the same thread. In our framework, the semantics of barriers are specified by the commutability predicate: they have no other effect than preventing some reorderings. In the case of  $\langle wr \rangle$ , we require that the commutability predicate satisfies  $(A\mathcal{P}_{\langle wr \rangle})$  for a precedence relation  $\mathcal{P}_{\langle wr \rangle}$  such that

$$(t, wr_{\varrho, v}^{W, I}) \mathcal{P}_{\langle wr \rangle} (t, \langle wr \rangle) \mathcal{P}_{\langle wr \rangle} (t, rd_{p, \iota})$$

(We do not have to specify that  $\langle wr \rangle$  has precedence over  $\overline{rd}$ , because, due to the conditions in *R2*, a read mark is never preceded by a read barrier in the temporary store.) This is a *local* barrier since it blocks only operations from the thread that issued it. Then for restoring an *SC* behavior to the example we are discussing, it is enough to insert this barrier in both threads:

$$\begin{array}{l} p := tt; \quad q := tt; \\ \langle wr \rangle; \quad \parallel \quad \langle wr \rangle; \\ r_0 := !q \quad r_1 := !p \end{array}$$

The threads will issue  $\langle wr \rangle$  before the reads  $rd_{q, \iota_0}$  and  $rd_{p, \iota_1}$ . Given the precedence relations we just assumed as a semantics for  $\langle wr \rangle$ , these reads cannot proceed until the barrier has disappeared from the temporary store. The rule *R8* requires, for a barrier to vanish, that it may be commuted with the previously issued operations. Then in the example above, this can only happen for  $\langle wr \rangle$  once the writes  $wr_{p, tt}$  and  $wr_{q, tt}$  have been globally performed.

We can deal in a similar way with the relaxation of the order  $\mathbf{W} \rightarrow \mathbf{W}$ , which when added to the previous relaxation characterizes the *PSO* memory model. And similarly with  $\mathbf{R} \rightarrow \mathbf{R}$  and  $\mathbf{R} \rightarrow \mathbf{W}$  which are sufficient to characterize the *RMO* model as described in [2]. In each case a corresponding local barrier,  $\langle ww \rangle$ ,  $\langle rr \rangle$  or  $\langle rw \rangle$  can be used to restore sequential consistency.

## 4.2 Early Reads and Writes

In this subsection, and the following one, we are concerned with architectures relaxing the atomicity of writes. There are several examples to illustrate the write early rule *R5*, in combination with *R2*, to show the ability for a thread to “**read-own-write-early**” or “**read-others'-write-early**”, according to the terminology of [2], that is the ability for a thread to read a write that has been previously issued, either by the thread itself or by a foreign thread, before the write updates the shared memory. An example of the first, which holds in *TSO* models, is as follows:

$$\begin{array}{l} p := tt; \quad q := tt; \\ r_0 := !p; (tt) \parallel r_2 := !q; (tt) \\ r_1 := !q (ff) \quad r_3 := !p (ff) \end{array}$$

where the unexpected outcome is indicated by the annotations ( $tt$ ) and ( $ff$ ) associated with the assignments. Let us assume that the write grain  $\mathcal{W}$  contains two sets  $W_0$  and  $W_1$  such that  $t_0 \in W_0$  and  $t_1 \in W_1$ . Then it is easy to see that from this thread system we can, using the write early rule  $R4$ , reach a configuration where the temporary store is  $\sigma_0 \cdot \sigma_1$  where

$$\begin{aligned}\sigma_0 &= (t_0, \text{wr}_{p,tt}^{W_0}) \cdot (t_0, \text{rd}_{p,\iota_0}) \cdot (t_0, \text{wr}_{r_0,\iota_0}) \cdot (t_0, \text{rd}_{q,\iota_1}) \\ \sigma_1 &= (t_1, \text{wr}_{q,tt}^{W_1}) \cdot (t_1, \text{rd}_{q,\iota_2}) \cdot (t_1, \text{wr}_{r_2,\iota_2}) \cdot (t_1, \text{rd}_{p,\iota_3})\end{aligned}$$

Then by  $R2$  both  $\iota_0$  and  $\iota_2$  can take the value  $tt$ , whereas, given that the order  $\mathbf{W} \rightarrow \mathbf{R}$  is relaxed (and that a read mark does not have precedence over a read), both  $\iota_1$  and  $\iota_3$  take the value  $ff$  from the shared store, before it is updated by performing the writes  $\text{wr}_{p,tt}^{W_0}$  and  $\text{wr}_{q,tt}^{W_1}$ .

As regards the **read-others-write-early** ability, the best known litmus test is IRIW (Independent Reads of Independent Writes):

$$p := tt \parallel q := tt \parallel \begin{array}{l} r_0 := !p; (tt) \\ r_1 := !q (ff) \end{array} \parallel \begin{array}{l} r_2 := !q; (tt) \\ r_3 := !p (ff) \end{array}$$

In our framework, this example is accounted for in the following way. Assume that  $\mathcal{W}$  contains two sets  $W_0$  and  $W_1$  such that  $\{t_0, t_2\} \subseteq W_0$  and  $\{t_1, t_3\} \subseteq W_1$ , with  $t_3 \notin W_0$  and  $t_2 \notin W_1$ . Then we have, using  $R5$  twice:

$$(S, \varepsilon, T) \xrightarrow[\uparrow, \mathcal{W}]{*} (S, (t_0, \text{wr}_{p,tt}^{W_0, \emptyset}) \cdot (t_2, \text{rd}_{p,\iota_0}) \cdot (t_1, \text{wr}_{q,tt}^{W_1, \emptyset}) \cdot (t_3, \text{rd}_{q,\iota_2}), T')$$

Now since the write of  $p$  is made visible to thread  $t_2$ , the identifier  $\iota_0$  can take the value  $tt$ , and similarly  $\iota_2$  takes the value  $tt$ , by the rule  $R2$ . Since the writes from  $t_0$  and  $t_1$  are not visible from  $t_3$  and  $t_2$  respectively, these threads may read the value  $ff$  from the shared memory for both  $q$  and  $p$ . One finally reaches a state where  $S'(r_0) = tt = S'(r_2)$  whereas  $S'(r_1) = ff = S'(r_3)$ . Notice that in this computation we never have to “commute” operations (the precedence relation could be anything here), that is, this computation proceeds in program order, and therefore inserting local barriers in  $t_2$  and  $t_3$  would not influence it. Similar examples that are discussed in [6, 14], such as WRC, RWC and CC, can be explained in the same way. This is the case for instance of WRC (Write-to-Read Causality) – without fence since, as with IRIW, we follow the program order here:

$$p := tt \parallel \begin{array}{l} r_0 := !p; (tt) \\ q := tt \end{array} \parallel \begin{array}{l} r_1 := !q; (tt) \\ r_2 := !p (ff) \end{array}$$

Here the write ( $p := tt$ ) is issued, and, with some appropriate assumption about the write grain, made visible to the second thread (but not to the third), which will then assign the value  $tt$  to  $r_0$ . Then the write ( $q := tt$ ) is globally performed, and, before the operation  $\text{wr}_{p,tt}^{W,I}$  reaches the store, the third thread is executed, reading the values  $tt$  for  $q$  in ( $r_1 := !q$ ) and  $ff$  for  $p$  in ( $r_2 := !p$ ). That is, the outcome  $S'(r_0) = tt = S'(r_1)$  and  $S'(r_2) = ff$  is allowed.

### 4.3 Global Barriers

In a model that enables the read-others’-write-early capability, one needs in the language some barrier having a *global* effect on writes, that is, a barrier that is prevented from vanishing by writes from foreign threads. We shall use here the case of PowerPC, as described by [14], to exemplify the framework. Indeed, the PowerPC architecture offers such a strong sync barrier, which imposes the program order to

be preserved between any pair of (local) reads and writes. This means that it enjoys the same precedence relations as  $\langle wr \rangle$ ,  $\langle ww \rangle$ ,  $\langle rr \rangle$  and  $\langle rw \rangle$ . The global effect of `sync` is the one suggested above: `sync` maintains the order between two writes, the first one being a visible write from a foreign thread, and the second being a local write. Then to specify the semantics of this barrier we just have to add the following:

$$t' \in W \Rightarrow (t, wr_{\rho,v}^{W,I}) \mathcal{P}_{\text{sync}}(t', \text{sync})$$

The PowerPC architecture also provides an `lwsync` barrier, which is weaker than `sync`. First, this is a  $\langle ww \rangle$ ,  $\langle rw \rangle$  and  $\langle rr \rangle$  barrier, but it does not order the pairs of writes and reads, to preserve some TSO optimizations. Therefore, we cannot define the semantics of `lwsync` by means of a binary precedence relation, as we did up to now. Nevertheless, the following precedences are part of the semantics of `lwsync` in our framework:

$$\begin{aligned} (t, \bar{rd}_\iota) \mathcal{P}_{\text{lw}}(t, \text{lwsync}) \quad \& \quad (t, rd_{\rho,t}) \mathcal{P}_{\text{lw}}(t, \text{lwsync}) \mathcal{P}_{\text{lw}}(t, wr_{\rho',v}^{W,I}) \\ t = t' \text{ or } t' \in W \Rightarrow (t, wr_{\rho,v}^{W,I}) \mathcal{P}_{\text{lw}}(t', \text{lwsync}) \end{aligned}$$

Next, we have to say that `lwsync` is a  $\langle rr \rangle$  barrier, even though it does not have precedence over reads. Then we assume that the commutability predicate satisfies the following:

$$\left. \begin{aligned} \sigma = \sigma_0 \cdot (t, \text{lwsync}) \cdot \sigma_1 \quad \& \\ \sigma_0 = \delta_0 \cdot (t, rd_{\rho,v}) \cdot \delta_1 \text{ or } \sigma_0 = \delta_0 \cdot (t, \bar{rd}_\iota) \cdot \delta_1 \end{aligned} \right\} \Rightarrow \neg(\sigma \overset{\text{rw}}{\vdash} (t, rd_{\rho,v'}))$$

This completes the definition of the semantics of `lwsync`. Let us see two examples. If we insert global barriers into the IRIW configuration, as follows:

$$\begin{array}{ccc} r_0 := !p; (tt) & r_2 := !q; (tt) \\ p := tt \parallel q := tt \parallel \text{lwsync}; & \parallel \text{sync}; \\ r_1 := !q (ff) & r_3 := !p (ff) \end{array}$$

then the unexpected outcome is still not prevented to occur. This is obtained as follows: the operation of the second thread ( $t_1$ ) is issued, and then the ones of  $t_3$ ,  $t_0$  and  $t_2$ , in that order. Then the visibility of  $(t_0, wr_{p,tt})$  is made global, and therefore  $t_2$  can read the value  $tt$  for  $p$ . Since the write from  $t_0$  is allowed to be performed immediately, the read mark left when performing  $rd_{p,t_0}$  may disappear. The `lwsync` from  $t_2$  is still prevented to vanish by the write from  $t_0$ , but it no longer blocks the second read of  $t_2$ .

In the case of the WRC litmus test [6], inserting `lwsync` barriers prevents the unexpected outcome showed in

$$\begin{array}{ccc} r_0 := !p; (tt) & r_1 := !q; (tt) \\ p := tt \parallel \text{lwsync}; & \parallel \text{lwsync}; \\ q := tt & r_2 := !p (ff) \end{array}$$

to occur. Similarly, inserting `sync` barriers in the third and fourth threads of the IRIW example restores an SC behavior. To see this, we have to explore all the possible behaviors, and this is where our software tool is useful.

## 5 The Simulator

The set of configurations that may be reached by running a program in the relaxed semantics can be fairly large, and it is sometimes difficult, and error prone, to find a path to some (un)expected final

state, or to convince oneself that such an outcome is actually forbidden, that is, unreachable. Then, to experiment with our framework, we found it useful to design and implement a simulator that allows us to exhaustively explore all the possible relaxed behaviors of (simple) programs. As usual, we have to face a state explosion problem, which is much worse than with the standard interleaving semantics.

Our simulator is written in JAVA. Its main function `step` computes all the configurations reachable in one step from a given configuration. A brute force simulator would then recursively use the `step` function, in a depth first manner, in order to compute reachable configurations that have an empty temporary store and a terminated thread pool, where all the thread expressions are values. This methodology does not consume much memory space, being basically proportional to the `log` of the number of reachable states or, similarly, to the depth of the tree induced by the `step` function. However, the number of configurations in this tree grows very fast with the size of the expression to analyse. For instance, with the example (1) given in the Introduction, this brute force strategy has been aborted after generating more than  $20 \times 10^{10}$  configurations and after half a day of computing, even if it is obvious that only four *different* final configurations may be reached. Therefore, a first improvement is to transform the tree traversal by a dag construction merging all the same configurations. Less configurations will be constructed and analyzed (only 60588 for the example), but all these configurations must be simultaneously in memory.

Several other optimizations have been used. In order to reduce the search space, in the simulator we use a refined rule *R5* where the visibility set  $W'$  is supposed to be either *Tid* or a subset of  $\text{live}(T) \cup \text{rdt}(\sigma_1)$  where the sets  $\text{live}(T)$  and  $\text{rdt}(\sigma)$  of thread identifiers are defined as follows:

$$\begin{aligned} \text{live}(\emptyset) &= \emptyset & \text{rdt}(\varepsilon) &= \emptyset \\ \text{live}((t, e) \| T) &= \text{live}(T) \cup \{t \mid e \notin \text{Val}\} & \text{rdt}((t, \xi) \cdot \sigma) &= \text{rdt}(\sigma) \cup \{t \mid \exists \varrho, \iota. \xi = \text{rd}_{\varrho, \iota}\} \end{aligned}$$

We have not presented this formulation in Figure 3 only because it is conceptually a bit more obscure. With this optimization, in our example, the number of configurations falls down from 60588 to 51068. A more dramatic optimization is obtained by introducing a distinction between “registers,” that are local to some thread, and shared references. As suggested above, the registers are denoted  $r_i$  in the examples. Indeed, these registers are not concerned by early reads from foreign threads, and therefore applications of the rule *R5* to them may be drastically restricted. In this way, the number of generated configurations in the case of example (1) decreases from 51068 to 13356 for instance. Furthermore, one may observe that, since removing an operation from a temporary store  $\sigma$  never depends on what follows this operation in  $\sigma$ , the strategy that consists in applying first the rules of Figure 2 for evaluating the threads before attempting anything else (that is, applying a rule from Figure 3) will never miss any final configuration. This allows us to generate only 2814 configurations in the case of example (1) for instance.

However, the optimized search strategy outlined above still fails in exploring exhaustively some complex litmus tests. In such cases, we make a tradeoff between time and space: for each temporary store that can be reached by applying the rules of Figure 2 as far as possible, we generate the reachable final configurations, but we do not share this state space among the various possible temporary stores. For instance, still regarding the example (1), there are 20 possible “maximal” temporary stores, and running independently the simulator in each case generates an average number of 500 configurations, so that the total of number of generated configurations following this simulation method raises up to 10280. Nevertheless this allowed us to successfully explore a large number of litmus tests, and in particular all the ones presented by Sarkar & al. [14] in their web files. We report upon this in the full version of the paper. Our simulator is available on the web page <http://www-sop.inria.fr/index/MemoryModels/>.

## 6 Conclusion

We have introduced a new, operational way to formalize the relaxed semantics of concurrent programs. Our model is flexible enough to account for a wide variety of weak behaviors, and in particular the odd ones occurring in a memory model that does not preserve the atomicity of writes. To our view, our model is also simple enough to be easily understood by the implementer and the programmer, and precise enough to be used in the formal analysis of programs.

## References

- [1] S. ADVE, H.-J. BOEHM, *Memory models: a case for rethinking parallel languages and hardware*, CACM Vol. 53 No. 8 (2010) 90-101. doi:10.1145/1787234.1787255
- [2] S. ADVE, K. GHARACHORLOO, *Shared memory consistency models: a tutorial*, IEEE Computer Vol. 29 No. 12 (1996) 66-76. doi:10.1109/2.546611
- [3] S. ADVE, M.D. HILL, *Weak ordering – A new definition*, ISCA'90 (1990) 2-14. doi:10.1145/325096.325100
- [4] M. ATIG, A. BOUAJJANI, S. BURCKHARDT, M. MUSUVATHI, *What's Decidable about Weak Memory Models?*, ESOP'12 (2012) 26-46. doi:10.1007/978-3-642-28869-2\_2
- [5] M. BATTY, S. OWENS, S. SARKAR, P. SEWELL, T. WEBER, *Mathematizing C++ concurrency*, POPL'11 (2011) 55-66. doi:10.1145/1925844.1926394
- [6] H.-J. BOEHM, S. ADVE, *Foundations of the C++ concurrency model*, PLDI'08 (2008) 68-78. doi:10.1145/1375581.1375591
- [7] G. BOUDOL, G. PETRI, *Relaxed memory models: an operational approach*, POPL'09 (2009) 392-403. doi:10.1145/1480881.1480930
- [8] G. BOUDOL, G. PETRI, *A theory of speculative computations*, ESOP'10, Lecture Notes in Comput. Sci. 6012 (2010) 165-184. doi:10.1007/978-3-642-11957-6\_10
- [9] S. BURCKHARDT, M. MUSUVATHI, V. SINGH, *Verifying local transformations on relaxed memory models*, CC'10, Lecture Notes in Comput. Sci. 6011 (2010) 104-123. doi:10.1007/978-3-642-11970-5\_7
- [10] K. GHARACHORLOO, D. LENOSKI, J. LAUDON, P. GIBBONS, A. GUPTA, J. HENNESSY, *Memory consistency and event ordering in scalable shared-memory multiprocessors*, ACM SIGARCH Computer Architecture News Vol. 18 No. 3a (1990) 15-26. doi:10.1145/325164.325102
- [11] L. LAMPORT, *How to make a multiprocessor computer that correctly executes multiprocess programs*, IEEE Trans. on Computers Vol. 28 No. 9 (1979) 690-691. doi:10.1109/TC.1979.1675439
- [12] J. MANSON, W. PUGH, S.A. ADVE, *The Java memory model*, POPL'05 (2005) 378-391. doi:10.1145/1040305.1040336
- [13] S. PARK, D.L. DILL, *An executable specification and verifier for Relaxed Memory Order*, IEEE Trans. on Computers, Vol. 48, No. 2 (1999) 227-235. doi:10.1109/12.752664
- [14] S. SARKAR, P. SEWELL, J. ALGLAVE, L. MARANGET, D. WILLIAMS, *Understanding POWER multiprocessors*, PLDI'11 (2011) 175-186. doi:10.1145/1993498.1993520
- [15] P. SEWELL, S. SARKAR, S. OWENS, F. ZAPPA NARDELLI, M. O. MYREEN, *x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors*, CACM Vol. 53 No. 7 (2010) 89-97. doi:10.1145/1785414.1785443