Relaxed Semantics of Concurrent Programming Languages

Gérard Boudol, Bernard Serpette
INRIA Sophia Antipolis
{Gerard.Boudol,Bernard.Serpette}@inria.fr

Gustavo Petri
Purdue University
gpetri@gmail.com

Abstract

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. We illustrate our approach by way of examples, and discuss some extensions. We have build 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++ (using a Pthread library). 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, [22]). This is also the semantics assumed by most verification methods. However, it is well-known [3] that this semantics is *not* the one we observe when running concurrent programs in optimizing execution environments, i.e. compilers and harware architectures, which are designed to run sequential programs as fast as possible. For instance, let us consider the program

$$\begin{array}{l} p := tt; \\ r_0 := !\,q \end{array} \quad \| \quad \begin{array}{l} q := tt; \\ r_1 := !\,p \end{array} \tag{1}$$

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 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 of a TSO machine [3] 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 [3]), 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 [4, 18], and has been widely advocated since then (see [2, 6, 28]). 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. (A way that has been explored [26, 29] is to associate exceptions with races, but in this approach optimizing code transformations is restricted to specific regions, and concurrent programming is constrained to observe a strict discipline, to avoid too many exceptions.) 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) [28] is still not completely fixed [32]. 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 [31]. 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, 33], which, besides being "*widely accessible to working programmers*" [33], allows us to use standard techniques to analyse and verify programs, proving properties such as the DRF guarantee [7] for instance. In [7, 33], 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's 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 [3]. 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 [22] 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 [3]. Our model then easily explains, in operational terms, the behavior of a series "litmus tests," such as IRIW, WRC, RWC and CC discussed in [6] for instance, and the tests from [31], 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 [31]. 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), and therefore our model 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 build to experiment with our semantics.

Compared to other formalizations of relaxed semantics, our model is truly operational. This means in particular that the temporary store is not considered modulo rewriting, as in [10]. Also, contrarily to [8] for instance, our model preserves a notion of *causality*: a read can only return a value that is present in the shared memory, or that is previoulsy written by some thread. Our notion of a temporary store is quite similar to the "reorder box" of [30], but formulated in the standard framework of programming language semantics. Regarding the relaxation of write atomicity, the only work we know that proposes an operational formulation of this feature is [31]. We think however that our formalization, by means of write visibility, is much simpler than the one suggested in this paper. 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.

2. The Core Language

Our language is a higher-order, imperative and concurrent language à la ML, that is a call-by-value λ-calculus extended with constructs to deal with a mutable store, to dynamically spawn threads and to synchronize their actions. (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 [16]. 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 $\mathcal{V}ar$ of variables, ranged over by $x,\ y,\ z\ldots$, the syntax is as follows:

$$
\begin{array}{rcll}
v & ::= & x \mid \lambda x e \mid tt \mid ff \mid () & \textit{values} \\
b & \in & \mathcal{B}ar & \textit{barriers} \\
e \in \mathcal{L} & ::= & v \mid (ve) \mid (\mathsf{if}\ v\ \mathsf{then}\ e_0\ \mathsf{else}\ e_1) & \textit{expressions} \\
 & \mid & (\mathsf{ref}\ v) \mid (!\,v) \mid (v_0 := v_1) & \\
 & \mid & (\mathsf{thread}\ e) \mid (\mathsf{join}\ v) & \\
 & \mid & (\mathsf{with}\ v\ \mathsf{do}\ e) \mid b &
\end{array}
$$

As usual, the variable x is bound in an expression λxe, and we consider expressions up to α-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 $(\mathsf{let}\ x = e_0\ \mathsf{in}\ e_1)$ 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 $(\mathsf{let}\ f = e_0\ \mathsf{in}\ (f e_1))$, or $(v := e)$ into $(\mathsf{let}\ x = e\ \mathsf{in}\ (v := x))$.

The barrier constructs are "no-ops" in the abstract (interleaving) semantics of the language. Such synchronization constructs are often considered low-level, and inserted at compile time [13, 21, 23, 24, 34]. 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 $\mathcal{B}ar$ here, so the language should actually be $\mathcal{L}(\mathcal{B}ar)$, but in the following we shall give some examples of useful barriers, and see how to formalize their semantics. The thread spawning construct $(\mathsf{thread}\, e)$, as well as the $(\mathsf{join}\, v)$ construct, are standard. In $(\mathsf{with}\ v\ \mathsf{do}\ e)$ the value v is intended to denote a reference. The intuitive meaning is that this expression acquires, if possible, the pointer denoted by v for exclusive use in e, and releases it upon termination. That is, in our language synchronization is only concerned with memory operations, and in particular acquiring/releasing pointers, not locks. We must point out that our semantical framework is not tied to this particular form of locking, and that other choices could be supported as well. For instance, the more sophisticated "load-reserve/store conditional" instruction of PowerPC should be amenable to a similar treatment.

As usual, to formalize the operational semantics of the language, we have to extend it, introducing some run-time values. We assume given a set $\mathcal{R}ef$ of *references*, ranged over by p, $q\ldots$ and a set $\mathcal{T}id$ of *thread indentifiers*, ranged over by t. These are the values returned by reference and thread creation respectively. Then the language is extended as follows:

$$
\begin{array}{rcll}
v \in \mathcal{V}al & ::= & \cdots \mid p \mid t & \textit{run-time values} \\
e \in \widehat{\mathcal{L}} & ::= & v \mid \cdots & \\
\bar{e} \in \mathcal{E}xpr & ::= & e \mid (v\bar{e}) \mid (\bar{e}\backslash p) & \textit{run-time expressions}
\end{array}
$$

The expression $(\bar{e}\backslash p)$ represents the program $\bar{e}$ running with an exclusive access to the pointer p, which has been previously acquired. If we were focusing on low-level memory models, we should distinguish *registers* form references. In this case, a write $r := !\,p$ where r is such a register is a *load* of p, whereas a *store* is just a write $p := v$ where p is a reference. This distinction is then useful in discussing reorderings and barriers, since a load is usually not

$$
\begin{array}{rcll}
(S,\mathcal{O},(t,\mathbf{E}[(\lambda x e v)])\parallel T) & \to & (S,\mathcal{O},(t,\mathbf{E}[\{x\mapsto v\}e])\parallel T) & \\
(S,\mathcal{O},(t,\mathbf{E}[(\mathsf{if}\ tt\ \mathsf{then}\ e_0\ \mathsf{else}\ e_1)])\parallel T) & \to & (S,\mathcal{O},(t,\mathbf{E}[e_0])\parallel T) & \\
(S,\mathcal{O},(t,\mathbf{E}[(\mathsf{if}\ ff\ \mathsf{then}\ e_0\ \mathsf{else}\ e_1)])\parallel T) & \to & (S,\mathcal{O},(t,\mathbf{E}[e_1])\parallel T) & \\
(S,\mathcal{O},(t,\mathbf{E}[(\mathsf{ref}\ v)])\parallel T) & \to & (S\cup\{p\mapsto v\},\mathcal{O},(t,\mathbf{E}[p])\parallel T) & \text{if}\quad p\in\mathcal{R}ef-\mathsf{dom}(S)\\
(S,\mathcal{O},(t,\mathbf{E}[(!\,p)])\parallel T) & \to & (S,\mathcal{O},(t,\mathbf{E}[v])\parallel T) & \text{if}\quad S(p)=v\\
(S,\mathcal{O},(t,\mathbf{E}[(p:=v)])\parallel T) & \to & (S[p:=v],\mathcal{O},(t,\mathbf{E}[()])\parallel T) & \\
(S,\mathcal{O},(t,\mathbf{E}[(\mathsf{thread}\ e)])\parallel T) & \to & (S,\mathcal{O},(t,\mathbf{E}[t'])\parallel(t',e)\parallel T) & \text{if}\quad t'\notin\mathsf{dom}(T)\\
(S,\mathcal{O},(t,\mathbf{E}[(\mathsf{join}\ t')])\parallel T) & \to & (S,\mathcal{O},(t,\mathbf{E}[()])\parallel T) & \text{if}\quad T(t')\in\mathcal{V}al\\
(S,\mathcal{O},(t,\mathbf{E}[(\mathsf{with}\ p\ \mathsf{do}\ e)])\parallel T) & \to & (S,\mathcal{O},(t,\mathbf{E}[e])\parallel T) & \text{if}\quad p\in\mathsf{dom}(\mathcal{O})\ \&\ \mathcal{O}(p)=t\\
(S,\mathcal{O},(t,\mathbf{E}[(\mathsf{with}\ p\ \mathsf{do}\ e)])\parallel T) & \to & (S,\mathcal{O}\cup\{p\mapsto t\}),(t,\mathbf{E}[(e\backslash p)])\parallel T) & \text{if}\quad p\notin\mathsf{dom}(\mathcal{O})\\
(S,\mathcal{O},(t,\mathbf{E}[(v\backslash p)])\parallel T) & \to & (S,\mathcal{O}\backslash p,(t,\mathbf{E}[v])\parallel T) & \\
(S,\mathcal{O},(t,\mathbf{E}[b])\parallel T) & \to & (S,\mathcal{O},(t,\mathbf{E}[()])\parallel T) &
\end{array}
$$

Figure 1: Reference Operational Semantics

considered as a write. 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, possibly involving references and thread identifiers. Notice however that these expressions $e\in\widehat{\mathcal{L}}$ do not contain the construct $(\_\backslash p)$. A step in the semantics consists in evaluating a redex inside an *evaluation context*. The syntax of the latter is as follows:

$$
\begin{array}{rcll}
\mathbf{E} & ::= & [\,] \mid \mathbf{E}[\mathbf{F}] & \textit{evaluation contexts}\\
\mathbf{F} & ::= & (v\,[\,]) \mid ([\,]\backslash p) & \textit{frames}
\end{array}
$$

As usual, we denote by $\mathbf{E}[\bar{e}]$ the run-time expression obtained by filling the hole in $\mathbf{E}$ by $\bar{e}$. This is defined by

$$
\begin{array}{rcl}
[\,][\bar{e}] & = & \bar{e}\\
\mathbf{E}[\mathbf{F}][\bar{e}] & = & \mathbf{E}[\mathbf{F}[\bar{e}]] \quad \text{where} \quad (v\,[\,])[\bar{e}] = (v\bar{e})\\
 & & \qquad\qquad\qquad\quad ([\,]\backslash p)[\bar{e}] = (\bar{e}\backslash p)
\end{array}
$$

The semantics is specified as small step transitions $C\to C'$ between configurations C, C' of the form $(S,\mathcal{O},T)$ where S, $\mathcal{O}$ and T are respectively the *store*, the *ownership* mapping and the *thread system*. The store S, also called here the *memory*, is a mapping from a finite set $\mathsf{dom}(S)$ of references to values. The thread system T is a mapping from a finite set $\mathsf{dom}(T)$ of thread identifiers, subset of $\mathcal{T}id$, to run-time expressions, and $\mathcal{O}$ is a mapping from a finite set $\mathsf{dom}(\mathcal{O})$ of references, contained in $\mathsf{dom}(S)$, into $\mathsf{dom}(T)$. If $\mathsf{dom}(T)=\{t_1,\ldots,t_n\}$ and $T(t_i)=\bar{e}_i$ we also write T as

$$(t_1,\bar{e}_1)\parallel\cdots\parallel(t_n,\bar{e}_n)$$

We say that a pointer p such that $p\in\mathsf{dom}(\mathcal{O})$ and $\mathcal{O}(p)=t$ is *owned* by thread t.

The reference operational semantics is given in Figure 1, where $S[p:=v]$, in the rule for reducing $(p:=v)$, means updating p in the store S with value v, and, in the rule for reducing $(v\backslash p)$, $\mathcal{O}\backslash p$ means $\mathcal{O}$ restricted to $\mathsf{dom}(\mathcal{O})-\{p\}$. One can see from the semantics of $(\mathsf{with}\ p\ \mathsf{do}\ e)$ that a pointer p can be temporarily acquired as a private reference by a thread, provided that it is not already owned by a different thread. Then a thread t is blocked when it tries to acquire, by executing $(\mathsf{with}\ p\ \mathsf{do}\ e)$, an exclusive access to a reference p that is currently private to another thread – i.e. $p\in\mathsf{dom}(\mathcal{O})$ with $t\neq\mathcal{O}(p)$. That is, reducing $(\mathsf{with}\ p\ \mathsf{do}\ e)$ is a synchronization operation. Obviously, for this construct to be really useful, the programmer is supposed to adhere to the intended discipline, that is, when a reference is used within a locking construct in a piece of code, it should be used in this way in all the threads of the program. Checking that such a discipline is enforced has been studied by many authors (see for instance [1, 9, 14, 19]), and this is a topic that we don't have to address here.

3. Relaxed Computations

3.1 Preliminary Definitions

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

$$RC\underset{\mathcal{M}}{\longrightarrow}RC'$$

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. These names are similar to the "placeholder variables" of [15] and the "prophecy variables" of [27]. The set $\mathcal{I}dent$ of identifiers is assumed to be disjoint from $\mathcal{V}ar\cup\mathcal{R}ef$, and is ranged over by ι. We shall use ϱ to range over $\mathcal{R}ef\cup\mathcal{I}dent$. The identifiers $\iota\in\mathcal{I}dent$ are *values* in the extended language, still denoted by v, but notice that $\mathcal{V}al$ denotes the set of (not relaxed) values, that do not contain any identifier ι. 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 ι in an expression e results in a valid expression, denoted $\{\iota\mapsto v\}e$.

Our next technical ingredient is the set $\mathcal{M}op(\mathcal{L})$ 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 $\mathcal{M}op(\mathcal{L})$ of memory operations comprises the barriers $b\in\mathcal{B}ar$, the *acquire* operations $\overset{\frown}{p}$ of acquiring a reference p, and the corresponding *release* operations $\overset{\smile}{p}$, the *read* and *write* operations, respectively denoted $\mathsf{rd}_{\varrho,\iota}$ and $\mathsf{wr}_{\varrho,v}^{W,I}$ where $\varrho\in\mathcal{R}ef\cup\mathcal{I}dent$, $\iota\in\mathcal{I}dent$, $W\subseteq\mathcal{T}id$ is a set of thread names, and $I\subseteq\mathcal{I}dent$ is a set of identifiers. We call the set W in $\mathsf{wr}_{\varrho,v}^{W,I}$ the *visibility* of the write, whereas I is the set of identifiers that have been bound to the written value v when performing a read (we comment on these components below). We also include in $\mathcal{M}op(\mathcal{L})$ the *spawning* and *joining* operations, respectively written $\mathsf{spw}_{t,e}$ and join_t. Finally, we introduce operations of the form $\overline{\mathsf{rd}}_\iota$ that we call a *read mark*, meaning that a read has occurred, where ι serves as identifying the corresponding write. That is, the syntax of memory operations is as follows:

$$
\begin{array}{rcll}
\xi\in\mathcal{M}op(\mathcal{L}) & ::= & \mathsf{rd}_{\varrho,\iota}\mid\overline{\mathsf{rd}}_\iota\mid\mathsf{wr}_{\varrho,v}^{W,I}\mid\zeta & \textit{memory operations}\\
\zeta\in\mathcal{S}ync & ::= & \mathsf{spw}_{t,e}\mid\mathsf{join}_t & \textit{synchronization operations}\\
 & & \mid\quad\overset{\frown}{p}\mid\overset{\smile}{p}\mid b &
\end{array}
$$

We can now define a *relaxed configuration* RC as a tuple

$$RC=(S,\mathcal{O},\sigma,T)$$

$$
\begin{array}{rcll}
(S,\mathcal{O},\sigma,(t,\mathbf{E}[(\lambda x ev)])\parallel T) & \hookrightarrow & (S,\mathcal{O},\sigma,(t,\mathbf{E}[\{x\mapsto v\}e])\parallel T) & \\
(S,\mathcal{O},\sigma,(t,\mathbf{E}[(\mathsf{if}\ tt\ \mathsf{then}\ e_0\ \mathsf{else}\ e_1)])\parallel T) & \hookrightarrow & (S,\mathcal{O},\sigma,(t,\mathbf{E}[e_0])\parallel T) & \\
(S,\mathcal{O},\sigma,(t,\mathbf{E}[(\mathsf{if}\ ff\ \mathsf{then}\ e_0\ \mathsf{else}\ e_1)])\parallel T) & \hookrightarrow & (S,\mathcal{O},\sigma,(t,\mathbf{E}[e_1])\parallel T) & \\
(S,\mathcal{O},\sigma,(t,\mathbf{E}[(\mathsf{ref}\ v)])\parallel T) & \hookrightarrow & (S,\mathcal{O},\sigma\cdot(t,\mathsf{wr}^{\emptyset,\emptyset}_{p,v}),(t,\mathbf{E}[p])\parallel T) & \text{where } p \text{ is fresh} \\
(S,\mathcal{O},\sigma,(t,\mathbf{E}[(!\,\varrho)])\parallel T) & \hookrightarrow & (S,\mathcal{O},\sigma\cdot(t,\mathsf{rd}_{\varrho,\iota}),(t,\mathbf{E}[\iota])\parallel T) & \text{where } \iota \text{ is fresh} \\
(S,\mathcal{O},\sigma,(t,\mathbf{E}[(\varrho:=v)])\parallel T) & \hookrightarrow & (S,\mathcal{O},\sigma\cdot(t,\mathsf{wr}^{\emptyset,\emptyset}_{\varrho,v}),(t,\mathbf{E}[()])\parallel T) & \\
(S,\mathcal{O},\sigma,(t,\mathbf{E}[(\mathsf{thread}\ e)])\parallel T) & \hookrightarrow & (S,\mathcal{O},\sigma\cdot(t,\mathsf{spw}_{t',e}),(t,\mathbf{E}[t])\parallel T) & \text{where } t'\notin\mathsf{dom}(T) \\
(S,\mathcal{O},\sigma,(t,\mathbf{E}[(\mathsf{join}\ t')])\parallel T) & \hookrightarrow & (S,\mathcal{O},\sigma\cdot(t,\mathsf{join}_{t'}),(t,\mathbf{E}[()])\parallel T) & \text{if } T(t')\in\mathcal{V}al \\
(S,\mathcal{O},\sigma,(t,\mathbf{E}[(\mathsf{with}\ p\ \mathsf{do}\ e)])\parallel T) & \hookrightarrow & (S,\mathcal{O},\sigma,(t,\mathbf{E}[e])\parallel T) & \text{if } p\in\lceil\mathbf{E}\rceil \\
(S,\mathcal{O},\sigma,(t,\mathbf{E}[(\mathsf{with}\ p\ \mathsf{do}\ e)])\parallel T) & \hookrightarrow & (S,\mathcal{O},\sigma\cdot(t,\overset{\frown}{p}),(t,\mathbf{E}[(e\backslash p)])\parallel T) & \text{otherwise} \\
(S,\mathcal{O},\sigma,(t,\mathbf{E}[(v\backslash p)])\parallel T) & \hookrightarrow & (S,\mathcal{O},\sigma\cdot(t,\overset{\frown}{p}),(t,\mathbf{E}[v])\parallel T) & \\
(S,\mathcal{O},\sigma,(t,\mathbf{E}[b])\parallel T) & \hookrightarrow & (S,\mathcal{O},\sigma\cdot(t,b),(t,\mathbf{E}[()])\parallel T) &
\end{array}
$$

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

where S, $\mathcal{O}$ and T are as above, and σ is a sequence of pairs (t,ξ), where $t\in\mathcal{T}id$ is a thread name and $\xi\in\mathcal{M}op(\mathcal{L})$ a memory operation. The meaning of (t,ξ) in a sequence σ is that ξ is a memory operation issued by thread t. The sequence σ 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 σ. We shall call such a σ a *temporary store*. We denote by $\Sigma_{\mathcal{L}}$ the set $\mathcal{T}id\times\mathcal{M}op(\mathcal{L})$, so that the set of temporary stores is $\Sigma^*_{\mathcal{L}}$, the set of finite sequences over $\Sigma_{\mathcal{L}}$. We denote by ε the empty sequence, and we write $\sigma\cdot\sigma'$ for the concatenation of the two sequences σ and σ'. We say that a relaxed configuration $(S,\mathcal{O},\sigma,T)$ 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 σ are performed. (A similar approach is used in [12].) 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 $\underset{\mathcal{M}}{\longrightarrow}$ into $\hookrightarrow$. In the rules for reducing $(\mathsf{ref}\ v)$ and $(!\,\varrho)$, "p fresh" and "ι fresh" mean that p and ι do not occur in the configuration. Regarding the reduction of $(\mathsf{with}\ p\ \mathsf{do}\ e)$, we denote by $\lceil\mathbf{E}\rceil$ the set of reference names held in the evaluation context $\mathbf{E}$, that is the set of references p such that $([]\backslash p)$ occurs in $\mathbf{E}$.

Notice that the threads are always executed in program order, and that nothing can prevent a reducible expression to be reduced, except for $(\mathsf{join}\ t)$, which is blocking, waiting for the termination of the thread t to join. 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 $(\mathsf{ref}\ v)$, has the same effect, once a new reference name is obtained. Notice that we could also issue, before the $\mathsf{wr}^{\emptyset,\emptyset}_{p,v}$, an operation like $(\mathsf{new}\,p)$, to be performed later. This might be closer to an implementation, but this complicates the framework with no obvious benefit. Reducing $(\mathsf{with}\ p\ \mathsf{do}\ e)$ or $(v\backslash p)$ has a similar effect of appending acquire or release instructions as pending operations. Reducing a dereferencing operation $(!\,\varrho)$ does not immediately return a proper value (read from the store), but creates and returns a fresh identifier $\iota\in\mathcal{I}dent$, which has not yet 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.

A relaxed configuration $(S,\mathcal{O},\sigma,T)$ can also perform actions that originate from the temporary store σ. 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 $\mathcal{M}=(\upharpoonleft,\mathcal{W})$ made of a *commutability* predicate $\upharpoonleft$ and a "*write grain*" $\mathcal{W}$. These two components provide a formalization of the approach of Adve and Gharachroloo in [3], 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 provides semantics for barriers. This first component $\upharpoonleft$ 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 $\sigma\upharpoonleft(t,\xi)$, then the operation ξ from thread t may be globally performed, as if it were the first one, and removed from the temporary store. We read $\sigma\upharpoonleft(t,\xi)$ as: (t,ξ) may overtake σ, or: σ allows (t,ξ) to be performed. We assume, as an axiom satisfied by any memory model, that the first operation in a temporary store is always allowed to execute, that is, for any ξ and t:

$$\varepsilon\upharpoonleft(t,\xi) \qquad \text{(E)}$$

The $\mathcal{W}$ component of a memory model is a set of subsets of $\mathcal{T}id$, comprising the set of the allowed write visibilities. In the relaxed semantics, with each write operation $\mathsf{wr}^{W,I}_{\varrho,v}$ is associated a visibility W, which is a (possibly empty) set of thread identifiers. 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

$$
\begin{array}{lll}
(S,\mathcal{O},\sigma,T) \xrightarrow[\curvearrowleft,\mathcal{W}]{} (S,\mathcal{O},\{\iota\mapsto v\}(\sigma_0\cdot\sigma_1,T)) & & R1\ (\textit{read}) \\
\quad \text{if} \quad \sigma=\sigma_0\cdot(t,\mathsf{rd}_{p,\iota})\cdot\sigma_1 \ \&\ \sigma_0 \curvearrowleft (t,\mathsf{rd}_{p,\iota}) \ \&\ S(p)=v & & \\
(S,\mathcal{O},\sigma,T) \xrightarrow[\curvearrowleft,\mathcal{W}]{} (S,\mathcal{O},\{\iota\mapsto v\}(\sigma_0\cdot(t',\mathsf{wr}_{\varrho,v}^{W,I\cup\{\iota\}})\cdot\sigma_1\cdot(t,\overline{\mathsf{rd}}_\iota)\cdot\sigma_2,T)) & & R2\ (\textit{read early}) \\
\quad \text{if} \quad \sigma=\sigma_0\cdot(t',\mathsf{wr}_{\varrho,v}^{W,I})\cdot\sigma_1\cdot(t,\mathsf{rd}_{\varrho,\iota})\cdot\sigma_2 \ \&\ t\in W \ \& & & \\
\qquad \sigma_1 \curvearrowleft (t,\mathsf{rd}_{\varrho,\iota}) \ \&\ \sigma_0 \curvearrowleft^{\mathcal{S}} (t,\mathsf{rd}_{\varrho,\iota}) & & \\
(S,\mathcal{O},\sigma,T) \xrightarrow[\curvearrowleft,\mathcal{W}]{} (S,\mathcal{O},\sigma_0\cdot\sigma_1,T) & & R3\ (\textit{read}) \\
\quad \text{if} \quad \sigma=\sigma_0\cdot(t,\overline{\mathsf{rd}}_\iota)\cdot\sigma_1 \ \&\ \sigma_0 \curvearrowleft (t,\overline{\mathsf{rd}}_\iota) \text{ or} & & \\
\qquad \sigma_0=\delta_0\cdot(t',\mathsf{wr}_{p,v}^{\mathcal{T}id,I\cup\{\iota\}})\cdot\delta_1 \ \&\ \delta_0 \curvearrowleft (t',\mathsf{wr}_{p,v}^{\mathcal{T}id,I\cup\{\iota\}}) & & \\
(S,\mathcal{O},\sigma,T) \xrightarrow[\curvearrowleft,\mathcal{W}]{} (S[p:=v],\mathcal{O},\sigma_0\cdot\sigma_1,T) & & R4\ (\textit{write}) \\
\quad \text{if} \quad \sigma=\sigma_0\cdot(t,\mathsf{wr}_{p,v}^{W,I})\cdot\sigma_1 \ \&\ \sigma_0 \curvearrowleft (t,\mathsf{wr}_{p,v}^{W,I}) \ \&\ v\in\mathcal{V}al & & \\
(S,\mathcal{O},\sigma,T) \xrightarrow[\curvearrowleft,\mathcal{W}]{} (S,\mathcal{O},\sigma_0\cdot(t,\mathsf{wr}_{\varrho,v}^{W',I})\cdot\sigma_1,T) & & R5\ (\textit{write early}) \\
\quad \text{if} \quad \sigma=\sigma_0\cdot(t,\mathsf{wr}_{\varrho,v}^{W,I})\cdot\sigma_1 \ \&\ t\in W'\in\mathcal{W} \ \&\ W\subset W'\in\mathcal{W} & & \\
(S,\mathcal{O},\sigma,T) \xrightarrow[\curvearrowleft,\mathcal{W}]{} (S,\mathcal{O}\cup\{p\mapsto t\},\sigma_0\cdot\sigma_1,T) & & R6\ (\textit{acquire}) \\
\quad \text{if} \quad \sigma=\sigma_0\cdot(t,\overset{\frown}{p})\cdot\sigma_1 \ \&\ \sigma_0 \curvearrowleft (t,\overset{\frown}{p}) \ \&\ p\notin\mathsf{dom}(\mathcal{O}) & & \\
(S,\mathcal{O},\sigma,T) \xrightarrow[\curvearrowleft,\mathcal{W}]{} (S,\mathcal{O}\backslash p,\sigma_0\cdot\sigma_1,T) & & R7\ (\textit{release}) \\
\quad \text{if} \quad \sigma=\sigma_0\cdot(t,\overset{\frown}{p})\cdot\sigma_1 \ \&\ \sigma_0 \curvearrowleft (t,\overset{\frown}{p}) & & \\
(S,\mathcal{O},\sigma,T) \xrightarrow[\curvearrowleft,\mathcal{W}]{} (S,\mathcal{O},\sigma_0\cdot\sigma_1,T) & & R8\ (\textit{barrier}) \\
\quad \text{if} \quad \sigma=\sigma_0\cdot(t,b)\cdot\sigma_1 \ \&\ \sigma_0 \curvearrowleft (t,b) & & \\
(S,\mathcal{O},\sigma,T) \xrightarrow[\curvearrowleft,\mathcal{W}]{} (S,\mathcal{O},\sigma_0\cdot\sigma_1,(t,e)\parallel T) & & R9\ (\textit{spawn}) \\
\quad \text{if} \quad \sigma=\sigma_0\cdot(t',\mathsf{spw}_{t,e})\cdot\sigma_1 \ \&\ \sigma_0 \curvearrowleft (t',\mathsf{spw}_{t,e}) & & \\
(S,\mathcal{O},\sigma,T) \xrightarrow[\curvearrowleft,\mathcal{W}]{} (S,\mathcal{O},\sigma_0\cdot\sigma_1,T) & & R10\ (\textit{join}) \\
\quad \text{if} \quad \sigma=\sigma_0\cdot(t,\mathsf{join}_{t'})\cdot\sigma_1 \ \&\ \sigma_0 \curvearrowleft (t,\mathsf{join}_{t'}) & &
\end{array}
$$

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

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. In particular, the semantics preserves the atomicity of writes whenever the $\mathcal{W}$ component satisfies the following *coherence* axiom:

$$\forall W\in\mathcal{W}.\ \forall t,t'\in\mathcal{T}id.\ t\neq t' \ \&\ \{t,t'\}\subseteq W \ \Rightarrow W=\mathcal{T}id \qquad \text{(C)}$$

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

$$\sigma \curvearrowleft^{\mathcal{S}} (t,\xi) \ \Leftrightarrow_{\mathrm{def}} \ \sigma\restriction\mathcal{S}ync \curvearrowleft (t,\xi)$$

where $\sigma\restriction\mathcal{S}ync$ is the restriction of the sequence σ to the set $\mathcal{S}ync$, that is the subsequence of σ containing only the issued operations in $\mathcal{S}ync$.

We now comment the rules. In all cases but the early ones ($R2$ and $R5$), performing an operation from the temporary store σ consists in checking that the operation can be moved, up to $\curvearrowleft$, at the head of σ, and then in removing the operation from σ while possibly performing some effect. Namely, such an effect is produced when the performed operation is a read, a write, an acquire or a release, and the reference that is concerned by the effect must be known in these cases. A read may return a value if it can be moved next to the head of the temporary store (rule $R1$), or to a corresponding, visible write ($R2$). In the case of $R2$, the read operations should not be blocked by synchronization operations (such as a barrier or an acquire) previously issued but not yet globally performed. This is expressed as $\delta_0 \curvearrowleft^{\mathcal{S}} (t,\mathsf{rd}_{\varrho,\iota})$. In the case of an early read, the read operation does not completely vanish, but is transformed in a read mark $\overline{\mathsf{rd}}_\iota$, where ι identifies the matching write. Such a read mark, which is only useful in relation with barriers (as discussed below), can be eliminated from the temporary store as specified by $R3$. Notice that when we say that a read $(t,\mathsf{rd}_{\varrho,\iota})$ can be "moved," this is only an image: there is no transformation of the temporary store, but only a condition on it, namely, in $R1$, $\sigma_0 \curvearrowleft (t,\mathsf{rd}_{p,\iota})$. In the rules $R1$ and $R2$ for read operations, there is a global replacement of the identifier ι 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 cannot appear in the store.) Similarly, a write operation $\mathsf{wr}_{\varrho,v}^{W,I}$ from the temporary store $\sigma_0\cdot(t,\mathsf{wr}_{\varrho,v}^{W,I})\cdot\sigma_1$ may update the memory (rule $R4$) when ϱ is a reference p, v is in $\mathcal{V}al$ and the write is allowed to commute with the preceding operations, that is $\sigma_0 \curvearrowleft (t,\mathsf{wr}_{p,v}^{W,I})$. 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. In this rule, we should also assume that $W'\subseteq\mathsf{dom}(T)$ or $W'=\mathcal{T}id$, in order to restrict its application to finitely many cases.

Notice that, although the first operation in the temporary store is always allowed to be executed, by axiom (E), in the case where this operation is an acquire $\overset{\frown}{p}$ the condition $p\notin\mathsf{dom}(\mathcal{O})$ might not hold. In such a case, a deadlock might occur, and the computation might be stuck. As usual, our abstract semantics does not say anything about such errors – think of $(tt())$ for instance –, whereas an implementation generally tries to recover from them. We shall see

later that some other kinds of errors may arise when extending the framework with speculation. Then we will consider as valid – that is, non erroneous – only the relaxed computations that end up with an empty temporary store, i.e. where the resulting configuration is normal.

An obvious remark about the relaxed semantics is that it contains in a sense the interleaving semantics: 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 $t \in \mathcal{T}id$, the read early rule $R2$ is restricted to the "read-own-write-early" capability [3]. Obviously, the memory model is coherent in that case, that is, it satisfies axiom (C). In the write early rule $R4$, 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 [3]).

3.3 Memory Models: Requirements

In the next section we illustrate the expressive power of our framework for relaxed computations, by showing programs exhibiting behaviors that are *not* allowed by the reference semantics. Many 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

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

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:

$$\sigma \cdot (\omega, \xi) \cdot \sigma' \leftthreetimes (\omega', \xi') \;\Rightarrow\; \neg\big((\omega, \xi)\,\mathcal{P}\,(\omega', \xi')\big) \qquad (\mathrm{A}_{\mathcal{P}})$$

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

$$(p := tt)\,;\,(r := !\,p)$$
$$(r := !\,p)\,;\,(p := tt)$$

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

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

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

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

$$\left.\begin{array}{r}\varrho \in \{\varrho'\} \cup \mathcal{I}dent\ \&\\ t' \in W \cup \{t\}\end{array}\right\} \;\Rightarrow\; (t, \mathsf{wr}^{W,I}_{\varrho,v}) \blacktriangleleft (t', \mathsf{wr}^{W',I'}_{\varrho',v'})$$

$$\varrho \in \{p\} \cup \mathcal{I}dent \;\Rightarrow\; \begin{cases}(t, \widehat{p}\,) \blacktriangleleft (t, \mathsf{rd}_{\varrho,\iota}) \blacktriangleleft (t, \widehat{p}\,)\ \&\\ (t, \widehat{p}\,) \blacktriangleleft (t, \mathsf{wr}^{W,I}_{\varrho,v}) \blacktriangleleft (t, \widehat{p}\,)\end{cases}$$

and

$$(t, \widehat{p}\,) \blacktriangleleft (t, \widehat{p}\,)\ \&\ (t, \widehat{p}\,) \blacktriangleleft (t, \widehat{p}\,)$$
$$(t, \mathsf{join}_{t'}) \blacktriangleleft (t, \xi) \blacktriangleleft (t, \mathsf{spw}_{t',e})$$
$$(t, \xi) \blacktriangleleft (t', \mathsf{join}_{t})$$

The first group of properties ensures 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

$$\varrho \in \{p\} \cup \mathcal{I}dent \;\Rightarrow\; (t, \mathsf{rd}_{\varrho,\iota})\,\mathcal{P}\,(t, \mathsf{rd}_{p,\iota'})$$

to the precedence relation. With $\blacktriangleleft$ we also assume that accesses to a reference p enclosed into acquire $\widehat{p}$ and release $\widehat{p}$ actions from the same thread cannot be moved outside such a critical section. This is needed to preserve the locking discipline enforced by using the block structured locking construct $(\mathsf{with}\ p\ \mathsf{do}\ e)$. For instance, without these precedence relations, from the program

$$(\mathsf{with}\ p\ \mathsf{do}\ p := (\mathsf{not}\ !\,p)) \;\parallel\; (\mathsf{with}\ p\ \mathsf{do}\ p := (\mathsf{not}\ !\,p))$$

starting with a configuration where the value of p in the store is ff, one could end up in the relaxed semantics with a state where the value of p is tt, violating the role of the locking construct. Finally with $\blacktriangleleft$ we require that the causality relations associated with spawning or joining a thread in the interleaving semantics are preserved. For instance, with

$$p := tt\,;\,(\mathsf{thread}\ (r := !\,p))$$

or

$$(t, p := tt) \;\parallel\; (t', (\mathsf{join}\ t)\,;\,r := !p)$$

it is not possible to get the outcome $r = ff$. The only $\blacktriangleleft$ precedences that relate two distinct threads are $(t, \mathsf{wr}^{W,I}_{\varrho,v}) \blacktriangleleft (t', \mathsf{rd}_{p,\iota})$, $(t, \mathsf{wr}^{W}_{\varrho,v}) \blacktriangleleft (t', \mathsf{wr}^{W'}_{\varrho',v'})$ where $t' \in W$ or $I \neq \emptyset \neq I'$, and $(t, \xi) \blacktriangleleft (t', \mathsf{join}_t)$. This means in particular that a thread t' "sees" the writes, previously issued by other threads, that include t' in their scope, and 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 [31]). 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* $(\leftthreetimes, \mathcal{W})$ *where* $\emptyset \in \mathcal{W}$*, and the commutability predicate* $\leftthreetimes \subseteq \Sigma^*_{\mathcal{L}} \times \Sigma_{\mathcal{L}}$ *satisfies the axioms* (E) *and* ($\mathrm{A}_{\blacktriangleleft}$)*.*

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 ($\mathrm{A}_{\blacktriangleleft}$) is vacuously true). This model is trivially coherent. All the examples discussed in the following section hold in the minimal, or *most relaxed*, memory model $\mathcal{M}_{\blacktriangleleft}(\mathcal{L}) = (\leftthreetimes_{\blacktriangleleft}, 2^{\mathcal{T}id})$, where $\leftthreetimes_{\blacktriangleleft}$ is the largest commutability predicate satisfying ($\mathrm{A}_{\blacktriangleleft}$), $2^{\mathcal{T}id}$ is the set of all subsets of $\mathcal{T}id$.

In this work we mainly use commutability properties that are generated by precedence relations, in the sense of axiom ($\mathrm{A}_{\mathcal{P}}$). Then one could think of defining a memory model as a pair

$(\mathcal{P}, \mathcal{W})$, instead of $(\triangleleft, \mathcal{W})$. 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 $\neg\big(\sigma \triangleleft (t,\xi)\big)$, not on the basis that σ contains an operation that has precedence over (t,ξ), but because there is a subsequence of σ which, as a whole, has precedence over it.

4. Examples

Now we examine some examples of programs exhibiting behaviors that are not allowed by the reference semantics, indicating in each case the property the memory model is supposed to have, and which synchronization construct the language could offer to counteract such deviating behaviors. We do not formally define specific memory models here, but only suggest, in each case, which are the commutability properties or the properties of the write grain that support some particular behavior. As we said, all these properties hold in the most relaxed memory model $\mathcal{M}_{\blacktriangleleft}(\mathcal{L})$. In the examples we use a standard syntax, and follow the usual convention about the outcome, be it forbidden or allowed in the relaxed operational semantics. That is, an assignment $r_i := e$ (where most often e is $!\,p$ for some p) is annotated with the value, like for instance (tt) or (ff), that the reference r_i is supposed not to have, or to have, in the final state. Similarly, whenever several assignments for the same reference are present, we decorate with (*final*) the one that is intended to provide the final value (forbidden or allowed) for the reference. In each case where the specifed outcome can be obtained in the relaxed semantics, we sketch a corresponding behavior. In all the examples we assume that the initial values of the references are ff. We shall omit the ownership component, which is irrelevant here. We also omit the superscript W in $(t, \mathsf{wr}^{W,I}_{\varrho,v})$ whenever $W = \emptyset$, and similarly for I.

4.1 Simple Relaxations

(1) Let us start with the most common relaxation, the one of the **W→R** order [3], supported by simple write buffering [11] as in TSO machines. That is, we are assuming that there is no precedence between $(t, \mathsf{wr}^{W,I}_{p,v})$ and $(t, \mathsf{rd}_{q,\iota})$ if $p \neq q$. The litmus test here is the program (1) given in the Introduction. If we let

$$\begin{array}{rcl} T & = & (t_0, p := tt\,;\, r_0 := !\,q) \,\|\, (t_1, q := tt\,;\, r_1 := !\,p) \\ \sigma & = & (t_0, \mathsf{wr}_{p,tt}) \cdot (t_0, \mathsf{rd}_{q,\iota_0}) \cdot (t_1, \mathsf{wr}_{q,tt}) \cdot (t_1, \mathsf{rd}_{p,\iota_1}) \end{array}$$

we have

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

Given that the order **W→R** may be relaxed, we have

$$\left.\begin{array}{l} p \neq q\ \& \\ t_0 \neq t_1 \end{array}\right\} \Rightarrow \left\{\begin{array}{l} (t_0, \mathsf{wr}_{p,tt}) \triangleleft (t_0, \mathsf{rd}_{q,\iota_0})\ \& \\ (t_0, \mathsf{wr}_{p,tt}) \cdot (t_1, \mathsf{wr}_{q,tt}) \triangleleft (t_1, \mathsf{rd}_{p,\iota_1}) \end{array}\right.$$

and therefore

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

where $\sigma' = (t_0, \mathsf{wr}_{p,tt}) \cdot (t_1, \mathsf{wr}_{q,tt})$. These write operations can now be executed, and we reach a final state (S', ε, 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 some synchronization means. Most often these are *barriers*, that disallow some relaxations, when inserted between memory operations. For instance, to forbid the **W→R** relaxation, a natural barrier to use is $\langle \mathsf{wr} \rangle$ (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 $\langle \mathsf{wr} \rangle$, we require that the commutability predicate satisfies $(\mathrm{A}_{\mathcal{P}_{\langle \mathsf{wr} \rangle}})$ for a precedence relation $\mathcal{P}_{\langle \mathsf{wr} \rangle}$ such that

$$(t, \mathsf{wr}^{W,I}_{\varrho,v})\, \mathcal{P}_{\langle \mathsf{wr} \rangle}\, (t, \langle \mathsf{wr} \rangle)\, \mathcal{P}_{\langle \mathsf{wr} \rangle}\, (t, \mathsf{rd}_{\varrho',\iota})$$

(We do not have to specify that $\langle \mathsf{wr} \rangle$ has precedence over $\overline{\mathsf{rd}}_\iota$, 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, we write:

$$\begin{array}{l} p := tt; \\ \langle \mathsf{wr} \rangle; \\ r_0 := !\,q \end{array} \;\Big\|\; \begin{array}{l} q := tt; \\ \langle \mathsf{wr} \rangle; \\ r_1 := !\,p \end{array}$$

The threads will issue $\langle \mathsf{wr} \rangle$ before the reads rd_{q,ι_0} and rd_{p,ι_1}. Given the precedence relations we just assumed as a semantics for $\langle \mathsf{wr} \rangle$, 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 $\langle \mathsf{wr} \rangle$ once the writes $\mathsf{wr}_{p,tt}$ and $\mathsf{wr}_{q,tt}$ have been globally performed.

(2) Regarding the relaxation of the order **W→W**, typical of a PSO architecture, and supported by write buffering with "jockeying" [11], the litmus test is

$$\begin{array}{l} p := tt; \\ q := tt \end{array} \;\Big\|\; \begin{array}{l} r_0 := !\,q; \\ r_1 := !\,p \end{array}$$

We have

$$\begin{array}{rcl} (S, \varepsilon, T) & \xrightarrow[\triangleleft,\mathcal{W}]{*} & (S, \sigma_0, (t_0, \langle\rangle) \,\|\, (t_1, r_0 := !\,q\,;\, r_1 := !\,p)) \\ & \xrightarrow[\triangleleft,\mathcal{W}]{*} & (S, \sigma_0 \cdot \sigma_1, (t_0, \langle\rangle) \,\|\, (t_1, r_1 := \iota_1)) \end{array}$$

where

$$\begin{array}{rcl} T & = & (t_0, (p := tt)\,;\,(q := tt)) \,\|\, (r_0 := !\,q)\,;\,(r_1 := !\,p) \\ \sigma_0 & = & (t_0, \mathsf{wr}_{p,tt}) \cdot (t_0, \mathsf{wr}_{q,tt}) \\ \sigma_1 & = & (t_1, \mathsf{rd}_{q,\iota_0}) \cdot (t_1, \mathsf{wr}_{r_0,\iota_0}) \cdot (t_1, \mathsf{rd}_{p,\iota_1}) \end{array}$$

Here we assume that there is no precedence between $(t, \mathsf{wr}_{p,v})$ and $(t, \mathsf{wr}_{q,v'})$ if $p \neq q$, and therefore $(t_0, \mathsf{wr}_{p,tt}) \triangleleft (t_0, \mathsf{wr}_{q,tt})$, that is, we can perform the write $(t_0, \mathsf{wr}_{q,tt})$, and then rd_{q,ι_0} to finally reach a state (S', ε, T') where $S'(r_0) = tt$ and $S'(r_1) = ff$.

(3) The same example can be used to illustrate the relaxation of **R→R**. As in the case of the first example, to counteract the **W→W** and **R→R** relaxations, the language should offer barriers like $\langle \mathsf{ww} \rangle$ and $\langle \mathsf{rr} \rangle$, with an obvious semantics in the memory model:

$$\begin{array}{c} (t, \mathsf{wr}^{W,I}_{\varrho,v})\, \mathcal{P}_{\langle \mathsf{ww} \rangle}\, (t, \langle \mathsf{ww} \rangle)\, \mathcal{P}_{\langle \mathsf{ww} \rangle}\, (t, \mathsf{wr}^{W',I'}_{\varrho',v'}) \\ (t, \mathsf{rd}_{\varrho,\iota})\, \mathcal{P}_{\langle \mathsf{rr} \rangle}\, (t, \langle \mathsf{rr} \rangle)\, \mathcal{P}_{\langle \mathsf{rr} \rangle}\, (t, \mathsf{rd}_{\varrho',\iota'}) \\ (t, \overline{\mathsf{rd}}_\iota)\, \mathcal{P}_{\langle \mathsf{rr} \rangle}\, (t, \langle \mathsf{rr} \rangle) \end{array}$$

(4) To illustrate the relaxation of **R→W**, given that relaxing **W→W** is also allowed, we use:

$$\begin{array}{l} r_0 := !\,p; \\ q := tt \end{array} \;\Big\|\; \begin{array}{l} r_1 := !\,q; \\ p := tt \end{array}$$

Then

$$\begin{array}{rcl} (S, \varepsilon, T) & \xrightarrow[\mathcal{P},\mathcal{W}]{*} & (S, \sigma_0, (t_0, \langle\rangle) \,\|\, (t_1, r_1 := !\,q\,;\, p := tt)) \\ & & \sigma_0 = (t_0, \mathsf{rd}_{p,\iota_0}) \cdot (t_0, \mathsf{wr}_{r_0,\iota_0}) \cdot (t_0, \mathsf{wr}_{q,tt}) \\ & \xrightarrow[\mathcal{P},\mathcal{W}]{*} & (S, \sigma_0 \cdot \sigma_1, (t_0, \langle\rangle) \,\|\, (t_1, \langle\rangle)) \\ & & \sigma_1 = (t_1, \mathsf{rd}_{q,\iota_1}) \cdot (t_1, \mathsf{wr}_{r_1,\iota_1}) \cdot (t_1, \mathsf{wr}_{p,tt}) \end{array}$$

Since

$$\begin{array}{l} (t_0, \mathsf{rd}_{p,\iota_0}) \cdot (t_0, \mathsf{wr}_{r_0,\iota_0})\ \triangleleft^{\mathcal{P}}\ (t_0, \mathsf{wr}_{q,tt}) \\ (t_0, \mathsf{rd}_{p,\iota_0}) \cdot (t_0, \mathsf{wr}_{r_0,\iota_0}) \cdot (t_1, \mathsf{rd}_{q,\iota_1}) \cdot (t_1, \mathsf{wr}_{r_1,\iota_1})\ \triangleleft^{\mathcal{P}}\ (t_1, \mathsf{wr}_{p,tt}) \end{array}$$

we conclude as in the previous cases that we can reach a state where $S'(r_0) = tt = S'(r_1)$.

According to [31], the **R→W** relaxation is not observed on machines implementing the PowerPC memory model, though it should be assumed as part of this model. As in the previous cases, a $\langle \mathsf{rw} \rangle$ barrier is useful to prevent this relaxation, with the obvious semantics:

$$\begin{array}{c} (t, \mathsf{rd}_{\varrho,\iota})\, \mathcal{P}_{\langle \mathsf{rw} \rangle}\, (t, \langle \mathsf{rw} \rangle)\, \mathcal{P}_{\langle \mathsf{rw} \rangle}\, (t, \mathsf{wr}^{W,I}_{\varrho',v}) \\ (t, \overline{\mathsf{rd}}_\iota)\, \mathcal{P}_{\langle \mathsf{rw} \rangle}\, (t, \langle \mathsf{rw} \rangle) \end{array}$$

(5) Our last example for this subsection, which is a simplification of the example in Fig. 2 of [28], shows that there is no value arising "out of thin air" in our model. Namely, with

$$p := !\,q \,\|\, q := !\,p$$

we can only get the outcome where the value of both p and q is ff, because identifiers cannot be stored into the memory.

4.2 Early Reads and Writes

(6) The litmus tests that illustrate the rule $R2$ (in combination with $R5$) are less standard. The first one below, which holds in TSO models, exemplifies the **read-own-write-early** capability [3], that is the ability for a thread to read a write that it has previously issued, even if this write is not yet globally performed:

$$\begin{array}{l} p := tt; \\ r_0 := !\,p; \\ r_1 := !\,q \end{array} \;\Big\|\; \begin{array}{l} q := tt; \\ r_2 := !\,q; \\ r_3 := !\,p \end{array}$$

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

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

Then by $R2$ both ι_0 and ι_2 can take the value tt, whereas, given that the order $\mathbf{W}{\to}\mathbf{R}$ is relaxed (and that a read mark does not have precedence over a read), both ι_1 and ι_3 take the value ff from the shared store, before it is updated by performing the writes $\mathsf{wr}^{W_0}_{p,tt}$ and $\mathsf{wr}^{W_1}_{q,tt}$. We let the reader see where to insert $\langle \mathsf{wr} \rangle$ barriers to restore an SC behavior in this case.

(7) Let us see another example where the read-own-write-early capability is exercised:

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

From this program one can reach a configuration where the temporary store is

$$\sigma = (t_0, \mathsf{wr}_{p,q}) \cdot (t_0, \mathsf{rd}_{p,\iota}) \cdot (t_0, \mathsf{wr}_{\iota,tt})$$

Then, assuming that the write grain contains W such that $t_0 \in W$ and $t_1 \notin W$, by $R5$ and $R2$ the temporary store can evolve into σ' where

$$\sigma' = (t_0, \mathsf{wr}^{W,\{\iota'\}}_{p,q}) \cdot (t_0, \overline{\mathsf{rd}}_{\iota'}) \cdot (t_0, \mathsf{wr}_{q,tt})$$

Assuming the $\mathbf{W}{\to}\mathbf{W}$ relaxation, we can then reach a configuration where the temporary store is $(t_0, \mathsf{wr}^{W,\{\iota'\}}_{p,q}) \cdot (t_0, \overline{\mathsf{rd}}_{\iota'})$, and the store S' is such that $S'(q) = tt$. Next, the operations from the second thread are performed, in program order, and we end up with a state where the store S'' satisfies $S''(r_0) = tt$ and $S''(r_1) = ff$.

(8) There are several other examples to illustrate the write early rule $R5$, in combination with $R2$, in particular to show the ability for a thread to **read-others'-write-early**, according to the terminology of [3], and more precisely to break the atomicity of writes. Such an example is the one in Figure 2(b) in [3], but the best known is perhaps IRIW (Independent Reads of Independent Writes):

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

In our framework, this example is accounted for in the following way. Let

$$\begin{array}{rcl} T' & = & (t_0, ()) \,\|(t_1, ()) \,\|(t_2, (r_0 := \iota_0)\,;\,(r_1 := !\,q)) \\ & & \|(t_3, (r_2 := \iota_2)\,;\,(r_3 := !\,p)) \\ \sigma & = & (t_0, \mathsf{wr}_{p,tt}) \cdot (t_2, \mathsf{rd}_{p,\iota_0}) \cdot (t_1, \mathsf{wr}_{q,tt}) \cdot (t_3, \mathsf{rd}_{q,\iota_2}) \end{array}$$

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 $R4$ twice:

$$(S, \varepsilon, T) \xrightarrow[\gamma,\mathcal{W}]{*} (S, \sigma, T') \xrightarrow[\gamma,\mathcal{W}]{*} (S, \sigma', T')$$

where

$$\sigma' = (t_0, \mathsf{wr}^{W_0}_{p,tt}) \cdot (t_2, \mathsf{rd}_{p,\iota_0}) \cdot (t_1, \mathsf{wr}^{W_1}_{q,tt}) \cdot (t_3, \mathsf{rd}_{q,\iota_2})$$

Now since the write of p is made visible to thread t_2, the identifier ι_0 can take the value tt, and similarly ι_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.

(9) Some other examples that are discussed in [6, 31] can be explained in a similar 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:

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

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 $\mathsf{wr}^{W,I}_{p,tt}$ 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.

(10) The argument is the same for RWC (Read-to-Write Causality):

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

with the resulting values tt for r_0, and ff for both r_1 and r_2.

(11) We let the reader see how to build a computation in program order for the CC example of [6]:

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

with the outcome $S'(r_0) = tt = S'(r_3)$, $S'(r_1) = 0$ and $S'(r_2) = ()$. Obviously, one has to make some hypotheses about the write grain to deal with this example. Some hints: the write $(p := tt)$ from the first thread is made available to the second thread, but not globally performed. The second thread, and then the third proceed – the write $(p := ())$ can be performed since it does not see $\mathsf{wr}^{W}_{p,tt}$ –, next the fourth executes $(r_2 := !\,p)$, reading the value $()$ for p. Finally, before $(r_3 := !\,p)$ is performed, the write by t_0 is made available to the fourth thread, either by extending its visibility or by globally performing it.

4.3 Global Barriers

The behaviors discussed with the series of examples (**8-11**) in the previous subsection are forbidden in a coherent memory model, that is one satisfying axiom (C). Here we discuss some barriers from the PowerPC model, where write atomicity is relaxed, that are to be used to restore sequential consistency. Clearly, in the case of a non coherent memory model, and more generally 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. 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 $\langle \mathsf{wr} \rangle$, $\langle \mathsf{ww} \rangle$, $\langle \mathsf{rr} \rangle$ and $\langle \mathsf{rw} \rangle$. 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. Since the $\langle \mathsf{ww} \rangle$

(and $\langle \mathsf{rw} \rangle$) precedences already imply that a sync has precedence over a local write, to complete the description of the semantics of this barrier we just have to add:

$$t' \in W \;\Rightarrow\; (t, \mathsf{wr}^{W,I}_{\varrho,v})\, \mathcal{P}_{\mathsf{sync}}\, (t', \mathsf{sync})$$

We can then explain for instance the examples IRIW+syncs, WRC+sync+ppo and WRC+ppo+sync from the test suite by Sarkar & al. [31]. Here $\langle \mathsf{ppo} \rangle$ is a fictitious barrier, meaning that the program order among memory operations is supposed to be maintained. (In the simulator $\langle \mathsf{ppo} \rangle$ is implemented as a barrier.)

(12) If we introduce the sync barrier in the IRIW example, the unexpected outcome is then forbidden to occur:

$$p := tt \;\Big\|\; q := tt \;\Big\|\; \begin{array}{l} r_0 := !\,p\,;\ (tt) \\ \mathsf{sync}; \\ r_1 := !\,q\,(\mathit{ff}) \end{array} \quad \Big\|\; \begin{array}{l} r_2 := !\,q\,;\ (tt) \\ \mathsf{sync}; \\ r_3 := !\,p\,(\mathit{ff}) \end{array}$$

The reason is that the reads $(t_2, \mathsf{rd}_{q,\iota_2})$ and $(t_3, \mathsf{rd}_{p,\iota_3})$ can only be performed once the preceding sync operation has vanished, which in turn is only possible if there is no write in the temporary store that the sync sees.

(13) The WRC+sync+ppo example is the same as WRC above, but with a sync barrier inserted in the second thread:

$$p := tt \;\Big\|\; \begin{array}{l} r_0 := !\,p\,;\ (tt) \\ \mathsf{sync}; \\ q := tt \end{array} \quad \Big\|\; \begin{array}{l} r_1 := !\,q\,;\ (tt) \\ r_2 := !\,p\,(\mathit{ff}) \end{array}$$

The outcome $S'(r_0) = tt = S'(r_1)$ and $S'(r_2) = \mathit{ff}$ can still be obtained, while performing the operations of the third thread in program order. The explanation is as above, except that, instead of globally performing the write $(t_1, \mathsf{wr}_{q,tt})$, we extend its visibility to the third thread. Indeed, there is no condition in the rule $R5$ that prevents this to be done.

(14) The WRC+ppo+sync example is basically the same, except that the sync barrier is placed in the third thread instead of the second:

$$p := tt \;\Big\|\; \begin{array}{l} r_0 := !\,p\,;\ (tt) \\ q := tt \end{array} \quad \Big\|\; \begin{array}{l} r_1 := !\,q\,;\ (tt) \\ \mathsf{sync}; \\ r_2 := !\,p\,(\mathit{ff}) \end{array}$$

and the same outcome is allowed again, while committing the second thread in program order. The reasoning is as in the case of WRC (without sync) above: the sync issued by the third thread t_2 is not prevented to vanish by the write to p from the first thread, which it does not see.

The PowerPC architecture also provides an lwsync barrier, which is weaker than sync. First, this is a $\langle \mathsf{ww} \rangle$, $\langle \mathsf{rw} \rangle$ and $\langle \mathsf{rr} \rangle$ 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:

$$(t, \overline{\mathsf{rd}}_\iota)\, \mathcal{P}_{\mathsf{lw}}\, (t, \mathsf{lwsync}) \quad \& \quad (t, \mathsf{rd}_{\varrho,\iota})\, \mathcal{P}_{\mathsf{lw}}\, (t, \mathsf{lwsync})\, \mathcal{P}_{\mathsf{lw}}\, (t, \mathsf{wr}^{W,I}_{\varrho',v})$$
$$t = t' \text{ or } t' \in W \;\Rightarrow\; (t, \mathsf{wr}^{W,I}_{\varrho,v})\, \mathcal{P}_{\mathsf{lw}}\, (t', \mathsf{lwsync})$$

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

$$\left.\begin{array}{l} \sigma = \sigma_0 \cdot (t, \mathsf{lwsync}) \cdot \sigma_1\ \& \\ \big(\sigma_0 = \delta_0 \cdot (t, \mathsf{rd}_{\varrho,v}) \cdot \delta_1 \text{ or} \\ \sigma_0 = \delta_0 \cdot (t, \overline{\mathsf{rd}}_\iota) \cdot \delta_1\big) \end{array}\right\} \;\Rightarrow\; \neg\big(\sigma \curvearrowleft^{\mathsf{lw}} (t, \mathsf{rd}_{p,v'})\big)$$

This completes the definition of the semantics of lwsync. Now let us see two examples taken from [31] (or from the supplement to this paper available on the web), illustrating the semantics of this barrier.

(15) The first one is IRIW+lwsync+sync, that is the IRIW example where the barriers lwsync and sync are inserted in the third and fourth threads respectively. This does not prevent the unexpected outcome from occurring, as follows: the operation from the threads are issued in the temporary store, in order. Then the visiblity of the write $\mathsf{wr}_{p,tt}$ from the first thread is extended to include the third one (but not the others). Then the third thread reads p from this write, and executes $\mathsf{wr}_{r_0,tt}$. The lwsync is still prevented to vanish by the write $\mathsf{wr}^{\{t_0,t_2\}}_{p,tt}$, but it has no precedence over $(t_2, \mathsf{rd}_{q,\iota_1})$, which can therefore proceeds. The third thread performs $\mathsf{wr}_{r_1,tt}$. Next the operations from threads t_1 and t_3 are performed (the sync is not blocked by the write from t_0, which it does not see), and finally $(t_0, \mathsf{wr}^{\{t_0,t_2\}}_{p,tt})$ and (t_2, lwsync) are executed.

(16) Another interesting example is WRC+lwsyncs:

$$p := tt \;\Big\|\; \begin{array}{l} r_0 := !\,p\,;\ (tt) \\ \mathsf{lwsync}; \\ q := tt \end{array} \quad \Big\|\; \begin{array}{l} r_1 := !\,q\,;\ (tt) \\ \mathsf{lwsync}; \\ r_2 := !\,p\,(\mathit{ff}) \end{array}$$

The unexpected outcome is actually forbidden here, as verified using our simulator. If he/she attempts to get it, the reader will see that the lwsync from the third thread t_2 is prevented to vanish by a read mark that cannot be removed from the temporary store, since the corresponding write $(t_1, \mathsf{wr}^{W,I}_{q,tt})$ is prevented to be performed by the lwsync of t_1, which sees the write $\mathsf{wr}^{W',I'}_{p,tt}$ from t_0.

In an Appendix we examine in a rather sketchy way another series of litmus tests, taken from the test suite of Sarkar & al. [31], most of them exemplifying the semantics of the lwsync barrier. In each case, an unexpected outcome is provided, which is either forbidden or allowed by our model. All these tests, as well as the previous ones, have been checked using our simulator.

5. Speculation

In the previous sections we have described and illustrated a framework that supports the formal description of "classical" (hardware) memory models, as they are presented in the survey of Adve and Gharachorloo [3]. Indeed, our $\mathcal{M}_{\blacktriangleleft}(\mathcal{L})$ model captures the essential features of relaxed memory models such as RCpc [18] or C++ [5, 6] (regardless of the specific synchronization primitives each model may offer), and in particular the relaxation of write atomicity. With sync and lwsync, it also captures some essential features of the PowerPC memory model. In this section we further illustrate the flexibility of our approach, by discussing a feature that is not always considered part of a memory model, but can easily be accomodated in our framework, namely *speculation*. According to [20], speculative techniques are not part of the memory model (and they are not explicitly considered in [3]), still some of these techniques are involved in the PowerPC model for instance. The best known speculative technique is *branch prediction* [35]. This consists in guessing a value, true or false, for the predicate in a conditional branching construct, and evaluating the corresponding branch. Obviously, the prediction might happen to be wrong, in which case a rollback mechanism must be used to possibly undo some operations and backtrack to the branching point. It is quite easy to extend our model of relaxed computations to formalize this kind of speculation. In our syntax, a conditional program $(\mathsf{if}\ v\ \mathsf{then}\ e_0\ \mathsf{else}\ e_1)$ branches on a value v. We do not consider the erroneous cases where this is not a boolean value (or a variable, for only closed programs should be evaluated). Since we have rules for the cases where v is tt or ff, the only case where speculation could occur is when v is an identifier ι resulting from a read. To speculate the value of ι, we introduce new memory operations $[\iota = tt]$ and $[\iota = \mathit{ff}]$. More generally, we extend the syntax as follows:

$$\xi \;::=\; \cdots \;\mid\; [\nu = bv]_\iota$$

where $\nu \in \mathcal{I}dent \cup \{tt, \mathit{ff}\}$ and $bv \in \{tt, \mathit{ff}\}$. The index ι (which is not concerned by a substitution $\{\iota \mapsto v\}$) is a pointer to the corresponding read operation. Then we add the following rules to the ones in Figure 2:

$$(S, \sigma, (t, \mathbf{E}[(\mathsf{if}\ \iota\ \mathsf{then}\ e_0\ \mathsf{else}\ e_1)]) \,\|\, T) \hookrightarrow (S, \sigma \cdot (t, [\iota = tt]_\iota), (t, \mathbf{E}[\{\iota \mapsto tt\} e_0]) \,\|\, T)$$

$$(S, \sigma, (t, \mathbf{E}[(\mathsf{if}\ \iota\ \mathsf{then}\ e_0\ \mathsf{else}\ e_1)]) \,\|\, T) \hookrightarrow (S, \sigma \cdot (t, [\iota = \mathit{ff}]_\iota), (t, \mathbf{E}[\{\iota \mapsto \mathit{ff}\} e_1]) \,\|\, T)$$

With these rules, the conditional branching construct is a kind of non deterministic choice. The semantics of the tags $[\nu = bv]_\iota$ is as follows: first a solved tag $[bv = bv]_\iota$ may vanish from the temporary store:

$$\sigma_0 \curvearrowleft [bv = bv]_\iota \;\Rightarrow\; (S, \sigma_0 \cdot (t, [bv = bv]_\iota) \cdot \sigma_1, T) \xrightarrow[\mathcal{M}]{} (S, \sigma_0 \cdot \sigma_1, T)$$

Second, we assume that a prediction cannot overtake the corresponding read, and that the write operations cannot overtake a prediction from the same thread:

$$(t, \overline{\mathsf{rd}}_\iota)\, \mathcal{P}_{\mathsf{bp}}\, (t, [bv = bv]_\iota)$$
$$(t, [\nu = bv]_\iota)\, \mathcal{P}_{\mathsf{bp}}\, (t, \mathsf{wr}^{W,I}_{\varrho,v})$$

To see an example of an unexpected behavior introduced by branch speculation, let us consider

$$(\mathsf{if}\ !\,p\ \mathsf{then}\ r := !\,p\ \mathsf{else}\ ()) \;\|\; p := tt$$

and explain how the value ff is a possible outcome for r. Recall that in the syntax of our language, one should write

$$(\mathsf{if}\ !\,p\ \mathsf{then}\ r := !\,p\ \mathsf{else}\ ())$$

as $(\mathsf{let}\ x = !\,p\ \mathsf{in}\ (\mathsf{if}\ x\ \mathsf{then}\ r := !\,p\ \mathsf{else}\ ()))$. Then in the program above the first thread first append a $\mathsf{rd}_{p,\iota}$ to the temporary store, and passes ι as x for the conditional branching. Speculating that $[\iota = tt]_\iota$, one can then read p again and obtain the value ff. The prediction is finally justified by the write $p := tt$ from the second thread. There are other similar examples, where we do not have to assume that reads of the same reference from the same thread may be commuted. Such an example is "MP+sync+ctrl" from [31], which we simplify into:

$$\begin{array}{l} p := tt; \\ q := tt \end{array} \quad \| \; (\mathsf{if}\ !\,q\ \mathsf{then}\ r := !\,p\ \mathsf{else}\ )$$

With branch prediction, a possible outcome is a configuration where the value of r is ff, without reordering the writes in the first thread (one can imagine a barrier in between to ensure this), and without performing an early read of q. The computation is as follows: the second thread issues $\mathsf{rd}_{q,\iota}$, and speculates that $\iota = tt$. Then, given the **R**→**R** relaxation, the read $!\,p$ can be performed, returning ff. Finally, the first thread proceeds normally, and the prediction may be justified. In PowerPC there is an isync barrier to prevent such a behavior, by inserting it at the beginning of the then branch. A natural semantics for this barrier is

$$(t, [\nu = bv]_\iota)\, \mathcal{P}_{\mathsf{isync}}\, (t, \mathsf{isync})\, \mathcal{P}_{\mathsf{isync}}\, (t, \mathsf{rd}_{\varrho,\iota'})$$

Another, slightly more complex example is PPOCA, again from [31]:

$$\begin{array}{l} r := tt; \\ p := tt \end{array} \quad \| \quad \begin{array}{l} (\mathsf{if}\ !\,p\ \mathsf{then}\ q := tt\ \mathsf{else}\ ()); \\ r_0 := !\,q\,;\, r_1 := !\,r \end{array}$$

The unexpected behavior is as follows: the value tt is speculated for p in the second thread, the write $q := tt$ is issued and made visible to the thread, and therefore $!\,q$ may return tt, while $!\,r$, overtaking the rest, returns ff. The prediction is then justified by performing the operations of the first thread, in program order. Notice that besides branch prediction, we assumed simple TSO facilities (**W**→**R** relaxation and read-own-write-early capability), and **R**→**R** relaxation to exhibit this behavior.

A different, more general approach to speculative computations has been proposed in [8]. One should notice that, as opposed to this approach, our model preserves causality. An example, which is a simplification of the example given in Figure 4 of [28], is:

$$(\mathsf{if}\ !\,p\ \mathsf{then}\ q := tt\ \mathsf{else}\ ()) \;\|\; (\mathsf{if}\ !\,q\ \mathsf{then}\ p := tt\ \mathsf{else}\ ())$$

Since any prediction must be justified by a read from the memory, or from a write in the temporary store, we cannot reach from this program a state where the value of both p and q would be tt. This example can also be used to illustrate wrong predictions: the tags $(t_0, [\iota_0 = tt]_{\iota_0})$ and $(t_1, [\iota_1 = tt]_{\iota_1})$ that the threads may append to the temporary store will never disappear, and the computation will be stuck, failing to flush the temporary store. As usual, our abstract semantics does not say anything about such a stuck configuration, and in particular it does not specify any way to get out of such errors, whereas an implementation generally tries to recover from them. However, one should notice that, since the predictions $[\iota = bv]_\iota$ have precedence over writes, a rollback mechanism implemented to recover from such a situation does not have to undo speculative writes.

To conclude this section, we observe that branch prediction is just a particular case of *value prediction* [17, 25]. Indeed, one could remove the two transition rules above for the conditional branching construct, once added a more general rule for speculating the value returned by a read:

$$(S, \sigma_0 \cdot (t, \mathsf{rd}_{p,\iota}) \cdot \sigma_1, T) \xrightarrow[\mathcal{M}]{} (S, \sigma_0 \cdot (t, \mathsf{rd}_{p,\iota}) \cdot (t, [\iota = v]_\iota) \cdot \{\iota \mapsto v\}(\sigma_1, T))$$

This involves a more general form of tags, namely $[\nu = v]_\iota$ where v is any value. The semantics of such tags is the same as in the case of branch prediction. We do not know any example where value prediction alone – that is, not resulting in guessing the value of the predicate in a conditional branching – introduces a non SC behavior. Notice that the example **(5)** that we considered in the previous section is not altered, since for the computation to succeed, any prediction must be justified by a read from the store or a preceding write in the temporary store. With value prediction, any program including some reducible read has infinitely many behaviors. Then in a model involving this speculation mechanism, value prediction should be restricted to finite sets of values.

6. 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 20×10^{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 $60\,588$ 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 $\mathcal{T}id$ or a subset of $\mathsf{live}(T) \cup \mathsf{rdt}(\sigma_1)$ where the sets $\mathsf{live}(T)$ and $\mathsf{rdt}(\sigma)$ of thread identifiers are defined as follows:

$$\begin{array}{rcl} \mathsf{live}(\emptyset) & = & \emptyset \\ \mathsf{live}((t,e) \,\|\, T) & = & \mathsf{live}(T) \cup \{\, t \mid e \notin \mathcal{V}al \,\} \\ \mathsf{rdt}(\varepsilon) & = & \emptyset \\ \mathsf{rdt}((t,\xi) \cdot \sigma) & = & \mathsf{rdt}(\sigma) \cup \{\, t \mid \exists \varrho, \iota.\, \xi = \mathsf{rd}_{\varrho,\iota} \,\} \end{array}$$

We have not presented this formulation in Figure 3 only because it is conceptually a bit more obscure. 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 $51\,068$ to $13\,356$ for instance. Furthermore, one may observe that, since removing an operation from a temporary store σ never depends on what follows this operation in σ, 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 $2\,814$ 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 $10\,280$. Nevertheless this allowed us to successfully explore a large number of litmus tests, and in particular all the ones presented by Sarkar & al. [31] in their web files, when they can be written in our language. Our simulator is available on the web from the authors.

7. 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.

There are some memory model features that were not considered here, but deserve to be examined along the lines we have drawn, such as: read-modify-write operations, C++ atomics, compiler optimizations, as in the JAVA Memory Model [28]. We plan to investigate these topics in future work.

References

[1] M. ABADI, C. FLANAGAN, S. FREUND, *Types for safe locking: static race detection for Java,* ACM TOPLAS Vol. 28 No. 2 (2006) 207-255.

[2] S. ADVE, H.-J. BOEHM, *Memory models: a case for rethinking parallel languages and hardware,* CACM Vol. 53 No. 8 (2010) 90-101.

[3] S. ADVE, K. GHARACHORLOO, *Shared memory consistency models: a tutorial,* IEEE Computer Vol. 29 No. 12 (1996) 66-76.

[4] S. ADVE, M. D. HILL, *Weak ordering – A new definition,* ISCA'90 (1990) 2-14.

[5] M. BATTY, S. OWENS, S. SARKAR, P. SEWELL, T. WEBER, *Mathematizing C++ concurrency,* POPL'11 (2011) 55-66.

[6] H.-J. BOEHM, S. ADVE, *Foundations of the C++ concurrency model,* PLDI'08 (2008) 68-78.

[7] G. BOUDOL, G. PETRI, *Relaxed memory models: an operational approach,* POPL'09 (2009) 392-403.

[8] G. BOUDOL, G. PETRI, *A theory of speculative computations,* ESOP'10, Lecture Notes in Comput. Sci. 6012 (2010) 165-184.

[9] C. BOYAPATI, R. LEE, M. RINARD, *Ownership types for safe programming: preventing data-races and deadlocks,* OOPSLA'02 (2002) 211-230.

[10] S. BURCKHARDT, M. MUSUVATHI, V. SINGH, *Verifying local transformations on relaxed memory models,* CC'10, Lecture Notes in Comput. Sci. 6011 (2010) 104-123.

[11] M. DUBOIS, CH. SCHEURICH, F. BRIGGS, *Memory access buffering in multiprocessors,* ISCA'86 (1986) 434-442.

[12] L. EFFINGER-DEAN, D. GROSSMAN, *Modular metatheory for memory consistency models,* University of Washington, Computer Science & Engineering Tech. Rep. UW-CSE-11-02-01 (2011).

[13] X. FANG, J. LEE, S. P. MIDKIFF, *Automatic fence insertion for shared memory multiprocessing,* ACM ICS'03 (2003) 285-294.

[14] C. FLANAGAN, M. ABADI, *Types for safe locking,* ESOP'99, Lecture Notes in Comput. Sci. 1576 (1999) 91-108.

[15] C. FLANAGAN, M. FELLEISEN, *The semantics of future and its use in program optimization,* POPL'95 (1995) 209-220.

[16] C. FLANAGAN, A. SABRY, B. F. DUBA, M. FELLEISEN, *The essence of compiling with continuations,* PLDI'93 (1993) 237-247.

[17] F. GABBAY, A. MENDELSON, *Using value prediction to increase the power of speculative execution hardware,* ACM Trans. on Computer Systems Vol. 16 No. 3 (1998) 234-270.

[18] 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.

[19] D. GROSSMAN, *Type-safe multithreading in Cyclone,* TLDI'03 (2003) 13-25.

[20] M. D. HILL, *Multiprocessors should support simple memory-consistency models,* IEEE Computer Vol. 31 No. 8 (1998) 28-34.

[21] A. KRISHNAMURTHY, K. YELICK, *Optimizing parallel programs with explicit synchronization,* PLDI'95 (1995) 196-204.

[22] L. LAMPORT, *How to make a multiprocessor computer that correctly executes multiprocess programs,* IEEE Trans. on Computers Vol. 28 No. 9 (1979) 690-691.

[23] D. LEA, *The JSR-133 cookbook for compiler writers,* available from the author's web page (2008).

[24] J. LEE, D. A. PADUA, *Hiding relaxed memory consistency with a compiler,* IEEE Trans. on Computers Vol. 50 No. 8 (2001) 824-833.

[25] M. H. LIPASTI, C. B. WILKERSON, J. P. SHEN, *Value locality and load value prediction,* ASPLOS'96 (1996) 138-147.

[26] B. LUCIA, L. CEZE, K. STRAUSS, S. QADEER, H. BOEHM, *Conflict exceptions: Providing simple parallel language semantics with precise hardware exceptions,* ISCA'10 (2010).

[27] S. MADOR-HAIM, R. ALUR, M. MARTIN, *Generating Litmus tests for contrasting memory consistency models,* Tech. Rep. CIS 934, University of Pennsylvania (short version in CAV'10) (2010).

[28] J. MANSON, W. PUGH, S. A. ADVE, *The Java memory model,* POPL'05 (2005) 378-391.

[29] D. MARINO, A. SINGH, T. MILLSTEIN, M. MUSUVATHI, S. NARAYANASAMY, *DRFx: A simple and efficient memory model for concurrent programming languages,* PLDI'10 (2010).

[30] 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.

[31] S. SARKAR, P. SEWELL, J. ALGLAVE, L. MARANGET, D. WILLIAMS, *Understanding POWER multiprocessors,* PLDI'11 (2011) 175-186.

[32] J. ŠEVČÍK, D. ASPINALL, *On validity of program transformations in the JAVA memory model,* ECOOP'08, Lecture Notes in Comput. Sci. 5142 (2008) 27-51.

[33] 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.

[34] D. SHASHA, M. SNIR, *Efficient and correct execution of parallel programs that share memory,* ACM TOPLAS Vol. 10 No. 2 (1988) 282-312.

[35] J. E. SMITH, *A study of branch prediction strategies,* ISCA'81 (1981) 135-148.

Appendix

(17) MP+lwsyncs:

$$\begin{array}{l|l} p := tt; & r_0 := !\,q\,(tt) \\ \mathsf{lwsync}; & \mathsf{lwsync}; \\ q := tt & r_1 := !\,p\,(ff) \end{array}$$

The unexpected outcome is forbidden, because lwsync is a $\langle$ww$\rangle$ and $\langle$rr$\rangle$ barrier, which is prevented to vanish by the writes it sees, and by a read mark.

(18) SB+lwsyncs:

$$\begin{array}{l|l} p := tt; & q := tt; \\ \mathsf{lwsync}; & \mathsf{lwsync}; \\ r_0 := !\,q\,(ff) & r_1 := !\,p\,(ff) \end{array}$$

This outcome is allowed, because lwsync is not a $\langle$wr$\rangle$ barrier, and therefore one may execute the reads first.

(19) LB+lwsyncs:

$$\begin{array}{l|l} r_0 := !\,p\,;\ (tt) & r_1 := !\,q\,;\ (tt) \\ \mathsf{lwsync}; & \mathsf{lwsync}; \\ q := tt & p := tt \end{array}$$

The outcome is forbidden, because lwsync is a $\langle$rw$\rangle$ barrier, and a global $\langle$ww$\rangle$ barrier.

(20) RWC+lwsync+sync:

$$p := tt \;\|\; \begin{array}{l} r_0 := !\,p\,;\ (tt) \\ \mathsf{lwsync}; \\ r_1 := !\,q\,(ff) \end{array} \;\|\; \begin{array}{l} q := tt; \\ \mathsf{sync}; \\ r_2 := !\,p\,(ff) \end{array}$$

(allowed). This test is similar to IRIW+lwsync+sync.

(21) ISA2+lwsyncs:

$$\begin{array}{l|l|l} p := tt; & r_0 := !\,q\,;\ (tt) & r_1 := !\,r\,;\ (tt) \\ \mathsf{lwsync}; & \mathsf{lwsync}; & \mathsf{lwsync}; \\ q := tt & r := tt & r_2 := !\,p\,(ff) \end{array}$$

(forbidden). Again, this test illustrates the fact that lwsync is a $\langle$ww$\rangle$, $\langle$rw$\rangle$ and $\langle$rr$\rangle$ barrier.

(22) R+lwsync+sync (R01 in [31]):

$$\begin{array}{l|l} p := tt; & q := ff\,;\ (final) \\ \mathsf{lwsync}; & \mathsf{sync}; \\ q := tt & r_0 := !\,p\,(ff) \end{array}$$

In our model this is a "forbid," whereas the unexpected outcome is allowed by the model of [31]. However, it has not been observed by Sarkar & al. in their experiments on PowerPC machines.

(23) S+lwsyncs:

$$\begin{array}{l|l} p := ff\,;\ (final) & r_0 := !\,q\,;\ (tt) \\ \mathsf{lwsync}; & \mathsf{lwsync}; \\ q := tt & p := tt \end{array}$$

(forbid). This is similar to MP+lwsyncs.

(24) 2+2W+lwsyncs:

$$\begin{array}{l|l} p := ff\,;\ (final) & q := ff\,;\ (final) \\ \mathsf{lwsync}; & \mathsf{lwsync}; \\ q := tt & p := tt \end{array}$$

Forbid: lwsync is a $\langle$ww$\rangle$ barrier.

(25) WWC+lwsyncs:

$$p := tt\ (final) \;\|\; \begin{array}{l} r_0 := !\,p\,;\ (tt) \\ \mathsf{lwsync}; \\ q := tt \end{array} \;\|\; \begin{array}{l} r_1 := !\,q\,;\ (tt) \\ \mathsf{lwsync}; \\ p := ff \end{array}$$

(forbid). This is similar to previous cases: lwsync is a $\langle$rw$\rangle$ barrier, and a global $\langle$ww$\rangle$ barrier.

(26) WRW+2W+lwsyncs:

$$p := ()\ (final) \;\|\; \begin{array}{l} r_0 := !\,p\,;\ (()) \\ \mathsf{lwsync}; \\ q := tt \end{array} \;\|\; \begin{array}{l} q := ff\,;\ (final) \\ \mathsf{lwsync}; \\ p := tt \end{array}$$

(forbid). Again, this example is similar to previous cases.

(27) WRW+WR+sync+lwsync:

$$p := tt \;\|\; \begin{array}{l} r_0 := !\,p\,;\ (tt) \\ \mathsf{sync}; \\ q := tt \end{array} \;\|\; \begin{array}{l} q := ff\,;\ (final) \\ \mathsf{lwsync}; \\ r_1 := !\,p\,(ff) \end{array}$$

(allowed). Since lwsync is not a $\langle$wr$\rangle$ barrier, the read rd_{p,ι_1} from thread t_2 may be performed first.

If we consider WRW+WR+lwsync+sync, replacing the barrier in the third thread by a stronger one, the unexpected outcome is forbidden in our model, whereas it is allowed in the model of [31]. This is a second test on which our models differ. However, this is again an outcome that has not been observed when running the test on PowerPC machines.

(28) WRR+2W+lwsync+sync:

$$p := ()\ (final) \;\|\; \begin{array}{l} r_0 := !\,p\,;\ (()) \\ \mathsf{lwsync}; \\ r_1 := !\,q\,(ff) \end{array} \;\|\; \begin{array}{l} q := tt; \\ \mathsf{sync}; \\ p := tt \end{array}$$

(allowed). Similar to IRIW+lwsync+sync.

If we consider WRR+2W+sync+lwsync, exchanging the two barriers, then this outcome is forbidden in our model. This is a third test where our model is more strict than the one of [31] (and again, this behavior has not been observed by Sarkar & al.).

(29) SRSW:

This is a variant of the IRIW litmus test, where there is only one reference which is written and read by the various threads. We assume here integer values:

$$p := 1 \,\|\, p := 2 \,\|\, \begin{array}{l} r_0 := !\,p\,;\ (1) \\ r_1 := !\,p\,(2) \end{array} \;\|\; \begin{array}{l} r_2 := !\,p\,;\ (2) \\ r_3 := !\,p\,(1) \end{array}$$

We let the reader see that, if the program order is maintained in the third and fourth thread (say by inserting a barrier having the effect of $\langle$rr$\rangle$, like lwsync), the outcome $S'(r_0) = 1 = S'(r_3)$ and $S'(r_1) = 2 = S'(r_2)$ is not possible, thanks to the restriction aiming at guaranteeing a coherence order on writes in rule $R4$.