UNIVERSITÉ DE NICE - SOPHIA ANTIPOLIS

ÉCOLE DOCTORALE STIC
SCIENCES ET TECHNOLOGIES DE L'INFORMATION
ET DE LA COMMUNICATION

T H E S E

pour obtenir le titre de

Docteur en Sciences

de l'Université de Nice - Sophia Antipolis
Spécialité : Informatique

Présentée et soutenue par

Gustavo PETRI

Operational Semantics of Relaxed Memory Models

Thèse dirigée par Gérard BOUDOL

et préparée à l'INRIA Sophia Antipolis, projet INDES

soutenue le 29 novembre 2010

Jury :

Martín ABADI	-	University of California - Santa Cruz, (Examinateur) Microsoft Research, Collège de France
Andrew APPEL	-	Princeton University (Rapporteur)
Gilles BARTHE	-	IMDEA Software (Président)
Gérard BOUDOL	-	INRIA/Indes (Directeur)
Marieke HUISMAN	-	University of Twente (Directrice)
Xavier LEROY	-	INRIA/Gallium (Examinateur)
Jean-Jacques LÉVY	-	INRIA/Moscova (Rapporteur)

Abstract

Most current multiprocessor architectures and shared memory parallel programming languages are not sequentially consistent for parallel programs. Their possible behaviors are characterized by *weak* or *relaxed memory models*. A memory model describes the way in which parallel programs can interact by reading and writing the shared memory. Thus, a relaxed memory model exhibits more behaviors than sequential consistency (a "strong" memory model). The fact that most architectures have relaxed memory models has been known for decades, and yet few programmers understand which are the exact behaviors a parallel program can have in such architectures. We argue in this thesis that the problem stems from the difficulty in understanding the specification of these relaxed memory models. Firstly because few architectures or programming languages provide a formal definition of their memory model. And secondly because the majority of the existing formal definitions are axiomatic, which hinders their understandability and makes them unsuitable for language-based techniques such as static analysis or model checking. We propose an alternative characterization of relaxed memory models. Our characterization is *operational*, which we believe makes it simpler to understand for the programmer, and better suited to standard language-based techniques.

Our first contribution in this thesis is the operational formalization of write-buffering architectures. Write-buffering is pervasive across multi-core architectures, and thus its understanding is fundamental for parallel programming in such architectures. By means of standard programming languages concepts, we prove that the standard *Data Race Free (DRF) guarantee* is satisfied by our formalization. Hence, reasoning about sequentially consistent computations is sound for programs free of simultaneous accesses on a single memory location.

Our second contribution is a framework for the operational characterization of speculative computation techniques. This framework allows us to formally define the intuitive notion of *valid* speculation. For this framework two languages are considered; a high-level programming language that supports locks; and a low-level programming language, closer to the Instruction Set Architecture (ISA) of a machine, that supports only barriers and a simple compare-and-swap instruction. We identify properties for programs of both of these languages that are sufficient to guarantee that only sequentially consistent behaviors can be observed when the programs are executed speculatively.

The final contribution is the instantiation of the write-buffering and speculative frameworks to formalize the Total Store Ordering (TSO), Partial Store Ordering (PSO), and Relaxed Memory Ordering (RMO) memory models of the Sparc architecture. In particular, we observe that the framework of write buffers is not well suited to formalize liberal relaxations as allowed by RMO. We prove

a correspondence result between the formalizations of PSO and TSO in both frameworks. The fact that RMO cannot be instantiated by means of write-buffers is a good indication that the speculative framework is more general than the one of write buffers.

Résumé

La plupart des architectures multiprocesseurs et des langages de programmation parallèle à mémoire partagée actuels ne sont pas séquentiellement consistant pour les programmes parallèles. Leurs comportements possibles sont caractérisés par des modèles mémoire faibles ou relâchés. Un modèle mémoire décrit la manière dont les programmes parallèles peuvent interagir par des lectures et des écritures dans la mémoire partagée. Ainsi, un modèle mémoire relâché présente plus de comportements que le modèle séquentiellement consistant (modèle mémoire "fort"). Le fait que la plupart des architectures ont des modèles mémoire relâchés est connu depuis des décennies, et peu de programmeurs comprennent quels sont les comportements exacts qu'un programme parallèle peut avoir dans de telles architectures. Nous soutenons dans cette thèse que le problème provient de la difficulté à comprendre la spécification de ces modèles de mémoire. Ceci, d'abord car peu d'architectures ou de langages de programmation donnent une définition formelle de leur modèle mémoire, et, ensuite, parce que la plupart des définitions formelles existantes sont axiomatiques, ce qui les rendent difficiles à comprendre et inadaptées à des techniques basées sur le langage, telles que l'analyse statique ou le model checking.

Notre première contribution dans cette thèse est la formalisation opérationnelle des architectures à tampons d'écriture (write buffers). Les write buffers sont omniprésents dans les architectures multi-core, et donc leur compréhension est fondamentale pour la programmation parallèle dans de telles architectures. En utilisant des concepts standard des langages de programmation, nous démontrons que la classique "Data Race Free (DRF) guarantee" est satisfaite dans notre formalisation. Par conséquent, raisonner par des calculs séquentiellement consistant est correct pour les programmes libres d'accès simultanés sur une même case mémoire.

Notre deuxième contribution est un framework pour la caractérisation opérationnelle des techniques de calcul spéculatif. Ce framework nous permet de définir formellement la notion intuitive de spéculation valide. Pour cette formalisation deux langues sont considérés, un langage de programmation de haut niveau avec un mécanisme d'exclusion mutuel par verrous, et un langage de programmation de bas niveau, plus proche de l'Instruction Set Architecture (ISA) d'une machine, avec des mécanismes de barrières mémoire et des instructions atomiques. Pour les programmes de ces deux langages, nous identifions les propriétés suffisantes pour garantir que seuls les comportements séquentiellement consistant peuvent être observés lorsque les programmes sont exécutés de manière spéculative.

La dernière contribution est l'instanciation de ces deux frameworks sémantiques pour formaliser les modèles mémoire "Total Store Ordering

(TSO)", "Partial Store Ordering (PSO)", et "Relaxed Memory Ordering (RMO)" de l'architecture Sparc. En particulier, nous observons que le framework des write buffers n'est pas bien adapté pour formaliser des relaxations trop libérales comme le permet RMO. Nous démontrons un résultat de correspondance entre les formalisations de PSO et TSO dans les deux frameworks. Le fait que RMO peut pas être instanciée par le framework des write buffers est une bonne indication que le framework spéculatif est plus générale que celui des write buffers.

Contents

Chapter 1

Introduction: On Relaxed Memory Models and Programming Languages

Writing correct and efficient computer programs requires a thorough understanding of the underlying programming language semantics, that is, how programs of that language are executed. Therefore, having a clear, simple and formal semantics for the programming language is a prerequisite for writing programs with strong guarantees, let alone verifying their correctness.

The techniques and methodologies used to provide formal semantics for sequential programming languages are the subject of a well established and mostly uncontroversial discipline. A particularly useful formalism is Structural Operational Semantics of Plotkin [1975, 1981], in which the state transitions happening during the execution of the program are described by means of rewriting rules. For shared memory parallel programming languages – that is programs with multiple processes (or threads) that communicate through the memory – there is a rather obvious extension of the structural operational semantics of sequential programs, in which at each step of the execution a process (or thread) is nondeterministically chosen and a step of this process is performed atomically (that is, without being interrupted by steps of other processes). This is commonly known as the *interleaving semantics* of parallel programs. Another name for this semantics, which is more common to the literature of relaxed memory models, is *sequential consistency*. The concept of sequentially consistent systems was introduced by Lamport [1979] and it is considered the standard semantics of shared memory parallel systems. Indeed, most verification techniques for multiprocess programs rely on sequential consistency [Owicki and Gries, 1976; Brookes, 2007; Lamport, 1977; Jones, 1983].

In spite of the simplicity of sequential consistency, this is not the semantics supported by most multiprocessor systems [SPARC, 1994; Intel Corporation, 2007; AMD, 2010; PowerPC, 2009]. On the contrary, most parallel computer architectures provide semantical models that exhibit more behaviors than those allowed by sequential consistency. These semantical models are know as "relaxed" or "weak" memory models [Dubois et al., 1998; Adve and Gharachorloo, 1996]. It is a well known fact that parallel programming is harder than sequen-

tial programming [Owicki and Gries, 1976]. The adoption of relaxed memory models adds an extra level to the difficulty of programming such systems, since programmers are required to consider behaviors that do not correspond to the ones of the more familiar sequential consistency. We will comment later on the issues of programmability in relaxed memory models.

The adoption of relaxed memory models, as opposed to the simpler sequential consistency, curiously has its roots in performance considerations as regards sequential programs rather than parallel ones. Indeed, parallel programming has not been exposed to the programmer at the programming language level (in mainstream programming languages), until rather recently. And even if these days it is gaining a lot of interest, it would be hard to claim that parallel programming is a common practice in the software industry. We say that relaxed memory models have been adopted for performance as regards sequential programs because: first, it is a reason of *performance* that leads to the incorporation of optimizing techniques that cause parallel programs to expose more behaviors than those of sequential consistent executions, we will consider this aspect in detail in the next section; and second, these optimizing techniques regard mainly sequential programs since the correctness consideration to qualify these optimizations as correct has been, for the most part, the semantics of *sequential programs*. This last aspect is partly due to the prolonged lack of interest in parallelism that we have mentioned before, implying that the development of these techniques has been rather oblivious to the semantics of parallel programs. Let us devote the following section to the consequences of these optimizing techniques on the semantics of parallel programs.

1.1 Performance First: The Need for Relaxed Memory Models

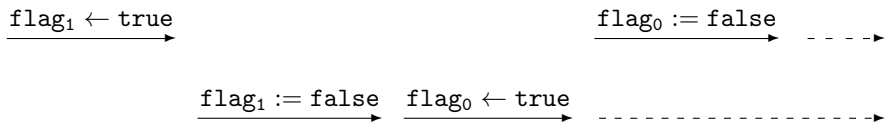
Performance is a crucial aspect of computer systems. It was early noticed that memory accesses have an important impact on the execution time of programs. Fortunately, the latency induced by memory requests can be greatly reduced by performing memory accesses in parallel, in an asynchronous way, hopefully without disrupting the semantics of programs. For a store to the memory this means that one can continue executing instructions after issuing the store, without waiting for an acknowledgment from the memory unit; and for reads this means that one can issue several reads in parallel, or even ahead of time. Techniques such as write buffering [Dubois et al., 1998], pipelining and instruction-level parallelism [Fisher, 1981; Hennessy and Patterson, 1996], branch prediction and speculative computation [Smith, 1981; Hennessy and Patterson, 1996], achieve this kind of effects and have been early adopted in commercial computer architectures, improving their overall performance. But this raises the following question: Does the semantics of programs running in machines empowered with these optimizations remain unchanged with respect to machines that do not implement the optimizations? The intuitive answer to this question should be that these semantics correspond to each other, as the word *optimization* seems to suggest. This is actually the case for *sequential programs*, and more generally, for programs running on a single processor. However, for parallel programs, these techniques can result in deviations from the sequentially consistent seman-

tics. Let us illustrate this issue by a very simple example taken from the seminal work by Lamport [1979] in the area of relaxed memory models. In this example we assume the initial value stored in the variables (or more precisely memory locations) $\texttt{flag}_0$ and $\texttt{flag}_1$ is the boolean `true`.

Example 1.1.

$$\left[\begin{array}{l} \texttt{flag}_0 := \texttt{false}\,; \\ \textsf{if } \texttt{flag}_1 \textsf{ then} \\ \qquad \textit{critical section 0} \end{array}\right] \parallel \left[\begin{array}{l} \texttt{flag}_1 := \texttt{false}\,; \\ \textsf{if } \texttt{flag}_0 \textsf{ then} \\ \qquad \textit{critical section 1} \end{array}\right]$$

In this work Lamport searches for conditions that suffice to guarantee that the execution of the parallel program is correct. He starts by observing that if the the write of $\texttt{flag}_0$ and read of $\texttt{flag}_1$ in the condition of the branching construct are performed in the reverse order in the first thread, there is no violation of the sequential semantics of the thread. That is to say, that the result of thread 1 (in isolation) is unaffected by performing these actions in the reverse order. However it is observed in that work that in this case, the mutual exclusion of the proposed algorithm does no longer work. Let us see a schema of a possible computation to support this claim. In the picture below we depict the thread on the left at the top and the one on the right at the bottom, where we label by $p := v$ the event of updating the memory location p with value v and by $p \leftarrow v$ the event of reading the value v from the memory location p:



We can see that the order in which instructions are performed in the left thread is reversed, which leads to a computation where both threads can see a value `true` for the flag and thus, both threads can enter their critical sections. Lamport concludes then that to "correctly execute multiprocess programs" the architecture must guarantee that "each processor issues memory requests in the order specified by its program" – this is commonly known as the *program order*. Unfortunately Lamport's requirement for "correct execution" (in fact for sequential consistency) is not met by most current multiprocessor architectures.

In the above example we have considered the result of reordering the execution of the read and write instructions in the program on the left, but we have not considered why this could happen. In fact, there are many ways in which common architectures could manifest that behavior. Indeed, the universal reason is the performance of programs. To have a better performance, architectures implement optimizations which could have the effect of reordering the actions to different processes. An example of a mechanism that performs this kind of reordering is write buffering [Dubois et al., 1998], that instead of performing writes immediately delays writes in internal buffers without interrupting the execution. In the example above we could consider that the write of $\texttt{flag}_0$ has been buffered, and it has only reached the memory after the read of $\texttt{flag}_1$ was performed. Other architectural optimizing techniques, such as pipelining and caching could have similar effects. We will discuss some of the effects of these techniques in the sequel.

In fact, these optimization techniques do not pertain only to hardware, but to software as well. It is indisputable that high-level programming languages provide a good abstraction level for programming complex algorithms. However, being at a higher level of abstraction makes it hard for the programmer to take full advantage of the low-level facilities of the machine in which the program will be deployed. To avoid considering such low-level details is precisely the purpose of a high-level programming language. Automatic program optimizations provide a mostly satisfactory answer to the trade-off between the ease of programming, achieved by high-level programming languages, and the efficiency of low-level mechanisms provided by the machine architecture. These optimizations could happen at different instances of the program life cycle (including compile-time and run-time). Notably, most modern compilers optimize code along the compilation process and some virtual machines, or just-in-time compilers optimize code along the execution of the program. Observe that a compiler that reorders memory access on different memory locations can lead to the result in the example above even if the machine architecture does not optimize the accesses. One might wonder though, if a compiler is allowed to reorder the code in this way, or, formulated as a question: Which is the criterion to decide whether a certain optimization is correct or not with respect to parallel programs?

A brief answer to this question is that the criterion for the validity of a compiler optimization – and for a hardware optimizing technique for that purpose – is the *sequential semantics* of the program. In other words, the de-facto standard is that a compiler optimization is considered correct if the sequential semantics of the original piece of source code corresponds to the sequential semantics of the optimized version of the code, where the program is assumed not to be parallel. This inevitably leads to behaviors like the one we considered in the previous example, which do not coincide with any sequentially consistent execution of the concurrent program. Since most compiler and hardware optimizations are parallelism "agnostic", in general parallel programming languages do not entail a sequentially consistent semantics. There are many ways in which program transformations can lead to behaviors that are not sequentially consistent for parallel programs, ranging from compiler optimizations to hardware optimizations; these sources are eloquently developed by Gao and Sarkar [1997]. Only recently the effects of compiler optimizations have been considered with respect to the semantics of parallel programs [Manson et al., 2005; Ševčík and Aspinall, 2008; Ševčík, 2009; Burckhardt et al., 2010]. However, the criterion for considering an optimization correct with respect to a parallel program in these works, is no longer that the source and optimized version correspond to each other with respect to their simplest semantics – that is the interleaving one – but rather that they correspond in their interleaving semantics for a particular class of programs; namely *Data-Race-Free* programs.

We will discuss the definition and the importance of this property shortly, but before moving on in the discussion of relaxed memory models it is interesting to notice here that there is a priority inversion regarding the correctness criterion for considering optimizations valid between sequential and parallel programs. While for sequential programs a simple semantics prevails as the driving criterion, for parallel programs performance prevails against a simple semantics (i.e. sequential consistency). The performance penalty required to implement sequential consistency is in general considered unacceptable for mul-

tiprocessor machines. It has been shown however, that for the class of data-race-free programs we can still benefit from a simple semantics without necessarily penalizing performance excessively [Adve and Hill, 1990; Gharachorloo et al., 1990]. We mention however that the complexities induced by programming with relaxed memory models have recently driven many researchers to reconsider whether sequential consistency is really unacceptable [Hill, 1998], and even more to reconsider the nondeterministic nature of shared memory parallel programming [Bocchino et al., 2009; Bergan et al., 2010].

Let us conclude this section by considering typical examples of behaviors to be found with relaxed memory models that could perhaps surprise the programmer. As we have mentioned before, optimizations having the effect of reordering writes with respect to subsequent reads can induce behaviors that are not sequentially consistent. This can be observed in the following example, where the initial memory contains a 0 value for both locations p and q and where we assume that r_0 and r_1 are registers local to a single processor (or thread):

Example 1.2.

$$\begin{bmatrix} p := 1\,; \\ r_0 := q \end{bmatrix} \quad \| \quad \begin{bmatrix} q := 1\,; \\ r_1 := p \end{bmatrix}$$

One might be surprised to learn that the final result $r_0 = r_1 = 0$ is indeed possible. But this is not so surprising if we are aware of the fact that many architectures reorder read memory accesses with respect to previous write accesses provided that these are on different references. In this example it suffices to execute both reads first to obtain the result in question. This behavior is typical of many memory architectures, including the x86 memory models [Owens et al., 2009; Intel Corporation, 2007; AMD, 2010], the SPARC [1994] memory models, and the Java programming language [Manson et al., 2005] just to mention a few. We refer to the work of Adve and Gharachorloo [1996] for a more exhaustive list of such architectures.

A similar phenomenon happens when the architecture has the effect of reordering write instructions on different locations. The following example is a clear indication of that:

Example 1.3.

$$\begin{bmatrix} p := 1\,; \\ q := 1 \end{bmatrix} \quad \| \quad \begin{bmatrix} r_0 := q\,; \\ r_1 := p \end{bmatrix}$$

If the architecture has the capability of reordering the write accesses in the left thread, the final result $r_0 = 1$ and $r_1 = 0$, which is not sequentially consistent becomes possible. This is typical of the PSO and RMO (Relaxed Memory Ordering) memory models of SPARC [1994] that we will consider in Chapter 4. Notice that if writes cannot be reordered but reads can, it suffices to reorder the reads of the right thread to have the exact same result, this is possible in the RMO memory model.

Let us consider a final example that is possible in the RMO memory model, and happens if one is allowed to reorder writes with respect to preceding reads on a different memory address:

Example 1.4.

$$\begin{bmatrix} r_0 := q\,; \\ p := 1 \end{bmatrix} \quad \| \quad \begin{bmatrix} r_1 := p\,; \\ q := 1 \end{bmatrix}$$

We can see that under that relaxation, the final result $r_0 = r_1 = 1$ becomes possible. This can happen in the Sparc RMO memory model as we will see.

Many more relaxations that we do not consider here are possible, in particular the Java memory model [Manson et al., 2005] allows behaviors for which current optimizing techniques do not exist yet. This has been considered in anticipation to future possible optimizations. We will discuss other common optimizations and relaxations by means of examples, as they become relevant to the discussion in this thesis.

1.2 Relaxed Memory Model: the Basics

Knowing that most parallel programming languages and computer architectures do not provide sequentially consistent semantics inevitably leads to the question: Which are the behaviors allowed for a parallel program? This is exactly the topic of memory models.

In essence a memory model specification dictates which are the possible values that a read on a memory location is capable of returning. Clearly, the possible values a read might return are intrinsically related to the original value of the location in the memory and writes issued to that location by the threads of the program. Actually, the answer to this question is trivial for sequential programs, where every read must return the last value written to that reference. For parallel programs, the interleaving semantics is the direct extension of the sequential semantics, since it induces a total order of memory accesses by choosing a single thread at a time and computing with that thread. However, for relaxed memory models there is a significant shift from the sequentially consistent semantics. In fact, memory accesses on a certain reference need not be totally ordered in the computation, meaning that it is not always obvious which is the last write to a certain reference. This is typically the case of concurrent accesses to the same reference with at least one of them being a write, a phenomenon known by the name of *data race*. Let us illustrate this issue with a very simple example with only two threads accessing a single reference. We take p to be an arbitrary memory location, and we consider that r_0 is a local variable to the left thread (or local register in a processor) and we assume the initial value of p to be 0.

Example 1.5.

$$\left[\begin{array}{l} p := 1\,; \\ r_0 := p \end{array} \right] \quad \| \quad \Big[p := 2 \Big]$$

For the sake of clarity let us consider the schema of the following interleaving computation, where as before, we depict the left thread on top and the right one below:

$$\xrightarrow{p := 1} \qquad \xrightarrow{p \leftarrow 2} \xrightarrow{r_0 := 2}$$

$$\xrightarrow{p := 2}$$

It is indisputable that this is an expectable behavior for the program above, in particular, it is perfectly sequentially consistent. But let us now consider a small variation of the semantics. Let us assume that every write will not be directly performed in the main memory but instead it will be stored in a buffer local to the processor, where only the processor owning the buffer can read. Clearly these buffers have to be flushed, updating the memory accordingly. Let us consider that in the computation we will denote by `update p v` the fact that a write that was standing in a buffer has finally been committed into the memory. Let us suppose that if a buffer is pending for update, only the thread issuing the write is able to read that value, and indeed, no other value can be read. Then we can see that the previous computation directly translates to something like:

$p := 1 \xrightarrow{} \texttt{update}\ p\ 1 \xrightarrow{} \qquad\qquad p \leftarrow 2 \xrightarrow{} r_0 := 2 \xrightarrow{}$

$p := 2 \xrightarrow{} \texttt{update}\ p\ 2 \xrightarrow{}$

And let us now consider what happens if we delay the update of the thread on the left after the one of the thread on the right:

$p := 1 \xrightarrow{} \qquad\qquad \texttt{update}\ p\ 1 \xrightarrow{} p \leftarrow 1 \xrightarrow{} r_0 := 1 \xrightarrow{}$

$p := 2 \xrightarrow{} \texttt{update}\ p\ 2 \xrightarrow{}$

Suddenly, adding just a small piece of hardware, which in fact does not necessarily reorder the execution of instructions, implies that the definition of last write is not univocally defined anymore. Maybe in this case one should talk about the last write to be *performed* (or *updated* from the buffer into the memory) rather than *issued*. In fact, these distinctions between performed and issued are delicate, and sometimes dependent on the particular optimization being considered. Moreover, the programmer producing the source code program does not need to be aware that there is an *update* action involved, and from the programming language point of view it is best that way.

What one can observe from the above example is that in relaxed memory models, if we need to keep the description to the actions that are recognizable at the source code level (thus excluding the `update` actions of the above example) the total order provided by the interleaving semantics cannot be assumed, and a better notion is an irreflexive partial order (that is an antisymmetric and transitive relation) between instructions. In the example above we can only assume that both writes happen before the read (for the particular interleaving considered) and that therefore the read action can see any of the two. Notice however that the result of the final computation we considered above corresponds to another interleaving of the source program. We see now that specifying a *correct* memory model with a semantics that is precise enough to be programmable is not an obvious task. We will discuss an operational formalization of write-buffering, similar to the one in this example in Chapter 2, where we will provide a syntactic representation for the buffers that are implicit in the example. We notice that since the specification of memory models considers partial orders and a *set* of possible values for a read action – as opposed to a a total order defining a unique value – it does not fit well in the standard semantical tools for formalizing operational semantics.

Another important aspect of the memory model is that it defines which are the *synchronization constructs* [Briggs, 1979; Dubois et al., 1998] that permit to impose constraints on the possible values a read might return, or more generally on the behavior of memory-related actions. For example, knowing that in a certain architecture accesses to different locations on the memory can be reordered, requires a mechanism to restore sequentially consistent executions; for this particular case such a mechanism could be barrier instructions. Notice that these mechanisms are mostly useless when considering sequential programs. Clearly the system is not forced to provide these mechanisms, but not being able to reestablish the standard interleaving semantics of parallel programs would make programming very hard if not impossible, which is why, to the best of our knowledge, all realistic memory models provide one or more such mechanisms. Thus, it is task of the memory model specification to define which are the synchronization mechanisms provided to restore sequential consistent executions – or more generally to prevent undesired relaxations.

Finally, the memory model specifies the *atomicity* of memory accesses. For example, in certain architectures accesses to double words are guaranteed to be atomic (that is uninterruptible) whereas other architectures might provide atomicity only for single words. Also the atomicity of instructions like compare-and-swap is precisely defined by the memory models specification. In this work we will concentrate mainly on the *ordering* and *visibility* aspects of relaxed memory models, rather than atomicity of individual memory accesses. We believe this aspect of memory models is not hard to model by considering that nonatomic memory locations correspond to more than one memory access in our framework.

1.3 Programmability: The Data Race Freeness Guarantee

We have seen that many multiprocessor architectures and high-level programming languages are not provided with a sequentially consistent semantics but rather a *relaxed memory model*. Let us once more motivate the discussion with a question: Which is, then, the exact semantics for parallel programs in such architectures or high-level programming languages? This question led researchers to investigate under which circumstances, or for which type of programs, the semantics of the relaxed model coincides with sequential consistency. Among the first to investigate this issue were Dubois et al. [1998], who identified the importance of synchronization in reestablishing sequential consistency for hardware with write-buffering capabilities. In that work the authors define a property of multiprocessor systems which they call *weak ordering*. The basic idea of weak ordering is to classify variables in two different classes:

- *Normal* variables, that might be shared and form part of the program logic but do not control the concurrent execution, and
- *Synchronization* variables, "which protect the access to shared writable operands or implement synchronization between processors."

Then, according to this classification of variables the authors propose the following conditions, which we will explain below:

A multiprocessor system is *weakly ordered* if:

1. accesses to global synchronizing variables are strongly ordered and if
2. no access to a synchronizing variable is issued in a processor before all previous global data accesses have been performed and if
3. no access to global data is issued by a processor before a previous access to a synchronizing variable has been performed.

The first condition, in brief, requires synchronizing accesses, that is accesses to synchronizing variables, to obey a sequentially consistent semantics with respect to each other. Notice that this item does not mention normal variables in any way. The terminology *strongly ordered* means that for the concerned events, in this case synchronizing memory accesses, there exists a total order which reflects the execution of these events in the program. In fact, every synchronizing read must see the value of the latest write on the same variable that immediately precedes it in this total order. In this sense, the value of a synchronizing access is uniquely determined by the interleaving of synchronizing actions. The second condition requires any "global" data access previous, in the sense of the order of the program text, to a synchronizing memory access to be performed before the latter. Here the terminology *performed* means that the effects of these memory accesses are globally visible to all the other threads. In other words, this condition requires that before engaging into a synchronizing memory access the effects of previously issued memory operations, synchronizing or normal, must be accessible (or visible) to any other processor in the system. Finally, the third item requires that accesses to global data, synchronizing or normal, be delayed until all previously issued synchronizing memory accesses are *performed*. In essence, it requires that memory accesses following a synchronizing one, do not start until the effects of the synchronizing access are visible to all the processors in the system.

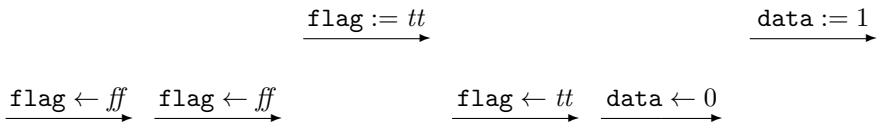
Let us see what is achieved by this property with a simple example. We consider a slight variation of the producer consumer algorithm of Adve and Gharachorloo [1996]:

Example 1.6.

$$\left[\begin{array}{l} \texttt{data} := 1\,; \\ \texttt{flag} := \mathit{tt} \end{array}\right] \;\|\; \left[\begin{array}{l} \mathsf{while}\ (\texttt{flag} = \mathit{ff})\ \mathsf{do}\ \mathsf{skip}\,; \\ \mathtt{r}_0 := \texttt{data} \end{array}\right]$$

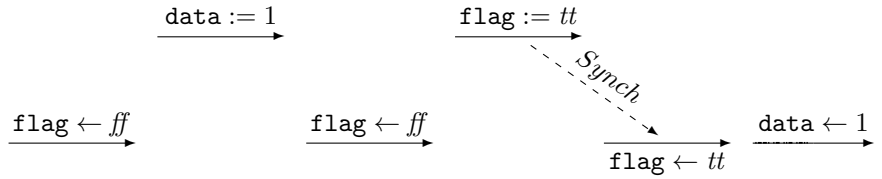
Let us analyze two cases, first consider the case where the variable `flag` is *normal* and later the one in which it is *synchronizing* according to the previous definition. To illustrate this point we will assume that *normal* write memory accesses can be reordered (where the reason for this reordering is left implicit) with respect to previously issued write accesses. If, in the first case, the variable `flag` is a normal one, nothing in the definition implies that the write to `data`, which we assume to be normal, and `flag` cannot be reordered. So a possible execution is presented in the following schema, where the computation of the

thread on the left is placed on top and the thread on the right is at the bottom.



We can easily understand in this execution why the nonsequentially consistent behavior where r_0 is set to 0 results from considering the reordering of *normal* write accesses to different references. Notice here that the above picture is just a simplification of the many ways in which this behavior could happen; in particular, the accesses to `data` and `flag` could happen at the same time, or even be delayed in buffers for example, here we depict only the moment in which this accesses are *performed* and not when they are *issued*.

However, if we consider the program more carefully we see that the `flag` variable is only there for controlling the parallel computation, actually to notify the fact that the variable `data` has been modified. Therefore, according to the definition of Dubois et al. [1998] stated above, this variable should be considered a synchronizing one. Let us reconsider the above execution but now assuming that `flag` is a synchronizing variable. If the above execution were permitted by the system we would have that the normal global access of `data` is performed only after the synchronizing access of `flag` is issued. A clear violation of the second requirement. Thus, the above execution does not correspond to a *weakly ordered* system. Indeed, we can see that all weakly ordered computations must have the following shape:



with possibly more, or less reads of the `flag` variable returning *ff* at the beginning of the lower thread. One can observe here that reading 0 for the variable `data` is impossible, which exactly corresponds to sequentially consistent computations of this program.

It might be instructive to observe here that if all communications – that is a write in one thread and a read of the same reference in a different thread – between normal memory accesses are separated by communications on synchronizing variables one gets sequentially consistent executions. This intuition will be further developed in the rest of this section.

In fact, it is natural when looking at these diagrams to make a connection between synchronizing accesses and the events in the *happens-before* definition of Lamport [1978]. In that work Lamport defines the happens-before relation to be the smallest relation such that if two events a and b are related by the program order (i.e. they are generated by the same processor, and a comes before b in the program text), then a happens-before b; and if a is the sending of a message by one process (in our case writing a synchronizing variable) and b is the receipt of that message by a different process (in our case a read obtaining the value of the previously mentioned write) then a happens-before b; and this relation is transitive. According to this definition we see that in the first case, when

`flag` is normal, the writing and reading of `data` are not happens-before related, whereas in the case where `flag` is synchronizing the writing of `data` necessarily *happens-before* the reading, and thus, it must see the value of the write. We will see that the happens-before relation is fundamental in the definition of some relaxed memory models.

Defining which variables should be synchronizing and which should not, highly depends on the relaxations allowed by the underlying relaxed memory model. In the definition of weak ordering that we have considered above the programmer is responsible for identifying which variables are synchronizing and which are normal. Indeed, this classification of variables depends on the underlying memory model, since in some cases variables that imply communication are not subject to memory model relaxations, and therefore do not need to impose extra synchronization, which could degrade the performance unnecessarily. An example of this can be found when considering the TSO (Total Store Ordering) relaxed memory model [SPARC, 1994] in contrast with the PSO (Partial Store Ordering) one [SPARC, 1994]. In the former writes cannot be reordered with respect to previous writes, and thus in examples like the one above there is no need to classify the `flag` variable as synchronizing (even if it does synchronize). However, in the latter one, this relaxation is possible, and thus `flag` has to be considered synchronizing. The fact that this classification can depend on the memory model in question implies that the programmer is still required to reason about the relaxed memory model, which is somewhat unsatisfactory from a high-level programming language point of view. This requires programmers, at the higher-level, to be aware of the intricacies of the underlying architecture, which is exactly what high-level programming languages attempt to avoid. This problem only gets exacerbated if we consider multiplatform programming languages like Java for instance, since it targets multiple architectures, and programs are subject to intermediate virtual machines that could optimize code as well.

A more satisfactory answer, from the programmer view point, to the problem of which programs provide sequentially consistent semantics under the relaxed memory model, was given by Adve and Hill [1990] and Gharachorloo et al. [1990] simultaneously. Adve and Hill [1990] propose what they called a *new definition* of weak ordering. This definition relies on a *synchronization model*, which is a programming discipline, assessing when synchronization needs to be performed. Programs satisfying this programming discipline are considered to be *correctly synchronized*. Then, the proposed weak ordering definition is as follows:

> Hardware is weakly ordered with respect to a synchronization model if and only if it appears sequentially consistent to all software that obey the synchronization model.

The idea here is that the compliance to the synchronization model must be determined only considering the semantics (in particular the sequentially consistent semantics) of the high-level programming language. Then, the guarantee establishes that if a program is correctly synchronized (according to the synchronization model), it will only exhibit sequentially consistent behaviors in the relaxed memory architecture. It is easy to observe that this definition dispenses the programmer from considering the low-level features of the relaxed memory model if she/he follows the synchronization model of the programming language.

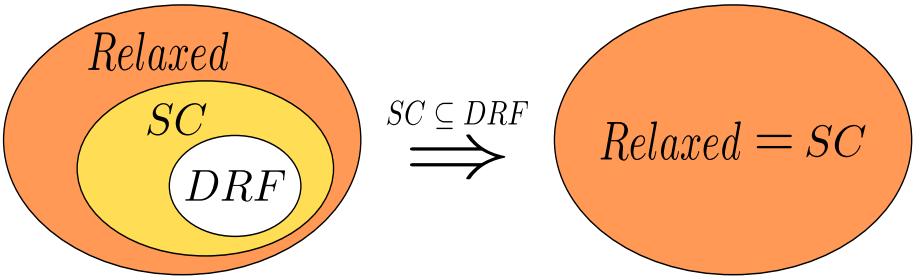


Figure 1.1: DRF Guarantee

The *new* weak ordering property, then, mandates that synchronization mechanisms at the high-level programming language have to imply synchronization at the lower-level one (that is at the level of the architecture).

In their work Adve and Hill [1990] propose the *DRF0* synchronization model. Departing from that work, systems satisfying the *new* weak ordering definition according to the DRF0 synchronization model are said to satisfy the *Data-Race-Freeness guarantee* [Manson et al., 2005], or put more pompously the *fundamental property* of relaxed memory models [Saraswat et al., 2007], indicating the importance of this property. The DRF0 synchronization model requires programs at the source (high-level) programming languages to be free of data races in their interleaving semantics. Therefore the synchronization model DRF0 implies that programs that do not have Data Races [therefore, Data Race Free (DRF)] in sequentially consistent executions of the source programming language should behave in the relaxed memory architecture in a sequentially consistent way. Hence, the programmer needs to consider only the interleaving semantics of her/his source program to guarantee that it does not have data races, and thereafter reasoning in a sequentially consistent way is sound. A graphical view of the property is given in Figure 1.1, where we consider the sets of possible computations of an arbitrary program P. On the left-hand side of the figure we can see the inclusion relation between the set of all possible executions of P in the relaxed architecture, under the label *Relaxed*, the set of executions of P that are sequentially consistent, under the label *SC*, and finally the set of executions of that P that are free of data races, under the label *DRF*. Clearly, since the DRF property is defined in terms of sequentially consistent computations, the set of computations *DRF* is included in the set *SC*. In fact, the problematic computations, in the sense that they are unexpected by the programmer, are the ones that are allowed by the relaxed semantics but are not sequentially consistent (that is the ones in $Relaxed \backslash SC$). Fortunately the DRF guarantee states that under the condition that all sequentially consistent executions are free of data races (that is $SC \subseteq DRF$) we have that the three sets collapse to a single one, as in the right-hand side of the figure. In this picture there are no problematic computations and thus, the programmer is freed from thinking about the executions that are not sequentially consistent.

In a way, one can think that the DRF guarantee transfers the importance of synchronization at the hardware level to the software level. Synchronization mechanisms at the software level can greatly vary from simple barriers (or fences) to more complex mutual exclusion mechanisms such as locks and mon-

itors, as in the case of Java, which means that many times the translation of these mechanisms to the lower-level (machine) language is far from trivial.

Another way to look at the DRF guarantee, in fact the view advocated by Adve and Hill [1990], is as a contract between the programmer and the programming environment (that includes the compiler and the hardware among other components). The idea here is that the environment guarantees that programs free of data races (for the DRF methodology) have a sequentially consistent semantics, and the programmer is compelled to write programs that are free of data races. This view of the DRF guarantee has the advantage that it enables the use of common optimization techniques for sequential programs. Compiler writers, and the architecture itself are allowed to perform the standard (sequential) optimizations provided that these do not tamper with synchronization mechanisms. The observation to make here is that to write programs that are free of data races, the synchronization mechanisms provided by the language have to be used. If optimizations do not exceed the limits (by reordering, or by speculating) of the synchronization of the program, then one has the guarantee that at most one thread at a time is accessing each piece of memory, and therefore it is safe to optimize it according to the rules of sequential programs. Indeed, one can go a step forward and consider that modifying a program by making it more synchronized will not incur in sequential consistency violations. This has been proposed for Java under the name of the "roach motel semantics" [Manson, 2004], but the Java memory model has been unfortunately shown to be unsound with respect to this kind of optimization [Ševčík, 2009]; a similar account for C and C++ Pthreads is given by Boehm [2007]. Some recent works consider which optimizations are valid under the view of the DRF guarantee [Saraswat et al., 2007; Ševčík, 2009; Burckhardt et al., 2010].

It is now widely accepted that the DRF guarantee is a good requirement for high-level programming languages. However, a problem with the DRF guarantee is that to check whether a program is DRF one must consider all possible computations of the interleaving semantics of that program. To some extent one can use data-race detection techniques [Abadi et al., 2006; Naik et al., 2006], but these techniques are not always accurate, even more so for programs using unconventional synchronization idioms.

Moreover, it is unclear whether data race freeness is a sufficient property. For instance, to implement high-level synchronization mechanisms (such as locks and monitors) in a low-level (machine) language one might have to resort to data races. If we consider an architecture in which the only available synchronization mechanisms are memory barriers (or memory fences) and compare-and-swap instructions, that do not necessarily imply barrier semantics (as it is the case in the language we will consider in Chapters 3 and 4) data races are needed to implement mutual exclusion mechanisms [Lamport, 1987]. Indeed, this kind of architecture is not arbitrary, SPARC [1994] is a good example here. It is therefore still necessary to have a definition of the semantics of programs containing data races as well, and moreover, to provide properties that guarantee the sequentially consistent execution of programs possibly with data races. We will consider these issues in Chapters 3 and 4. These topics are also studied by Alglave et al. [2010].

1.4 The Specification of Relaxed Memory Models

Not only understanding the intuitive behaviors of relaxed memory models is difficult, but also formalizing their semantics is in general problematic. This is so because the techniques used in the formalization of relaxed memory models are inherently more complex than those involved in the semantics of sequential programs. Let us briefly review the current state of memory model specifications. We can see that most memory models fit in one or more of the following categories:

- *Informal specifications*: Unfortunately, many hardware memory models descriptions fall in this category. These specification documents generally include a number of examples with depicted behaviors and some text stating whether the behavior in question is allowed or not, and why. Such example programs showing a relaxation of the memory model or a constraint imposed by it, are so called *litmus tests* and are ubiquitous in the literature of relaxed memory models starting with the early works of Collier [1992].

  Salient examples in this category are the Intel's memory ordering white paper [Intel Corporation, 2007] and AMD's memory system description [AMD, 2010], which have recently been reconsidered and formalized by Sarkar et al. [2009] and Owens et al. [2009]. We could also include ARM's memory model documentation [ARM, 2008] and Power's one [PowerPC, 2009] in this class.

  We refer to the works of Zappa Nardelli et al. [2009]; Adve and Boehm [2010] for a discussion of why this style of specification is not a good practice. Fortunately, the tendency seems to be towards more formal specifications as argued by Sewell et al. [2010].

- *Axiomatic specifications*: Most formal memory model specifications fall in this category. Unlike the previous category these specifications are mathematically sound and define in a precise way the behaviors of programs.

  In this category we find the specification of the Sparc memory model [SPARC, 1994], the Alpha's memory model [Compaq, 2002], the Java memory model [Manson et al., 2005] and the formalization of the x86 model by Sarkar et al. [2009], among others.

  This kind of specification is an enormous improvement with respect to the category we considered above, but it has some limitations. On the one hand, some axiomatic definitions of memory models are hard to understand [Adve and Boehm, 2010] and even more to practice, in the sense that knowing whether a certain behavior is possible or not becomes a daunting task [Ševčík and Aspinall, 2008], where "dedicated software" is needed to deal with these formalizations [Sarkar et al., 2009]. Moreover, to have reasonably understandable formalizations the partial orders and axioms involved have to be kept to a small number. We believe that the failure to keep these orders and axioms to the reasonably understandable is what has lead to the bugs recently found in Java [Cenciarelli et al., 2007; Ševčík and Aspinall, 2008]; notably Cenciarelli et al. [2007] report

a counter example to a theorem by Manson et al. [2005]. On the other hand, this kind of specification is generally ill suited for language-based techniques that certainly need to deal with the semantics of programs to guarantee their correctness. Indeed, axiomatic specifications of memory models do not formally consider the language, but only events representing memory operations. These reasons motivate our work on operational specifications of relaxed memory models.

A major difficulty with the formalizations of axiomatic type is that one needs to consider the validity of certain program behavior by initially *posing* a set of events and a number of relations between these events (an event structure). In some sense the notion of *computation* as a sequence of steps generated by the program (as defined by Plotkin [1981]), and even more the notion of *program* are absent of the event structure corresponding to a program behavior. For example, the only binding between the program and the posed events is given by the *program order* which is, one among many other orders required to describe a "computation" in the axiomatic sense. To verify whether a certain result is plausible one cannot simply enumerate all possible executions of the program. Instead one must propose an ad hoc candidate event structure and check that all the axioms required by the formalization are satisfied by the proposed structure. Verifying that a certain behavior is not possible requires then, proving that for all candidate instantiation of the components of the formalization at least one of the axioms does not hold. This kind of formalization is clearly *nongenerative* (in the sense of Jagadeesan et al. [2010]), which as we will see soon limitates its usability.

We will illustrate the kind of reasoning required to deal with axiomatic formalizations with a small example below.

- *Operational specifications*: Indeed, it is the thesis of our work that relaxed memory model formalizations should be operational.

  Specifications in this category are provided in terms of abstract state machines [Shen et al., 1999; Saraswat, 2004], rewriting systems [Boudol and Petri, 2009, 2010] following the style advocated by Plotkin [1981] or even event structures [Cenciarelli et al., 1999].

  There are not many memory models formalized in this style. Attempts to provide operational descriptions for relaxed memory models can be found by Shen et al. [1999], and for the particular case of Java we find the works of Saraswat [2004]; Cenciarelli et al. [1999] previous to the revision of the Java memory model and by Cenciarelli et al. [2007]; Jagadeesan et al. [2010] for the current specification of the Java memory model. We report [Cenciarelli and Knapp, 2007] that serious problems have been identified in the work of Cenciarelli et al. [2007]. The proposed formalization by Jagadeesan et al. [2010] is an overapproximation of the current Java memory model [Manson et al., 2005], which is justified due to its complexity. The recent C++ memory model [Boehm and Adve, 2008] belongs to this kind as well. However, the C++ prescribes sequentially consistent semantics for DRF programs, and it does leave the semantics of programs with data races undefined. In that sense we can consider that the C++ memory model is trivially operational, relying on the DRF guarantee. Fi-

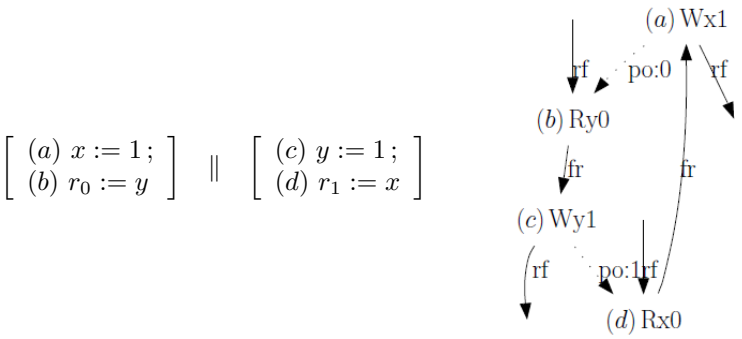


Figure 1.2: Axiomatic characterization of example 1.2.

nally, Owens et al. [2009] and Sewell et al. [2010] present an operational formalization that corresponds to their axiomatic model of the x86 architecture. Their formalization is very similar to the one presented by Boudol and Petri [2009] and can be seen as a particular case of this last one.

In our view there are several advantages in having an operational formalization, as opposed to an axiomatic one. To start with, we consider that this kind of formalization is easier to understand and therefore a better tool for the programmer. Moreover, these models define the possible executions of the program, which could enable the use of techniques such as model checking [Clarke et al., 1999] as considered by Atig et al. [2010]. Another advantage is that this approach enables standard language-based techniques, which in general are proved correct by means of structural induction principles as proposed by Plotkin [1981] and Felleisen and Hieb [1992]. This last advantage is not a minor one since it opens the door to the use of standard static analysis techniques such as type-systems, something we will exploit in future works.

To emphasize the importance of operational descriptions of memory models let us do a simple exercise. Let us consider the axiomatic formalization of Example 1.2 from Section 1.1 in light of the formalization by Alglave et al. [2010]. The example is reproduced in Figure 1.2 with references p and q replaced by x and y respectively to comply with the example as it is presented by Alglave et al. [2010]. Moreover, we have annotated the instructions to the left with letters to represent the events generated by each of them. Thus (a) means the event generated by the execution of the write of x in the left-hand thread and so on. To justify that the result $r_0 = r_1 = 0$ is possible we have, as required by axiomatic models, to pose a candidate execution. In the work of Alglave et al. [2010] a candidate execution X has the following shape:

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

where $\mathbb{E}$ is the set of events, in our case simply the set $\{(a), (b), (c), (d)\}$; $\xrightarrow{po}$ is the relation conveying the program order among the events of $\mathbb{E}$, in our case $(a) \xrightarrow{po} (b)$ and $(c) \xrightarrow{po} (d)$. We refrain from explaining every single relation required in this execution witness, referring the reader to [Alglave et al., 2010] for more details. Suffice it to say that the relation $\xrightarrow{dp}$, and $\xrightarrow{ws}$ are empty for the example in question. And finally, the relation $\xrightarrow{rf}$ standing for *reads-from*

indicates which write event fulfills a certain read event; in other words $w \xrightarrow{rf} r$ means that the event r obtains the value written by the event w. Since in our case both reads see the initial value for references x and y we can say that $\xrightarrow{rf}$ is also empty. So far we have an execution candidate, it remains to be seen whether this execution candidate is valid according to the axioms of the memory model. Let us be more abstract from this point on. Indeed, to be to able consider the axioms one has to build the orders $\xrightarrow{fr}$, standing for *from-read*, $\xrightarrow{ghb}$ standing for *global-happens-before* and numerous other ordering relations among events, that we refrain from detailing here. Once all these orders are constructed one can verify the axioms. These axioms, which we will not state, anticipating that they all hold for the example in question include: $\mathsf{uniproc}(\mathsf{X})$ guaranteeing the consistency of the happens-before relation with the individual program order of each thread, by checking the acyclicity of the transitive closure of the union of these orders; $\mathsf{thin}(\mathsf{X})$ requiring that values read can be justified by a read in the computation (cf. thin-air reads in the Java memory model), among many others. Once more, we invite the reader to check the paper by Alglave et al. [2010] for further details. A figure, extracted from that work [Alglave et al., 2010] showing the events and the different orders among them can be found at the right of Figure 1.2. It should be noticed here that we are not arguing against the formalization by Alglave et al. [2010], which constitutes a titanic effort to formalize and clarify existing relaxed memory models, but we intend to suggest that axiomatic formalizations can, and in general do, become very quickly unmanageable for nonexperts. Moreover, axiomatic formalizations do not immediately support language-based techniques for the analysis and verification of parallel programs. On the other hand operational specifications are much more tractable in the cases where one can find a good machine abstraction. The example above is trivial to explain with a semantics with write-buffers as we considered in Example 1.6.

However, providing an operational formalization for relaxed memory models is not a straightforward task due to the fact that standard techniques for specifying the semantics of sequential programming languages do not immediately translate to relaxed memory models. We already mentioned that sequential consistency is the only semantics that is an immediate generalization of the semantics of sequential programs. It is the subject of this thesis to present new techniques for the operational formalization of relaxed memory models.

Before we conclude this introduction to the field of relaxed memory models we would like to express an opinion. It is beyond doubt that the data-race-freeness guarantee is a fundamental property for the programmability of relaxed memory models. Some programming languages, like Ada [Ledgard, 1983], and the more recent C++ memory model proposal [Boehm and Adve, 2008] consider that programs with data-races are erroneous and then, these languages provide interleaving semantics (standing on the data-race-free guarantee) for programs without errors and no semantics for programs with data-races. This is also the approach taken by Reynolds [2004] for the formalization of the semantics of parallel programs. The reasons, at least in the case of C++, for taking this approach are deep and well justified. This approach has also been considered for Java, but to preserve Java's safety guarantees the model had to be extended to programs containing data races as well. However, the problem with this approach is that even if data-races are errors, most programming languages do

not offer the capability of detecting them at runtime without incurring in high performance penalties. Moreover, programs with such errors many times do not fail to produce a result; hence, a possibly unnoticed erroneous result. On the other hand, from the point of view of modularity, it is not always possible to know whether a call to library code is data-race-free, which could compromise the correctness of the whole program, since incurring in a data-race renders the semantics of the program undefined. We think that the memory model specification for C++ by Boehm and Adve [2008] is a great step forward towards having a formal and dependable formalization of parallel C++ , but we feel that the relaxed memory models community has to address the issue of programs with data-races in a more satisfactory way as discussed in the works by Adve and Boehm [2010] and Boehm [2009]. We cannot conclude without mentioning some interesting research in the direction of finding data-races at runtime [Adve et al., 1991] and raising exceptions [Marino et al., 2010], and detecting data-races statically [Abadi et al., 2006; Naik et al., 2006].

1.5 Summary of Contributions

The main contributions of this thesis are presented in the following three chapters and can be summarized as follows:

- In Chapter 2 we present an *operational semantics* for relaxed memory models with *write-buffering* capabilities. The novelty of our approach is that we describe the semantics by means of a standard operational semantics for an imperative core-ML programming language extended with a dynamic thread creation construct and high-level locks for mutual exclusion. The semantics we provide is simple and covers many of the behaviors present in realistic memory models, like the Total Store Ordering (TSO) and the Partial Store Ordering (PSO) memory models of SPARC [1994].

  Our semantics can be proved sound with respect to the standard Data Race Freeness guarantee that we have discussed before. We prove this result using standard programming languages techniques introduced by Berry and Lévy [1979]; Lévy [1980] for the λ-calculus. These techniques are pervasive throughout this thesis. The proof of the DRF guarantee that we present here is interesting not only for its implications on programmability, but also because it is a good exercise for our formalization techniques, making use of well established theories on programming languages semantics.

  It should be noticed that this chapter is essentially the work presented in [Boudol and Petri, 2009].

- In Chapter 3 we present an operational approach for *speculative computation* for two variations of the language of Chapter 2: one including locks and the other providing only barriers and a compare-and-swap instruction for handling synchronization. Speculative computations are modeled in this semantics by generalizing the standard evaluation contexts by Felleisen and Hieb [1992] to permit computing "ahead of time". As part of our formalization we define *valid* speculations, that in essence correspond to computations where each thread produces a trace that is

"equivalent" (according to a definition similar to the equivalence by permutations by Berry and Lévy [1979]) to a sequential execution of the thread.

Indeed, the speculative techniques we consider are highly motivated by common relaxed memory model relaxations. We are able, by means of this speculative semantics, to capture program relaxations that were not captured by the model of Chapter 2.

An important property of the program speculations treated in this chapter is that the standard data race free guarantee does not hold for the semantics we consider. We propose a different property for the semantics of the language with locks, and we prove that this "robustness" property is enough to guarantee that the results of computations of the speculative framework are those obtained from sequentially consistent executions. Interestingly, for the language without locks, we observe that the standard data race free guarantee does not help, nor does the property discussed for the language with locks. We introduce an alternative property which is based on the reorderings appearing in valid computations. Enforcing this property requires the use of barriers.

This chapter is based on our previous work [Boudol and Petri, 2010] but has significant differences with respect to that work, which will be discussed in more detail in the chapter.

- Finally in Chapter 4 we instantiate the formalizations of Chapters 2 and 3 to the memory models of the Sparc family [SPARC, 1994]. In particular, the TSO and PSO memory models can be formalized in both the framework of write-buffers, and the one of speculations. We therefore provide a formal proof that both formalizations are equivalent, that is, departing from a computation in one of the formalizations we can find one in the other formalization that has the exact same behavior. This proof provides an indication that the formalization of speculative computations of Chapter 3 is more general than the one with write-buffers of Chapter 2. In particular, the RMO memory models can be formalized with the first and not with the latter.

  The technical contents of this chapter have not yet been published.

In order to preserve the homogeneity of this thesis we have not included other previous works [Huisman and Petri, 2007, 2008] regarding the Java Memory Model [Manson et al., 2005]. However, it must be acknowledged that those works strongly influenced the views and results presented in this thesis.

To simplify the presentation and the comparison of the different formalizations we use several times the same examples, many of which are standard in the literature. To ease the readability we have included in Appendix A a compendium of all the examples of relaxations appearing in the thesis with references to the pages where they are discussed.

Chapter 2

An Operational Formalization of Relaxed Memory Models: Write Buffers

In the previous chapter (1) we have argued that there is a significant mismatch between the expected interleaving semantics of parallel programs and their behaviors when running under relaxed memory model architectures. Providing a comprehensible operational model for relaxed memory architectures is the topic of this chapter. We will formalize here by means of standard programming languages techniques, an operational semantics of a high-level programming language that supports simple, but powerful relaxations with respect to its sequentially consistent semantics.

The particular relaxations that we will consider here are introduced by *write-buffering* [Dubois et al., 1998]: a technique that reduces the performance penalties induced by the latency of storing data into the main memory by delaying its completion and continuing, in an asynchronous fashion, with subsequent actions. We will see that many of the standard relaxations present in common memory models can be explained through this simple technique.

With the support of this operational semantics, we will prove a standard property of relaxed memory models, namely the "fundamental property of relaxed memory models" [Saraswat et al., 2007] that we have mentioned in the introduction of the thesis. It might be worth repeating here that this property serves as a correctness criterion for relaxed memory models, and at the same time facilitates the programmability of architectures providing such models. Moreover, the proof that we present of this result is not only interesting for its implications, but also for the techniques used. Formalizing the semantics by means of techniques that are standard in the programming languages literature makes it possible to reuse well established theories of concurrency and programming languages. To be concrete, we will support our developments by adapting concepts of the *true concurrency* and the λ-calculus literature, and the core of the proof is a simple bisimulation. These concepts should not be particularly

surprising for readers acquainted with programming languages semantics; and yet, they seldom appear in the literature of relaxed memory models.

Write Buffering Let us now provide a brief introduction to write buffering, and its semantical implications. Accesses to the memory have been early identified as an important bottleneck for the performance of programs that make heavy use of the memory. The penalties induced by accessing the memory can be greatly reduced by – relatively simple – hardware optimization techniques such as caching, pipelining and the buffering of accesses to the memory [Briggs, 1979; Kroft, 1981]. In general these optimizing techniques are innocuous – in the sense that they do not alter the semantics – for sequential programs. However, as identified by Collier in his seminal work [Collier, 1992], the rules governing the execution of parallel programs are modified by adding these optimizations. In other words, the semantics of parallel programs can diverge from the intuitive *interleaving semantics* as a side effect of adding such optimizations. In some sense, we can think that for parallel programs, these optimizations trade performance for simplicity of programming. Then some questions arise naturally: Which are the exact behaviors that are introduced by adding such techniques? And, how can the programmer still guarantee that the execution of parallel programs will not run into unexpected errors? Clearly, semantical techniques other than the usual interleaving semantics are required to answer these questions.

Dubois, Scheurich and Briggs, in [Dubois et al., 1998], were among the first to investigate the semantical effects of buffering memory accesses in multiprocessors. In that work they first identify programs whose execution differs from their *expected* sequentially consistent semantics when allowing some of these optimizations. Once these *hazards* are identified they proceed to propose conditions *on programs* that suffice to guarantee that these corner cases will not arrive during the execution; hence, that the programs behave as prescribed by their sequentially consistent semantics. We take the work of Dubois et. al. as a starting point to develop an operational semantics[1] that accounts for the effects of write-buffering – in the sense of [Dubois et al., 1998]. However we will consider a different property to guarantee the sequentially consistent execution of parallel programs following [Adve and Hill, 1990; Gharachorloo et al., 1990]; namely, the *absence of data races* (that is concurrent conflicting accesses to the same reference) in their interleaving semantics. We consider the proof of this property as an interesting checkpoint of our general approach.

To motivate the topic of write buffering it might be of interest to consider a classical (see [Adve and Gharachorloo, 1996]), and critical example of a failure to provide sequentially consistent semantics, we are talking about Dekker's early mutual exclusion algorithm (of Example 1.1):

$$\left[\begin{array}{l} \texttt{flag}_0 := \texttt{false}\,; \\ \texttt{if flag}_1 \texttt{ then} \\ \qquad \textit{critical section 0} \end{array}\right] \quad \| \quad \left[\begin{array}{l} \texttt{flag}_1 := \texttt{false}\,; \\ \texttt{if flag}_0 \texttt{ then} \\ \qquad \textit{critical section 1} \end{array}\right]$$

If we consider that initially the memory locations $\texttt{flag}_0$ and $\texttt{flag}_0$ are set to `true` the assignement to the flag by each of the threads communicates to the other thread its intention to engage in its critical section. Notice that a thread

[1] Knowledge of [Dubois et al., 1998] is not required to comprehend the contributions of our work.

enters its critical section only after verifying that the other one did not signal its intention to do so as well. It is easy to check that in all sequentially consistent executions of this code, at most one of the threads reaches its critical section. However, relaxing the execution order of writes w.r.t. subsequent (in the sense of the program text) reads permits reading the flag, for instance $\texttt{flag}_1$ for the thread on the left, before the effects of its previous write, to $\texttt{flag}_0$ in this case, have been made visible to the other thread. Then both threads can reach their critical sections; exactly the behavior the algorithm is meant to avoid. One possible explanation for the reordering of these events, and the one on which this chapter is based, is the presence of hardware that delays the writes to the flags instead of performing them immediately.

Another interesting example (also presented in [Adve and Gharachorloo, 1996] and given in 1.6) is the producer-consumer algorithm (also known as safe publication):

$$\left[\begin{array}{l} \texttt{data} := 1\,; \\ \texttt{flag} := \texttt{true} \end{array}\right] \;\|\; \left[\begin{array}{l} \mathsf{while\ not}\ \texttt{flag}\ \mathsf{do\ skip}\,; \\ \texttt{r} := \texttt{data} \end{array}\right]$$

Here, the thread on the left can obtain an outdated value of `data` if the update to `flag` (initially assumed to be `false`) reaches the right thread before the update to `data`. As we already suggested in the previous chapter, synchronization mechanisms can prescribe this reordering, and guarantee that events are made visible in a way that overcomes these problems. We will come back to this point later.

To describe the behaviors of programs under write-buffering we need to have a syntactic representation of buffers. However a clarification is due: in this work we will treat write-buffers as a mere formalization artifact, *not* necessarily reflecting *real hardware* architectures. Indeed, the effects introduced by write-buffering are common to many hardware (or software) optimizations, such as caching, instruction pipelining, the buffering of memory accesses and the reordering of instructions performed at compilation just to mention some. We are clearly inspired by the write-buffering technique, but we do not pretend model the exact hardware in our semantics; only its behavior.

Let us move on by making the notion of write-buffer more concrete: by write-buffers we mean mappings from memory locations (or references) to sequences (with FIFO semantics) of values written to the memory that are pending to be updated[2]. Formally, a buffer B is a mapping of the form:

$$B = \{p_1 \mapsto v_1^1 \dots v_{n_1}^1, \dots, p_k \mapsto v_1^k \dots v_{n_k}^k\}$$

where p_i is a memory location and v_j^i is the j-th value that is pending for update for reference p_i. We will represent parallel programs in runtime as trees containing the threads in the leaves and buffers in the internal nodes. The syntax for thread systems containing write buffers is given by:

$$T \;::=\; e \;\mid\; \langle B \rangle T \;\mid\; (T \| T')$$

where e is an expression of the programming language which represents a thread of execution. The idea here is that buffers hold values of writes issued by any of

[2]There are other possible interpretations of write-buffers. We will considered a different one in subsequent chapters.

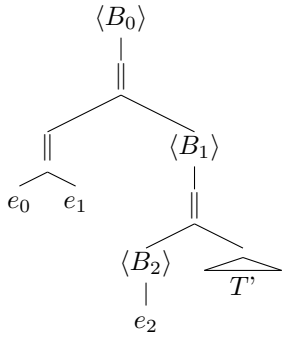


Figure 2.1: $T = \langle B_0 \rangle((e_0 \| e_1) \| \langle B_1 \rangle(\langle B_2 \rangle e_2 \| T'))$

the threads that lie under it (in the tree where the root is at the top). One can think of these thread systems as descending trees with unary nodes containing buffers, branching nodes separating parallel sub-systems and threads as the leaves. Thus, the thread system $T = \langle B_0 \rangle((e_0 \| e_1) \| \langle B_1 \rangle(\langle B_2 \rangle e_2 \| T'))$, where e_i represents an arbitrary thread expression and B_i represents an arbitrary buffer, can be depicted as in Figure 2.1.

As we said before, the introduction of write-buffers changes the semantics of writing and reading into the memory. Now a thread writing on a reference p – as it would result from executing the expression $(p := v)$ – creates a new buffer containing the value v for reference p, rather than directly updating the contents of the main memory. Later in the execution of the program, this buffered update will be lazily propagated towards the main memory. We will make this intuition formal in the semantic rules that follow. Importantly, to preserve data dependencies present in the program code, reading a reference p – the result of reducing the expression $(!p)$ in ML's notation – must acknowledge the effects of previous writes to p by the same thread. We will choose here to allow reads to obtain the last-in *pending* value in the buffers for that reference. If there is no such pending value in the buffers the value is retrieved from the memory. An alternative approach would be to delay the execution of reads until there are no pending writes on the same reference; this resembles the semantics of the IBM 370 architecture according to [Adve and Gharachorloo, 1996]. We chose here the former approach as it is more general, and has interesting semantical implications that the latter does not have. Then in our formalization reading a reference p does not constrain pending writes on references other than p.

To better understand this formalization of write buffers let us consider some classical examples. As a side note on conventions, many of the examples we will consider here are ubiquitous in the literature of relaxed memory models, a tradition probably initiated by the work of Collier [Collier, 1992]. These examples are globally know as *litmus tests*, and their intention is to prove a particular relaxation of a memory model w.r.t. sequential executions of the test. We will follow this tradition by considering these examples as they are presented in the literature whenever possible. In particular, we will denote by r_i references that are in general considered to be local to a thread (in particular registers). In our language we do not deal with registers, but we will consistently use references named r_i in a single thread to conform to that rule. Let us now proceed with the examples.

Since reads retrieve values of pending writes in the buffers we can see that in the example that follows (introduced in Example 1.2) – where we assume that the initial values of p and q are 0:

Example 2.1 (Write Read Reordering)**.**

$$\begin{bmatrix} p := 1\,; \\ r_0 := (!q) \end{bmatrix} \quad \| \quad \begin{bmatrix} q := 1\,; \\ r_1 := (!p) \end{bmatrix}$$

Both reads can result in 0 values at the end of the execution, obtaining as a final result $r_0 = r_1 = 0$, which clearly disagrees with the standard interleaving semantics of the program. To show how this could happen it suffices to execute the writes of p and q first obtaining the following trace (where we include the buffers to the left of each thread):

$$\begin{array}{rlcl} & \emptyset\ \left[p := 1\,; r_0 := (!q)\right] & \| & \emptyset\ \left[q := 1\,; r_1 := (!p)\right] \\ \xrightarrow{*} & \langle [p \leftarrow 1] \rangle\ \left[r_0 := (!q)\right] & \| & \langle [q \leftarrow 1] \rangle\ \left[r_1 := (!p)\right] \\ \xrightarrow{*} & \langle [p \leftarrow 1] \rangle\ \left[r_0 := 0\right] & \| & \langle [q \leftarrow 1] \rangle\ \left[r_1 := 0\right] \end{array}$$

in which we have an intermediate configuration where the two writes are buffered and the reading redexes are enabled. Since no pending updates on q is present in the buffer of the left thread, the read returns the initial value in the memory, and similarly for the thread on the right and reference p reaching the third configuration in the trace. This kind of behavior is sometimes described as a relaxation of the *write to read* order, following [Adve and Gharachorloo, 1996], and symbolically denoted by $\mathbf{W} \to \mathbf{R}$.

Another significant relaxation provided by the kind of buffers considered here is the reordering of writes, which allows writes to different references – obviously reordering writes on the same reference would violate the sequential semantics – to reach the memory in a different order than the one imposed by the program. This characteristic is called the *write to write* ordering relaxation in [Adve and Gharachorloo, 1996], and it is denoted by $\mathbf{W} \to \mathbf{W}$. Let us illustrate it with the Example 1.3 of the previous section.

Example 2.2 (Write Write Reordering)**.**

$$\begin{bmatrix} p := 1\,; \\ q := 1 \end{bmatrix} \quad \| \quad \begin{bmatrix} r_0 := (!q)\,; \\ r_1 := (!p) \end{bmatrix}$$

Thus, in the example above a possible result (assuming in the initial memory $p = q = 0$) is $r_0 = 1$ and $r_1 = 0$, which can happen if the buffer update of q reaches the memory before that of p. The capability of reordering writes to different references in the buffers is called *jockeying* in [Dubois et al., 1998].

To finish with examples of the relaxations of this chapter let us consider the following one.

Example 2.3 (Read Own Write Early)**.**

$$\begin{bmatrix} p := 1\,; \\ r_0 := (!p)\,; \\ r_1 := (!q) \end{bmatrix} \quad \| \quad \begin{bmatrix} q := 1\,; \\ r_2 := (!q)\,; \\ r_3 := (!p) \end{bmatrix}$$

As usual, we take the initial memory to contain a 0 for every reference. This example is slightly more intricate than the previous ones. One can observe that the first write and the first read of each thread are on the same reference; which makes them conflicting actions, and thus are not affected by the $\mathbf{W} \to \mathbf{R}$ relaxation we discussed before. We have thus far not talked about relaxing the order of reads w.r.t. subsequent reads, which indeed is not a behavior introduced by write buffering alone. So one might ask, considering that no read to read reordering is available: is $r_0 = r_2 = 1$ and $r_1 = r_3 = 0$ a possible result of this program? It could be surprising to see that the behavior is actually possible. The key insight here is that, as we briefly mentioned, reads are allowed to retrieve their contents from buffers yet to be updated. Thus, the first read of each thread is "forced", by the sequential dependencies of the program, to read its previous update to the same reference. However, these updates need not be visible to the other thread at the time of the second read, and thus a value of 0 can be returned for the second read. This relaxation w.r.t. the sequentially consistent semantics cannot be simply stated in terms of reordering of actions of one kind w.r.t. actions of another (or the same) kind. In [Adve and Gharachorloo, 1996] this relaxation is know as the capability to *read own writes early* and it is common to many memory models [SPARC, 1994; Intel Corporation, 2007; AMD, 2010; PowerPC, 2009; ARM, 2008].

Let us conclude this introduction by providing a small summary of the relaxations allowed by write-buffering as considered here, and by providing a brief description of our approach. We will in the following sections present a core high-level parallel programming language and two different operational semantics for it. The former is the standard interleaving semantics of the language, the later one includes the write-buffers we have introduced above. The semantics with write-buffers allows for many standard memory model relaxations, in particular, it allows for write read reordering ($\mathbf{W} \to \mathbf{R}$), it allows for write write reordering ($\mathbf{W} \to \mathbf{W}$) and it allows thread to see their *own writes early*. With the formalization of these two semantics at hand, we will be able to prove that our semantics of write-buffers supports the standard data race freeness guarantee of relaxed memory models.

2.1 The Language

To highlight the important aspects of relaxed memory models without obfuscating the technical definitions with unnecessary constructs we have chosen a simple language close to core ML; that is a call-by-value imperative λ-calculus augmented with two concurrency constructs; namely, thread creation and a block-based locking construct. The syntax of the language is the following:

$$
\begin{array}{rcll}
v & ::= & x \mid \lambda x e \mid \mathit{tt} \mid \mathit{ff} \mid () & \textit{values} \\
e & ::= & v \mid (e_0 e_1) \mid (\mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2) & \textit{expressions} \\
 & \mid & (\mathsf{ref}\, e) \mid (!\, e) \mid (e_0 := e_1) & \\
 & \mid & (\mathsf{thread}\, e) \mid (\mathsf{with}\ \ell\ \mathsf{do}\ e) &
\end{array}
$$

where x is a variable belonging to the set $\mathcal{V}ar$, constants tt and ff correspond to boolean values – which for simplicity are the only values available at the source language. We will generically denote the set of values of the language

by $\mathcal{V}al$. Functions are defined by means of the λxe, which binds the variable x in the expression e. Expressions will be regarded up to α-conversion, i.e. up to the renaming of bound variables. We denote by $e\{x \mapsto e'\}$ the capture avoiding substitution of the free occurrences of variable x in e by the expression e'. We shall use the notation ($\mathsf{let}\ x = e_0\ \mathsf{in}\ e_1$) which is syntactic sugar for the expression ($\lambda x e_1 e_0$), which will also be denoted by $e_0\,;e_1$ whenever x does not occur free in e_1. The expressions of the language are very standard comprising applications, conditionals and imperative constructs for: reference creation ($\mathsf{ref}\,e$), dereferencing ($\mathbf{!}\,e$) and memory update ($e_0 := e_1$). We have also included a construct for creating parallel threads ($\mathsf{thread}\,e$) and a block-based synchronization construct ($\mathsf{with}\ \ell\ \mathsf{do}\ e$), where ℓ is taken from an infinite set $\mathcal{L}ocks$ of locks, and whose intended semantics is to exclusively acquire the lock ℓ, reduce the expression e and release the exclusive access on ℓ.

One can observe that we have not included syntax for recursion. This is not a weakness of our approach since adding it would not pose any difficulties. We do not add it here simply for perspicuity.

In order to describe the operational semantics of the language we need to introduce some run-time values and expressions. Firstly, to manage the memory we introduce the set $\mathcal{R}ef$ of *reference* names, ranged over by p, q, ..., which are values in the run-time language. We assume that $\mathcal{R}ef$ is a disjoint set from $\mathcal{L}ocks$, and represents the set of references in the memory (also called memory locations, addresses, or pointers). Secondly, we need to introduce the expression $(e\backslash\ell)$ that represents that the expression e is being evaluated while holding the lock ℓ. Thus the run-time language becomes:

$$\begin{array}{rcll} p,\ q\ldots & \in & \mathcal{R}ef & \textit{references} \\ v & ::= & \cdots \mid p & \textit{run-time values} \\ e & ::= & \cdots \mid (e\backslash\ell) & \textit{run-time expressions} \end{array}$$

where we use dots to avoid repeating the productions that were presented before.

We follow the approach of [Felleisen and Hieb, 1992] to describe the transitions of the *call-by-value* semantics of our language, by decoupling the run-time expression e into an *evaluation context*, defined by the grammar $\mathbf{E}$ below, and a redex to be reduced, defined by r below. Evaluation contexts are just expressions of the language where some subexpression has been replaced by a hole (denoted by []). As it is standard, we denote by $\mathbf{E}[e]$ the expression resulting from filling the hole in $\mathbf{E}$ with the expression e.

$$\begin{array}{rcll} \mathbf{E} & ::= & [] \mid (\mathbf{E}\,e) \mid (v\,\mathbf{E}) & \textit{evaluation contexts} \\ & \mid & (\mathsf{ref}\,\mathbf{E}) \mid (\mathbf{!}\,\mathbf{E}) \mid (\mathbf{E} := e) \mid (v := \mathbf{E}) & \\ & \mid & (\mathsf{if}\ \mathbf{E}\ \mathsf{then}\ e_0\ \mathsf{else}\ e_1) \mid (\mathbf{E}\backslash\ell) & \\ r & ::= & (\lambda x e v) \mid (\mathsf{ref}\,v) \mid (\mathbf{!}\,p) \mid (p := v) & \textit{redexes} \\ & \mid & (\mathsf{if}\ v\ \mathsf{then}\ e_0\ \mathsf{else}\ e_1) \mid (\mathsf{thread}\,e) & \\ & \mid & (\mathsf{with}\ \ell\ \mathsf{do}\ e) \mid (v\backslash\ell) & \end{array}$$

Clearly, there are expressions that represent errors in the computation, for example dereferencing a value other than a reference is an erroneous situation. This is expressed in the following definition:

Definition 2.4 (Faulty expression)**.** *We say an expression e is* faulty *if it contains a subexpression which has any of the following forms:*

- $e = (ve')$*, where v is not a functional value (i.e. v does not have the form $(\lambda x e'')$, for every x and e''), or*
- $e = (!v)$ *or* $e = (v := e')$*, where $v \notin \mathcal{R}ef$, or*
- $e = (\mathsf{if}\ v\ \mathsf{then}\ e_0\ \mathsf{else}\ e_1)$*, where v is not a boolean value (i.e. $v \notin \{tt, ff\}$).*

An important property of evaluation contexts is that: every nonfaulty expression is either a value or it can be uniquely decomposed into an evaluation context and a redex. The evaluation contexts we have defined here impose a *call-by-value* reduction strategy for our language. We establish the above intuition formally in the following lemma:

Lemma 2.5. *For any expression e of the run-time language, either e is a value, or there are unique $\mathbf{E}$, an evaluation context, and e', a redex or a faulty expression, such that $e = \mathbf{E}[e']$.*

Proof. By induction on the expression e. □

2.2 The Semantics

We will define now the call-by-value semantics – which we will regard as the *reference semantics* – of our language. This semantics will be presented in two phases: in a first phase we define the semantics of expressions, which correspond to the execution of single threads; in a second phase we introduce global configurations, that coordinate in a coherent way the steps of the different threads and their relation with the global store and state of locks. In fact, actions β, $\swarrow$ and $\searrow$ are not necessary to coordinate different threads, since these actions are effect free; however we shall have a use for them in the sequel. This style of presenting the semantics in two stages will be recurrent along this document. To be able to compose different threads in the global semantics we will need to know which is the action being performed in each step taken by the thread. Hence, we use labeled transitions to describe the semantics of single threads, where labels represent the action being taken. The set $\mathcal{A}ct$ of actions is given by:

$$a \in \mathcal{A}ct \quad ::= \quad \beta \mid \swarrow \mid \searrow \mid \nu_{p,v} \mid \mathsf{rd}_{p,v} \mid \mathsf{wr}_{p,v} \mid \mathsf{spw}_e \mid \overset{\frown}{\ell} \mid \overset{\frown}{\ell}$$

where e is a closed expression in the action spw_e. Then, reducing an application generates a β labeled transition; taking the `then` branch of a conditional generates the $\swarrow$ action, and taking the `else` branch produces $\searrow$; creating a new reference p with a value v – assumed to be closed – is denoted by $\nu_{p,v}$; reading the value v from the reference p is denoted by $\mathsf{rd}_{p,v}$, and similarly writing is denoted by $\mathsf{wr}_{p,v}$ – here, again, we assume the value v to be closed –; spawning a new thread to execute closed expression e generates the action spw_e; finally, acquiring the lock ℓ is denoted $\overset{\frown}{\ell}$ and, symmetrically, releasing it is denoted by $\overset{\frown}{\ell}$.

$$
\begin{array}{rcl}
\mathbf{E}[(\lambda x e v)] & \xrightarrow{\beta} & \mathbf{E}[\{x \mapsto v\}e] \\
\mathbf{E}[(\mathsf{if}\ tt\ \mathsf{then}\ e_0\ \mathsf{else}\ e_1)] & \xrightarrow{\swarrow} & \mathbf{E}[e_0] \\
\mathbf{E}[(\mathsf{if}\ ff\ \mathsf{then}\ e_0\ \mathsf{else}\ e_1)] & \xrightarrow{\searrow} & \mathbf{E}[e_1] \\
\mathbf{E}[(\mathsf{ref}\, v)] & \xrightarrow{\nu_{p,v}} & \mathbf{E}[p] \\
\mathbf{E}[(p := v)] & \xrightarrow{\mathsf{wr}_{p,v}} & \mathbf{E}[()] \\
\mathbf{E}[(!\, p)] & \xrightarrow{\mathsf{rd}_{p,v}} & \mathbf{E}[v] \\
\mathbf{E}[(\mathsf{thread}\, e)] & \xrightarrow{\mathsf{spw}_e} & \mathbf{E}[()] \\
\mathbf{E}[(\mathsf{with}\ \ell\ \mathsf{do}\ e)] & \xrightarrow{\widehat{\ell}} & \mathbf{E}[(e \backslash \ell)] \\
\mathbf{E}[(v \backslash \ell)] & \xrightarrow{\overset{\frown}{\ell}} & \mathbf{E}[v]
\end{array}
$$

Figure 2.2: Single expression operational semantics

2.2.1 The Semantics of Single Expressions

The full semantics of expressions (or single threads) is presented in Figure 2.2. The most salient aspects of these transitions are the rules for reading and acquiring locks. In the first case one can notice that the value read (namely v in the action $\mathsf{rd}_{p,v}$) is not necessarily present in the left-hand side of the production; we can consider the value v as being guessed, we will later see that it must agree with the contents of the store at the time the action happens in the global semantics. Similarly, in the case of acquiring the lock ℓ (in the action $\widehat{\ell}$) the lock is guessed to be free; accordingly we will check this condition when composing threads in the global semantics.

2.2.2 The Global Semantics

To describe the transitions of multithreaded programs we need to compose the semantics of single threads, as given previously, into a single *configuration* that includes the threads, the store and the state of the locks involved in the program. Let us start by defining the configurations of the run-time semantics. Configurations have the form:

$$C = (S, L, T) \qquad \textit{strong configurations}$$

which comprises: a *store* – also called the *memory* – $S : \mathcal{R}ef \rightharpoonup \mathcal{V}al$, which is a mapping from a finite set $\mathsf{dom}(S)$ of references to values, representing the contents of the main memory; a set of locks L, representing the locks currently held by some thread in the configuration; and finally a thread system T, which is generated by the syntax:

$$T \quad ::= \quad e \ \mid\ (T \| T') \qquad \textit{strong thread systems}$$

where we denote by $(T\|T')$ the parallel composition of the threads in T and T'. In what follows we will qualify these configurations as *strong* to distinguish

them from another kind of configurations that will be presented in Section 2.4. For reasons that will be established later, we do *not* assume here that parallel composition is commutative nor associative.

As usual, we shall consider only *well-formed* configurations, meaning that any reference that occurs somewhere in the configuration belongs to the domain of the store, that is, it is bound to a value in the memory – we shall not define this property, which is preserved in the operational semantics, more formally. For instance, if e is an expression of the source language, the initial configuration $(\emptyset, \emptyset, e)$ is well-formed.

To focus on a certain thread we introduce thread system contexts, generated by the following definition:

$$\mathbf{T} \quad ::= \quad [] \;\mid\; (\mathbf{T} \| T) \;\mid\; (T \| \mathbf{T})$$

which represents a thread system with a hole []. As the reader might expect the notation $\mathbf{T}[T]$ represents the thread system context $\mathbf{T}$ where the hole has been replaced by the thread system T.

In what follows, we will not only need to know *what* action is being taken, but also *where* in the thread system the action happens. For that purpose we adopt the notion of *occurrence*. These occurrences are unique identifiers of subtrees in the thread system. The set $\mathcal{O}cc$ of occurrences contains sequences of symbols $\leftarrow$ and $\rightarrow$, meaning the left and the right subtrees in a parallel composition. We will use the variable *occ* or simply o to range over the set $\mathcal{O}cc$. We shall use various kinds of *sequences* in the following, which we collectively denote by σ, $\xi \ldots$ (and later also γ), and therefore we fix a few notations regarding sequences.

Notation 2.6. *We denote by ε the empty sequence, and the concatenation of the sequence σ' after the sequence σ is denoted $\sigma \cdot \sigma'$. The prefix ordering is denoted $\leq$, that is, $\sigma \leq \sigma'$ if $\sigma' = \sigma \cdot \sigma''$ for some σ''. The length of σ is $|\sigma|$.*

For each thread system T and occurrence *occ*, we define the subsystem (subtree) T/occ of T at occurrence *occ* – if this is indeed an occurrence of a subtree –, in the obvious way, that is:

$$\begin{aligned} T/\varepsilon &= T \\ (T\|T')/\leftarrow \cdot occ &= T/occ \\ (T\|T')/\rightarrow \cdot occ &= T'/occ \end{aligned}$$

(otherwise undefined), and we define similarly $\mathbf{T}/occ$. The (unique) occurrence of the hole in a parallel context, that is *occ* satisfying $\mathbf{T}/occ = []$, is denoted $@\mathbf{T}$. Whenever *occ* is an occurrence in T, that is T/occ is defined, we denote by $T[occ \leftarrow T']$ the thread system obtained from T by replacing the subtree T/occ at occurrence *occ* by T' (we omit the formal definition, by induction on *occ*, which should be obvious). With the notion of an occurrence, we are able to say where, that is, in which thread, a reduction occurs: an occurrence is similar to a thread identifier in a thread system.

The global operational semantics, again given by means of transition rules, is presented in Figure 2.3. The labels that decorate the transitions indicate the action being produced (a) and where (or which thread) in the thread system the step is taken ($@\mathbf{T}$). We will use this information in the following proofs. One can observe that at each step a single thread from the thread system is

$$\frac{e \xrightarrow{a} e'}{(S, L, \mathbf{T}[e]) \xrightarrow[@\mathbf{T}]{a} (S', L', \mathbf{T}[e'])}\ (*) \qquad \frac{e \xrightarrow{\mathsf{spw}_{e''}} e'}{(S, L, \mathbf{T}[e]) \xrightarrow[@\mathbf{T}]{\mathsf{spw}_{e''}} (S, L, \mathbf{T}[(e' \| e'')])}$$

Figure 2.3: Multithreaded semantics: Strong (*Interleavings*)

nondeterministically chosen and performs a step atomically. The resulting store S' and lock pool L' after a step depend on the action a being taken by the chosen thread as follows – where we denote by $S[p \leftarrow v]$ the store resulting from updating the reference p with the value v:

$$(*) = \begin{cases} a \in \{\beta, \swarrow, \searrow\} & \Rightarrow \quad S' = S \ \&\ L' = L \\ a = \nu_{p,v} & \Rightarrow \quad p \notin \mathsf{dom}(S) \ \&\ S' = S \cup \{p \mapsto v\} \ \&\ L' = L \\ a = \mathsf{rd}_{p,v} & \Rightarrow \quad S(p) = v \ \&\ S' = S \ \&\ L' = L \\ a = \mathsf{wr}_{p,v} & \Rightarrow \quad S' = S[p \leftarrow v] \ \&\ L' = L \\ a = \overset{\frown}{\ell} & \Rightarrow \quad \ell \notin L \ \&\ S' = S \ \&\ L' = L \cup \{\ell\} \\ a = \overset{\frown}{\ell} & \Rightarrow \quad S' = S \ \&\ L' = L \backslash \{\ell\} \end{cases}$$

Hence, the global semantics binds the actions of the different threads in a coherent way, with their effects being reflected in the global configuration. For instance, the read transition requires that the contents of the reference being read coincide in the store with the returned value, and a similar argument applies for acquiring a lock. Moreover, write actions modify the store, thus making the side effects of each thread immediately visible to all the other threads in the system. We shall not develop more on this semantics, as it is the standard interleaving semantics for this language.

We will consider the relation "$\rightarrow$" between configurations such that $C \rightarrow C'$ if there exists a and o such that $C \xrightarrow[o]{a} C'$; and we will denote by $C \xrightarrow{*} C'$ the reflexive and transitive closure of the "$\rightarrow$" relation. We say that C' is reachable from C if $C \xrightarrow{*} C'$. Our main result will be established for configurations that are reachable from an initial configuration of the form $(\emptyset, \emptyset, e)$ where e is a closed expression of the source language. More generally, we shall consider *regular* configurations, where at most one thread can hold a lock, and where a lock held by some thread is indeed in the lock context. This is defined as follows:

Definition 2.7 (Regular Configuration)**.** *A configuration $C = (S, L, T)$ is* regular *if and only if it satisfies*

i) if $T = \mathbf{T}[\mathbf{E}[(e \backslash \ell)]] \ \Rightarrow \ \ell \in L$, and
ii) if $T = \mathbf{T}_0[\mathbf{E}_0[(e_0 \backslash \ell)]] = \mathbf{T}_1[\mathbf{E}_1[(e_1 \backslash \ell)]]$ then $\mathbf{T}_0 = \mathbf{T}_1$, $\mathbf{E}_0 = \mathbf{E}_1$ and $e_0 = e_1$.

For instance, any configuration of the form $(\emptyset, \emptyset, e)$ where e is an expression of the source language is regular. The following should be obvious:

Remark 2.8. *If C is regular and $C \rightarrow C'$ then C' is regular.*

For the sake of simplicity, the only available values of our language are boolean values; however we prefer to present examples (or *litmus tests*) in their traditional form: that is using integer values. We will mostly use the values 0 and 1 which are trivially convertible to boolean values. Notice that the inclusion of integers in our language would be trivial as well. Let us reconsider a possible interleaving of the Example 1.2 presented in the introduction of this chapter, where we assume the initial store contains p and q with the default value of 0, and we leave β steps implicit:

$$\begin{array}{lcl}
\Big(\{p\mapsto 0, q\mapsto 0\},\emptyset, & p:=1\,;\ r_0:=(!q) & \parallel \quad q:=1\,;\ r_1:=(!p)\Big) \xrightarrow[\nwarrow]{\mathsf{wr}_{p,1}} \\
\Big(\{p\mapsto 1, q\mapsto 0\},\emptyset, & r_0:=(!q) & \parallel \quad q:=1\,;\ r_1:=(!p)\Big) \xrightarrow[\nearrow]{\mathsf{wr}_{q,1}} \\
\Big(\{p\mapsto 1, q\mapsto 1\},\emptyset, & r_0:=(!q) & \parallel \quad r_1:=(!p)\Big) \xrightarrow[\nwarrow]{\mathsf{rd}_{q,1}} \\
\Big(\{p\mapsto 1, q\mapsto 1\},\emptyset, & r_0:=1 & \parallel \quad r_1:=(!p)\Big) \xrightarrow[\nearrow]{\mathsf{rd}_{p,1}} \\
\Big(\{p\mapsto 1, q\mapsto 1\},\emptyset, & r_0:=1 & \parallel \quad r_1:=1\Big)
\end{array}$$

Notice that this is just one of the possible executions. It should be clear that no matter which thread starts first, the execution must start with a write to the store, which disallows the result $r_0 = r_1 = 0$ discussed in the introduction. We will reconsider this example when presenting the relaxed semantics in the next section.

To state the core result of this chapter, namely the *fundamental property* of relaxed memory models, we need first to define precisely what a *data-race* is, and then, what it means for a program to be free of data races.

Definition 2.9 (Data-Race)**.** *We will say that a configuration $C = (S, L, T)$ contains a data-race whenever $T = \mathbf{T}[\mathbf{E}[(p := v)]] = \mathbf{T}'[\mathbf{E}'[r]]$ with $\mathbf{T} \neq \mathbf{T}'$ and $r \in \{(!p), (p := w) | w \in \mathcal{V}al\}$.*

In other words, C has a data-race if there are two threads that can immediately perform actions on a certain reference p, such that at least one of the actions is a write, and the other is either a read or a write. This is a standard definition in concurrency theory. In particular, we do not consider concurrent accesses on the same lock to be a data-race. An alternative definition of data-race is presented in [Lamport, 1978], which required a definition of when an event *happens before* another event in the execution and two events are considered to form a data-race if they are not ordered by the happens-before relation. This definition is most commonly found in works regarding relaxed memory models, however programs containing data races according to this later definition can be proved to contain data races as in our more primitive definition [Boyland, 2008; Boehm and Adve, 2008]. We will discuss the happens before relation in the next section.

The definition of data-race can be easily lifted to programs.

Definition 2.10 (DRF Program)**.** *We say a configuration C is data-race free (DRF for short) if every configuration C' reachable from C by the interleaving semantics (i.e. $C \xrightarrow{*} C'$) contains no data-race. An expression e is data-race free if the configuration $(\emptyset, \emptyset, e)$ is data-race free.*

In the semantics of this chapter the lock construct is completely abstract. We do not discuss here the implementation of such a construct. However, it must

be acknowledged that such mechanism has to be implemented in a lower-level language to provide a compiler for the language of this section. In general, lock constructs will also be implemented using shared memory, such as the one we have in the store S of our configurations. Interestingly, most implementations of this construct will involve some kind of data-race in the language used to implement the source language considered here.

2.3 Concurrency, Conflict and Event Ordering

Inspired by the literature on *true concurrency semantics* [Boudol and Castellani, 1988, 1994] and Berry and Lévy's notion of *equivalence by permutation* of transitions in a computation [Berry and Lévy, 1979], we will formally define when an event "inherently precedes" another event. In some sense, this definition relates to the *happens-before* definition of Lamport [Lamport, 1978], but we also include the conflict notion into this definition. Some concepts that we will need in the sequel are those of *concurrency*, for which we will use the previous defined occurrences, and *conflict*, that identifies events that compete in accessing the memory in a possibly destructive way; meaning that they should not happen simultaneously to avoid hazardous situations. Also, in general, conflicting events cannot be sequentially reordered, since their reordering can lead to different results even for sequential programs. We notice here that these developments exhibit a great resemblance with those of [Boudol and Castellani, 1988, 1994].

Intuitively, we say two occurrences o_0 and o_1 are concurrent if they point to disjoint subtrees of the thread system in the configuration. This can be easily observed by comparing the occurrences:

Definition 2.11 (Concurrency)**.** *Two occurrences o and o' are concurrent, in notation $(o \smile o')$, if neither is a prefix of the other: $(o \not\leq o')$ & $(o' \not\leq o)$.*

We can easily observe that replacing subtrees in disjoint branches of a thread system renders the same result regardless of the order in which the replacements are performed.

Remark 2.12. *Given a thread system T, and two occurrences $o_0, o_1 \in \mathcal{O}cc$ such that $o_0 \smile o_1$ and both T/o_0 and T/o_1 are defined, then for all T_0 and T_1 we have:*

$$(T[o_0 \leftarrow T_0])[o_1 \leftarrow T_1] = (T[o_1 \leftarrow T_1])[o_0 \leftarrow T_0]$$

Notice that the previous remark applies identically to thread contexts rather than thread systems. Another property that is also obvious is that parts of the thread subsystem that are disjoint to the one being updated remain unmodified after the replacement.

Remark 2.13. *Given a thread system T, and two occurrences $o_0, o_1 \in \mathcal{O}cc$ such that $o_0 \smile o_1$ and both T/o_0 and T/o_1 are defined, for all T' we conclude that*

$$T/o_1 = (T[o_0 \leftarrow T'])/o_1$$

The *conflict* relation, which is standard in concurrency literature, is a binary relation on actions, whose intended meaning is to relate actions that have effects

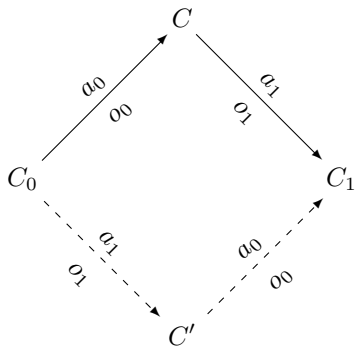


Figure 2.4: Diagram of the Asynchrony Lemma (2.15)

that possibly modify the result of the other related action, or restrict its use. Thus, a write on reference p and a subsequent read on p are conflicting actions, since the side effect of writing p generally modifies the result of reading p. To be more precise, conflict is defined as follows:

Definition 2.14 (Conflict)**.** *We define the conflict relation, denoted by #, to be the following binary relation on actions:*

$$\begin{aligned} \# \triangleq & \bigcup_{p\in\mathcal{R}ef,\ v,w\in\mathcal{V}al} \{(\mathsf{wr}_{p,v},\mathsf{wr}_{p,w}),(\mathsf{wr}_{p,v},\mathsf{rd}_{p,w}),(\mathsf{rd}_{p,v},\mathsf{wr}_{p,w})\} \\ & \cup \bigcup_{\ell\in\mathcal{L}ocks} \{\overset{\frown}{\ell},\overset{\frown}{\ell}\}\times\{\overset{\frown}{\ell},\overset{\frown}{\ell}\} \end{aligned}$$

The following *asynchrony* lemma – for details on the terminology refer to [Boudol and Castellani, 1988, 1994] – forms the core of the definition of the permutation equivalence of transitions. The lemma states that whenever we have two successive transitions that are nonconflicting and concurrent, in the sense of the above definitions, then the order in which these transitions are performed is irrelevant as regards the final configuration. A diagram of the statement of the lemma can be found in Figure 2.4. Alternatively, we could have proved a *diamond* property, as depicted in Figure 2.5, as it is the case in [Boudol and Castellani, 1988, 1994], stating that if there are two concurrent and nonconflicting enabled transitions in a certain configuration C_0 rendering two different configurations C and C' respectively, then a configuration C_1 can be found to which C and C' converge after performing the transition not performed in C. This will be the formalization of choice for the next chapter.

Lemma 2.15 (Asynchrony)**.** *Assuming a well-formed configuration C_0 and given a computation $C_0 \xrightarrow[o_0]{a_0} C \xrightarrow[o_1]{a_1} C_1$ such that $\neg(a_0 \# a_1)$ and $o_0 \smile o_1$, then there exists a unique (up to α-conversion) configuration C' with $C_0 \xrightarrow[o_1]{a_1} C' \xrightarrow[o_0]{a_0} C_1$.*

Proof. Let $C_i = (S_i, L_i, T_i)$, $C = (S, L, T)$ and $C' = (S', L', T')$. From the hypotheses we have

$$C_0 = (S_0, L_0, \mathbf{T}_0[\mathbf{E}_0[r_0]]) \xrightarrow[@\mathbf{T}_0]{a_0} (S, L, \mathbf{T}_0[T'_0]) = C$$

which means that $T = \mathbf{T}_0[T'_0]$, and

$$C = (S, L, \mathbf{T}_1[\mathbf{E}_1[r_1]]) \xrightarrow[@\mathbf{T}_1]{a_1} (S_1, L_1, \mathbf{T}_1[T'_1]) = C_1$$

where $T'_i = e_i$ if $a_i \neq \mathsf{spw}_{e'_i}$ or $T'_i = (e_i \| e'_i)$ otherwise. We know that $T/o_0 = T'_0$ and $T/o_1 = \mathbf{E}_1[r_1]$ with $o_0 \smile o_1$. That means, from remark 2.13, that $T_0/o_1 = \mathbf{E}_1[r_1]$ and therefore there is $\mathbf{T}'_1$ (precisely $\mathbf{T}'_1 = T[o_1 \leftarrow []]$) such that $T_0 = \mathbf{T}'_1[\mathbf{E}_1[r_1]]$, meaning that a_1 can be produced as the first step, and by remark 2.13 a_0 can be produced as the second one. It remains to see that the reordered steps produce the same resulting thread systems T'_i (a single expression in case both are not spw_{e_i} actions) and the final stores and lock pools are the same. To that end, we proceed by case analysis on the actions a_0 and a_1.

- $\{a_0, a_1\} \cap \{\beta, \swarrow, \searrow, \mathsf{spw}_{e'}\} \neq \emptyset$. These cases are trivial since at least one of the actions does not modify and/or depend in any way on the store S nor the lock pool L. Clearly the resulting thread systems are the same and the lock pool and store only change as defined by the action that uses the store or the lock pool, if any.
- $a_0 = \mathsf{wr}_{p,v}$. Let us be precise in this case, the other cases will follow the same reasoning. We have from hypothesis $\neg(a_0 \# a_1)$ that $a_1 \notin \{\mathsf{wr}_{p,w}, \mathsf{rd}_{p,w}\}$. Let $a_1 = \mathsf{wr}_{q,w}$ with $q \neq p$, then clearly $T'_1 = T'_0 = \langle\rangle$, and this is irrelevant of the order in which redexes are reduced, or the store S and lock pool L, and we have that in one case $S_1 = (S[p \leftarrow v])[q \leftarrow w]$ and on the other case $S_1 = (S[q \leftarrow w])[p \leftarrow v]$, which are clearly the same store. If $a_1 = \mathsf{rd}_{q,w}$ then we have $S'(q) = w = S(q)$ and thus $T'_1 = w$ in both cases, and again the final store only depends on a_0. The cases of $a_1 \in \{\overset{\frown}{\ell}, \overset{\frown}{\ell}, \nu_{q,w}\}$ use the same argument as before.
- $a_0 = \nu_{p,v}$. Then $a_1 \notin \{\mathsf{wr}_{p,w}, \mathsf{rd}_{p,w}\}$, since the reduction of $(\mathsf{ref}\ v)$ cannot affect the contents of a different thread (implied by the $o_0 \smile o_1$ hypothesis) and the configuration C_0 is well-formed. The cases where $a_1 \in \{\mathsf{rd}_{q,w}, \mathsf{wr}_{q,w}\}$ are similar to the previous case. Let us consider the case where $a_1 = \nu_{q,w}$: clearly $p \neq q$ because $p \in \mathsf{dom}(S)$ with $S = S_0 \cup \{p \mapsto v\}$ and the semantic rule requires that $q \notin \mathsf{dom}(S)$. Again, performing the actions in the reverse order renders $S_1 = S \cup \{p \mapsto v\} \cup \{q \mapsto w\}$, the order of the unions being irrelevant. The other cases are easy and do not deserve being developed.
- $a_0 = \overset{\frown}{\ell}$. We have that $a_1 \notin \{\overset{\frown}{\ell}, \overset{\frown}{\ell}\}$ and $S = S_0$. Then, the action a_1 can be performed in C_0 obtaining C where we know that $\ell \notin C$. Also, notice that the effect of a_1 on the store S will be directly propagated to the store S_1, and the same happens for the lock pool L_1. The case for $a_0 = \overset{\frown}{\ell}$ is similar to this one.

□

We can now define the equivalence by permutation of transitions on computations. The intention of the equivalence by permutations is to equalize computations that are indistinguishable up to the interleaving of actions of different threads. In other words, two computations will be considered equivalent if their only difference resides in the order in which actions of different threads are performed; however, the results of reads and writes must be the same. In this

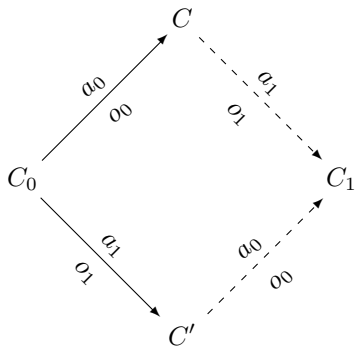


Figure 2.5: Diagram of the Diamond Property

way, we can consider the classes of executions that are indistinguishable up to irrelevant interleavings. A *computation* is a sequence of (decorated) transitions

$$\gamma = C_0 \xrightarrow[o_1]{a_1} C_1 \ \cdots \ C_{n-1} \xrightarrow[o_n]{a_n} C_n$$

More formally, γ is a sequence of steps $C_i \xrightarrow[o_{i+1}]{a_{i+1}} C'_i$ such that $C_{j+1} = C'_j$ for any j, but one should notice that, given an initial configuration C_0, the sequence of actions and occurrences is enough to determine the whole computation. Then the equivalence by permutation is the congruence (with respect to concatenation, which, on computations, is only partially defined) on such sequences generated by the asynchrony property:

Definition 2.16 (Permutation Equivalence)**.** *We define the equivalence by permutation of transitions to be the least equivalence $\simeq$ on computations satisfying the following conditions:*

i) if $C \xrightarrow[o_0]{a_0} C_0 \xrightarrow[o_1]{a_1} C'$ *with* $\neg(a_0 \# a_1)$ *and* $o_0 \smile o_1$*, then*

$$C \xrightarrow[o_0]{a_0} C_0 \xrightarrow[o_1]{a_1} C' \simeq C \xrightarrow[o_1]{a_1} C_1 \xrightarrow[o_0]{a_0} C'$$

where C_1 is determined as in the Asynchrony Lemma 2.15;

ii) $\gamma_0 \simeq \gamma'_0$ & $\gamma_1 \simeq \gamma'_1 \Rightarrow \gamma_0 \cdot \gamma_1 \simeq \gamma'_0 \cdot \gamma'_1$.

Some clear consequences of this definition are that if $\gamma \simeq \gamma'$ then $|\gamma| = |\gamma'|$, and γ and γ' perform the same actions at the same occurrences (and in the same number for each of such pair), possibly in a different order. The main purpose of this definition is to allow us to formally define an event ordering relation with respect to a computation γ. To introduce this last notion, let us first see an example. Let $e_0 = (p := v)$, $e'_0 = (p := v')$, $e_1 = (\lambda x(\lambda z z e'_0)\langle\rangle)$ and $T = (e_0 \| e_1)$. Then for S satisfying $p \in \mathsf{dom}(S)$ we have

$$\gamma = (S, \emptyset, T) \xrightarrow[\curvearrowleft]{\mathsf{wr}_{p,v}} (S', \emptyset, (\langle\rangle \| e_1)) \xrightarrow[\curvearrowright]{\beta} (S', \emptyset, (\langle\rangle \| (\lambda z z e'_0))$$
$$\xrightarrow[\curvearrowright]{\mathsf{wr}_{p,v'}} (S'', \emptyset, (\langle\rangle \| (\lambda z z \langle\rangle))) \xrightarrow[\curvearrowright]{\beta} (S'', \emptyset, (\langle\rangle \| \langle\rangle))$$

This computation is equivalent to the following one:

$$\gamma' = (S, \emptyset, T) \xrightarrow[\curvearrowright]{\beta} (S, \emptyset, (e_0 \| (\lambda z z e_0'))) \xrightarrow[\curvearrowleft]{\mathsf{wr}_{p,v}} (S', \emptyset, (() \| (\lambda z z e_0')) \xrightarrow[\curvearrowright]{\mathsf{wr}_{p,v'}} (S'', \emptyset, (() \| (\lambda z z ())) \xrightarrow[\curvearrowright]{\beta} (S'', \emptyset, (() \| ()))$$

where we have permuted the first two steps. We cannot go any further, since the second step in this computation (γ') is conflicting with the third, which in turn is not concurrent with the last one (these last two steps are in "program order" since they are performed from the same thread). In this example we can say that the first write ($\mathsf{wr}_{p,v}$) "inherently precedes" (this will be formally defined below) the second one, and also precedes the second β reduction. Moreover the first β reduction "inherently precedes" the second write ($\mathsf{wr}_{p,v'}$) and the second β reduction. This example shows that a given action, say $\mathsf{wr}_{p,v}$ in the example above, may have in a given computation several different relations with another action, for example β in γ. Then the notion of action, even if complemented with its occurrence, is not the right basis to define the ordering we are looking for. We have to introduce the notion of an *event*, as follows:

Definition 2.17 (Events)**.** *An* event *in a computation γ is a pair $(a, o)^i$ decorated by a positive integer i, where the action a is performed at occurrence o in the computation γ, and i is less than the number of such pairs (a, o) in γ (that is, $(a, o)^i$ is the i-th occurrence of action a performed at occurrence o in γ). We denote by $\mathsf{Events}(\gamma)$ the set of events determined by the computation γ.*

For instance, for the computation γ above, we have

$$\mathsf{Events}(\gamma) = \{(\mathsf{wr}_{p,v}, \curvearrowleft)^1, (\mathsf{wr}_{p,v'}, \curvearrowright)^1, (\beta, \curvearrowright)^1, (\beta, \curvearrowright)^2\}$$

For a given computation

$$\gamma = C_0 \xrightarrow[o_1]{a_1} C_1 \ \cdots \ C_{n-1} \xrightarrow[o_n]{a_n} C_n$$

we denote by $\partial(\gamma)$ the sequence $(a_1, o_1)^1 \cdots (a_n, o_n)^k$ of events of γ in temporal order, that is, as they appear successively in γ. We can finally define the *event ordering* determined by a computation:

Definition 2.18 (Event Ordering)**.** *Given a computation γ, we say that an event $(a, o)^i \in \mathsf{Events}(\gamma)$* inherently precedes *$(a', o')^j \in \mathsf{Events}(\gamma)$ in γ, in notation $(a, o)^i \leq_\gamma (a', o')^j$, if and only if in every $\gamma' \simeq \gamma$, the i-th occurrence of (a, o) precedes the j-th occurrence of (a', o').*

It should be clear that this is indeed an ordering, that is, a reflexive, transitive and anti-symmetric relation. Moreover, two conflicting actions can never be permuted, and therefore if

$$\partial(\gamma) = \sigma_0 \cdot (a, o)^i \cdot \sigma_1 \cdot (a', o')^j \cdot \sigma_2$$

with $a \# a'$ then $(a, o)^i \leq_\gamma (a', o')^j$. For instance, in the computation γ above, we have

$$(\mathsf{wr}_{p,v}, \curvearrowleft)^1 \leq_\gamma (\mathsf{wr}_{p,v'}, \curvearrowright)^1 \leq_\gamma (\beta, \curvearrowright)^2$$

and

$$(\beta, \Gamma')^1 \leq_\gamma (\mathsf{wr}_{p,v'}, \Gamma')^1$$

The event ordering in a computation contains in particular the "program order," that relates actions (a, o) and (a', o') in temporal order, such that $o \leq o'$ (this ordering also takes into account the "creation of redexes" identified by Lévy [Lévy, 1980] in the λ-calculus).

Remark 2.19. *Let γ be a computation such that $\partial(\gamma) = \sigma_0 \cdot (a, o)^i \cdot \sigma_1 \cdot (a', o')^j \cdot \sigma_2$ and $o \leq o'$ then $(a, o)^i \leq_\gamma (a', o')^j$.*

To conclude this section we prove a property of DRF programs that will be crucial in establishing our main result. This property shows in particular that if, in a computation starting from a DRF configuration, two conflicting actions are performed in different threads, then in between the two there must be synchronization, that is an action $\overset{\curvearrowleft}{\ell}$ of releasing a lock. Let us define:

Definition 2.20 (Well-Synchronized)**.** *A configuration C is* well-synchronized *if, for any computation*

$$C = C_0 \xrightarrow[o_1]{a_1} C_1 \ \cdots \ C_{n-1} \xrightarrow[o_n]{a_n} C_n$$

where $a_i \# a_j$ (with $i < j$) and $o_i \smile o_j$ then there exists h such that $i \leqslant h \leqslant j$, $a_h = \overset{\curvearrowleft}{\ell}$ and $o_i \leq o_h$.

This is similar to the DRF0 property of [Adve and Hill, 1990]. To show that DRF programs are well-synchronized, we first need a lemma stating that, two events can be moved to occur in the reverse temporal order in a computation γ, if the first of them (in the serialization order of γ) does not inherently precede the second one:

Lemma 2.21 (Transposition)**.** *Let γ be a computation such that*

$$\partial(\gamma) = \sigma_0 \cdot (a, o)^i \cdot \sigma_1 \cdot (a', o')^j \cdot \sigma_2$$

with $(a, o)^i \not\leq_\gamma (a', o')^j$. Then there exists γ' with $\gamma' \simeq \gamma$ and such that $\partial(\gamma') = \sigma_0 \cdot \sigma_1' \cdot (a', o')^j \cdot (a, o)^i \cdot \sigma_1'' \cdot \sigma_2$.

Proof. The proof is given by induction on $|\sigma_1|$. If $\sigma_1 = \varepsilon$, we have neither $o \leq o'$ nor $a \# a'$, since otherwise we would have $(a, o)^i \leq_\gamma (a', o')^j$, and therefore $o \smile o'$ and $\neg(a \# a')$. Then by definition of the permutation equivalence we can commute these two steps. If $\sigma_1 = (a'', o'')^h \cdot \xi$ there are two cases:

- $(a'', o'')^h \not\leq_\gamma (a', o')^j$. In this case we apply twice the induction hypothesis, to transpose first $(a'', o'')^h$ and $(a', o')^j$, and then the latter with $(a, o)^i$.
- $(a'', o'')^h \leq_\gamma (a', o')^j$. We do not have $o \leq o''$, since otherwise we would have $(a, o)^i \leq_\gamma (a', o')^j$ by Remark 2.19 and the transitivity of $\leq_\gamma$, and therefore $o \smile o''$. Similarly, it is impossible that $a \# a''$, and then by definition of the permutation equivalence we can transpose $(a, o)^i$ with $(a'', o'')^h$, and conclude using the induction hypothesis.

□

We can now establish a connection between data race free configurations and well-synchronized ones.

Proposition 2.22. *DRF regular configurations are well-synchronized.*

Proof. We will show that for any computation γ starting from a DRF regular configuration C, if

$$\partial(\gamma) = \sigma_0 \cdot (a, o)^i \cdot \sigma_1 \cdot (a', o')^j \cdot \sigma_2$$

with $(a, o)^i \leq_\gamma (a', o')^j$ and $o \smile o'$, then there exist ℓ, o'', k, ξ_0 and ξ_1 such that $(a, o)^i \cdot \sigma_1 \cdot (a', o')^j = \xi_0 \cdot (\overset{\frown}{\ell}, o'')^k \cdot \xi_1$ with $o \leq o''$, i.e. C is well-synchronized. We proceed by induction on $|\sigma_1|$.

- $\sigma_1 = \varepsilon$. In this case we must have $a \# a'$, since otherwise we could commute $(a, o)^i$ and $(a', o')^j$, contradicting $(a, o)^i \leq_\gamma (a', o')^j$. Then we proceed by cases on $a \# a'$. If $a, a' \in \{\mathsf{wr}_{p,v}, \mathsf{rd}_{p,v}\}$ for some p and v, where either a or a' is $\mathsf{wr}_{p,v}$, then e is not data-race free, since $o \smile o'$. Finally the only possible case is $a, a' \in \{\overset{\frown}{\ell}, \overset{\frown}{\ell}\}$ for some ℓ. Since one cannot acquire twice the same lock consecutively in a computation, either a or a' is $\overset{\frown}{\ell}$. Moreover, since C is a regular configuration, the same holds for the configuration performing $(a, o)^i$, by Remark 2.8, and therefore either $a = \overset{\frown}{\ell}$, or $a = \overset{\frown}{\ell}$, $a' = \overset{\frown}{\ell}$ and $o' = o$ which contradicts $o \smile o'$.
- $\sigma_1 = (a'', o'')^h \cdot \xi$. We distinguish again two cases.
  1. $(a'', o'')^h \leq_\gamma (a', o')^j$. If $o'' \smile o'$ then we use the induction hypothesis to conclude. Otherwise, we have $o'' \leq o'$, and since $o \smile o'$ this implies $o'' \smile o$, because if $o'' < o$ then we would also have $o \leq o''$ which in turn implies $o \leq o'$, contradicting $o \smile o'$. Now if $a \# a''$ we argue as in the base case, and otherwise we can commute $(a, o)^i$ with $(a'', o'')^h$, and then apply the induction hypothesis.
  2. $(a'', o'')^h \not\leq_\gamma (a', o')^j$. In this case we use the Transposition Lemma above, and conclude using the induction hypothesis.

□

2.4 The Weak Semantics

The main contribution of this chapter is the characterization of a *relaxed memory model* for the language of the previous section by means of a small-step operational semantics, and the proof of the fundamental property of memory models for this semantics. As mentioned in the introduction of the chapter, the key additional ingredient of this formalization is *write-buffers*. Even if the formalization of buffers presented here is inspired by the work of Dubois et al. [1998] it is not bound to a particular machine architecture. Instead here we try to reflect any hardware or software mechanism whose effect is to delay the commitment of writes into the main store. Some instances of such mechanisms are the reordering of memory accessing commands (at any level, ranging from compiler optimizations to pipelining at the hardware level), noncoherent caching and write-buffering just to mention some.

Let us now introduce some notations to talk about buffers. The precise meaning of buffers is captured by the following syntax:

$$B \quad ::= \quad \epsilon \mid B \triangleleft [p \mapsto v]$$

where $[p \mapsto v]$ is a pair consisting of $p \in \mathcal{R}ef$; and v, a run-time value of the language. This syntax corresponds to queues where ϵ is the empty list constructor and $\triangleleft$ is the cons, in reverse order, i.e. it adds the element at the end of the list. We prefer to have a concrete representation of buffers rather than having an abstract one, because, as we will see in Chapter 4, different interpretations of buffers (as a function, or as a sequence) render different semantics, that correspond to different memory models. In this chapter we will regard a buffer as determining a partial function from a set of references to sequences of values. That is, we denote by $B(p)$ the (FIFO) ordered list of pending writes on reference p in the buffer B as defined below, where we use the notations for sequences that we introduced in 2.6:

$$B(p) \triangleq \begin{cases} \varepsilon & \text{if } B = \epsilon \\ B'(p) & \text{if } B = B' \triangleleft [q \mapsto v] \ \&\ p \neq q \\ B'(p) \cdot v & \text{if } B = B' \triangleleft [p \mapsto v] \end{cases}$$

We will use the notion of buffers as a sequence in Chapter 4.

By abuse of notation we will denote by $B \triangleleft B'$ the concatenation of two buffers defined as follows:

$$B \triangleleft B' \triangleq \begin{cases} B & \text{if } B' = \epsilon \\ (B \triangleleft B'') \triangleleft [p \mapsto v] & \text{if } B' = B'' \triangleleft [p \mapsto v] \end{cases}$$

In this chapter, two buffers determining the same function are equivalent:

Definition 2.23 (Buffer equivalence)**.** *The buffers equivalence relation is the least equivalence $\equiv$ between buffers satisfying:*

$$\frac{p \neq q}{B_0 \triangleleft [p \mapsto v] \triangleleft [q \mapsto w] \triangleleft B_1 \equiv B_0 \triangleleft [q \mapsto w] \triangleleft [p \mapsto v] \triangleleft B_1}$$

This definition implies that for all buffers B and B' with $B \equiv B'$ we have that for all reference p the sequence of values for p in B is the same as the one in B', i.e. $B(p) = B'(p)$. For the rest of this chapter we will consider buffers up to the $\equiv$ equivalence.

We can now incorporate the notion of buffers into the thread system. The definition of thread systems containing buffers is as follows:

$$\Theta \ ::= \ T \ \mid \ \langle B \rangle \Theta \ \mid \ (\Theta \| \Theta') \qquad \textit{weak thread systems}$$

An example of one such system was presented in the Figure 2.1 in the introduction of the chapter. Now configurations are triples of the form:

$$C = (S, L, \Theta) \qquad \textit{weak configurations}$$

These configurations are called *weak* as opposed to the *strong* configurations that we presented earlier. Again, we will only consider *well-formed* weak configurations, which in particular adds the constraint that if in the configuration

$$\frac{e \xrightarrow{a} e' \qquad a \neq \mathsf{spw}_{e''}, \mathsf{wr}_{p,v}}{(S, L, \mathbf{\Theta}[e]) \underset{@\mathbf{\Theta}}{\overset{a}{\hookrightarrow}} (S', L', \mathbf{\Theta}[e'])}\ (*) \qquad \frac{e \xrightarrow{\mathsf{spw}_{e''}} e'}{(S, L, \mathbf{\Theta}[e]) \underset{@\mathbf{\Theta}}{\overset{\mathsf{spw}_{e''}}{\hookrightarrow}} (S, L, \mathbf{\Theta}[(e' \| e'')])}$$

$$\frac{e \xrightarrow{\mathsf{wr}_{p,v}} e'}{(S, L, \mathbf{\Theta}[e]) \underset{@\mathbf{\Theta}}{\overset{\mathsf{wr}_{p,v}}{\hookrightarrow}} (S, L, \mathbf{\Theta}[\langle \epsilon \triangleleft [p \mapsto v] \rangle e'])}$$

Figure 2.6: Multithreaded Semantics: Relaxed

(S, L, Θ) the reference p appears in the domain of a buffer contained in the thread system Θ, then $p \in \mathsf{dom}(S)$. Thread system contexts also need to be modified to encompass the inclusion of write-buffers, they become:

$$\mathbf{\Theta} \ ::= \ [] \ \mid \ \langle B \rangle \mathbf{\Theta} \ \mid \ (\mathbf{\Theta} \| \Theta) \ \mid \ (\Theta \| \mathbf{\Theta})$$

As we mentioned earlier, the semantics of reads will return the value of the latest pending write on any of the buffers visible to the thread; and if there is no such pending write, it will return the current value in the store. To give a precise statement for that intuition we adopt the notation $(S, \mathbf{\Theta})(p)$ to denote the latest value of p in the buffered thread context $\mathbf{\Theta}$ and store S. However, before defining it we need an auxiliary function that retrieves the ordered list of values from the buffers in the path from the hole in the context to the root, denoted here by $\mathbf{\Theta}(p)$:

$$\mathbf{\Theta}(p) \ \triangleq \ \begin{cases} \varepsilon & \text{if } \mathbf{\Theta} = [] \\ B(p) \cdot \mathbf{\Theta}'(p) & \text{if } \mathbf{\Theta} = \langle B \rangle \mathbf{\Theta}' \\ \mathbf{\Theta}'(p) & \text{if } \mathbf{\Theta} = (\mathbf{\Theta}' \| \Theta) \text{ or } \mathbf{\Theta} = (\Theta \| \mathbf{\Theta}') \end{cases}$$

Now obtaining the latest visible value for reference p on a certain weak thread context $\mathbf{\Theta}$ and store S is defined as follows:

$$(S, \mathbf{\Theta})(p) \ \triangleq \ \mathsf{last}(S(p) \cdot \mathbf{\Theta}(p))$$

We will denote by $\mathbf{\Theta}\dagger$ the fact that there are no pending writes on any reference in the buffers from occurrence of the hole in the context to the root of the thread system, to be precise:

$$\mathbf{\Theta}^{\dagger} \ \Leftrightarrow_{\text{def}} \ \forall p.\ \mathbf{\Theta}(p) = \varepsilon$$

Finally, before defining the semantics we need to extend the notion of occurrence, defined in the previous section, to buffered thread systems. For that purpose we will include a new symbol $\downarrow$, denoting the fact that the the occurrence has a buffer at this position. Then, the definition of subtree at occurrence *occ* is redefined in the obvious way, that is:

$$\langle B \rangle \mathbf{\Theta} / (\downarrow \ \cdot \ occ) \ = \ \mathbf{\Theta} / occ$$

where the remaining clauses remain the same.

We can now define the *weak semantics* ($C \overset{a}{\underset{o}{\hookrightarrow}} C'$) of our language in Figures 2.6 and 2.7 on the previous page, where we have:

$$(*) = \begin{cases} a \in \{\beta, \swarrow, \searrow\} & \Rightarrow \quad S' = S \;\&\; L' = L \\ a = \nu_{p,v} & \Rightarrow \quad p \notin \mathsf{dom}(S) \;\&\; S' = S \cup \{p \mapsto v\} \;\&\; L' = L \\ a = \mathsf{rd}_{p,v} & \Rightarrow \quad (S, \boldsymbol{\Theta})(p) = v \;\&\; S' = S \;\&\; L' = L \\ a = \overset{\frown}{\ell} & \Rightarrow \quad \ell \notin L \;\&\; S' = S \;\&\; L' = L \cup \{\ell\} \\ a = \overset{\frown}{\ell} & \Rightarrow \quad \boldsymbol{\Theta}^{\dagger} \;\&\; S' = S \;\&\; L' = L/\{\ell\} \end{cases}$$

Some of the transition rules change w.r.t. the interleaving semantics presented in Figure 2.3. Let us develop the most salient changes:

1. the write action ($\mathsf{wr}_{p,v}$) does not modify directly the store S but rather creates a new buffer that is immediately visible only to the thread performing the write. Thus, the write is delayed for the other threads. We give in Figure 2.7 the rules that allow such pending writes to propagate through the thread system hierarchy and eventually modify the store (which we will also call commit the write).

2. the read operation ($\mathsf{rd}_{p,v}$) inspects the contents of the buffers that affect the thread performing the action, and retrieves the latest pending write, if there is one. In case there is no such write the value is retrieved from the store. The function $(S, \boldsymbol{\Theta})(p)$ takes care of retrieving the latest value for p that affects the issuing thread.

3. the unlock operation ($\overset{\frown}{\ell}$) requires that buffers that directly affect the thread performing it to be empty. This can be interpreted as a *memory barrier*, or a *flush* instruction that proceeds until all previously delayed effects (in this case pending writes) have been globally committed, and affect all other threads. This is the intended meaning of the notation $\boldsymbol{\Theta}\dagger$. Notice however, that actions to acquire and release locks are atomic, in the sense that they cannot be interrupted. Also, these instructions are possibly blocking: locking requires that the lock be free and unlocking requires that the buffers be empty. Nevertheless, unlocking cannot be indefinitely blocked, since (as we will soon see) flushing the buffers cannot be blocked by other actions, and one can always choose to update the pending buffers to release the corresponding lock.

Now we present the rules related to updating the buffers. To do so we need to introduce some notations. We will represent by $\mathsf{W}(B)$ the set of references included in the buffer B, i.e. :

$$\mathsf{W}(B) \triangleq \begin{cases} \emptyset & \text{if } B = \epsilon \\ \{p\} \cup \mathsf{W}(B') & \text{if } B = B' \triangleleft [p \mapsto v] \end{cases}$$

To remove pending updates from the buffer when committing them into the memory, we will adopt the notation $B{\downarrow}p$ which stands for the buffer resulting from removing the oldest, i.e. first-in, pending write on the reference p in buffer

$$\frac{B(p) = v \cdot s}{(S, L, \langle B \rangle \Theta) \underset{\varepsilon}{\hookrightarrow} (S[p \leftarrow v], L, \langle B{\downarrow}p \rangle \Theta)} \qquad \frac{}{(S, L, \mathbf{\Theta}[\langle \epsilon \rangle \Theta]) \underset{@\mathbf{\Theta}}{\hookrightarrow} (S, L, \mathbf{\Theta}[\Theta])}$$

$$\frac{B_1(p) = v \cdot s}{(S, L, \mathbf{\Theta}[\langle B_0 \rangle \langle B_1 \rangle \Theta]) \underset{@\mathbf{\Theta}}{\hookrightarrow} (S, L, \mathbf{\Theta}[\langle B_0 \triangleleft [p \mapsto v] \rangle \langle B_1{\downarrow}p \rangle \Theta])}$$

$$\frac{B(p) = v \cdot s}{(S, L, \mathbf{\Theta}[(\langle B \rangle \Theta_0 \| \Theta_1)]) \underset{@\mathbf{\Theta}}{\hookrightarrow} (S, L, \mathbf{\Theta}[\langle \epsilon \triangleleft [p \mapsto v] \rangle (\langle B{\downarrow}p \rangle \Theta_0 \| \Theta_1)])}$$

$$\frac{B(p) = v \cdot s}{(S, L, \mathbf{\Theta}[(\Theta_0 \| \langle B \rangle \Theta_1)]) \underset{@\mathbf{\Theta}}{\hookrightarrow} (S, L, \mathbf{\Theta}[\langle \epsilon \triangleleft [p \mapsto v] \rangle (\Theta_0 \| \langle B{\downarrow}p \rangle \Theta_1)])}$$

Figure 2.7: Relaxed Semantics: Buffer Update Rules

B :

$$B{\downarrow}p \triangleq \begin{cases} \varepsilon & \text{if } B = \epsilon \\ B' & \text{if } B = B' \triangleleft [p \mapsto w] \ \&\ p \notin \mathsf{W}(B') \\ (B'{\downarrow}p) \triangleleft [q \mapsto w] & \text{if } B = B' \triangleleft [q \mapsto w] \ \&\ (p \neq q \text{ or } p \in \mathsf{W}(B')) \end{cases}$$

The rules for updating pending writes into the memory are presented in Figure 2.7. It is important to note here that these rules are nondeterministically triggered whenever they are enabled. Thus, buffers get lazily updated by means of these rules. Note also that the update rules do not include an action label, for that reason we can call them generically *silent steps*. Let us comment on each of these rules:

1. the first rule, describes the behavior of updating the memory by committing a pending write in the upper-most buffer (w.r.t. the thread system);
2. the second rule simply allows to eliminate buffers from the system when they are empty;
3. the third rule propagates pending writes in a buffer to its parent buffer;
4. and finally, the two last rules propagate pending writes upwards on a parallel composition node, the rules consider the left and right subtrees of the parallel node.

It is important to notice that the buffer update rules, propagate writes in a manner that respects the data dependencies of threads. In other words, write accesses issued by a single thread to a certain reference reach the memory (and for that matter, the intermediate buffers) in the same order as they were issued by the program. However, writes to different references can reach the store in different order (cf. jockeying in [Dubois et al., 1998]). By these means we can now explain the behavior presented in Example 1.3 in the introduction where we have omitted the steps that remove empty buffers as well as β reductions, and we have simplified the example by removing the local variables r_0 and r_1:

$$
\begin{array}{lcll}
\Big(\{p \mapsto 0, q \mapsto 0\}, & p := 1\,;\ q := 1 & \| & (!q)\,;\,(!p)\Big) \xhookrightarrow[\curvearrowleft]{\mathsf{wr}_{p,1}} \\
\Big(\{p \mapsto 0, q \mapsto 0\}, & \langle \epsilon \lhd [p \mapsto 1]\rangle (q := 1) & \| & (!q)\,;\,(!p)\Big) \xhookrightarrow[\curvearrowleft \cdot \downarrow]{\mathsf{wr}_{q,1}} \\
\Big(\{p \mapsto 0, q \mapsto 0\}, & \langle \epsilon \lhd [p \mapsto 1]\rangle (\langle \epsilon \lhd [q \mapsto 1]\rangle \langle\rangle) & \| & (!q)\,;\,(!p)\Big) \hookrightarrow_{\curvearrowleft} \\
\Big(\{p \mapsto 0, q \mapsto 0\}, & \langle \epsilon \lhd [p \mapsto 1] \lhd [q \mapsto 1]\rangle \langle\rangle & \| & (!q)\,;\,(!p)\Big) \hookrightarrow_{\curvearrowleft} \\
\Big(\{p \mapsto 0, q \mapsto 0\}, & \langle \epsilon \lhd [q \mapsto 1]\rangle (\langle \epsilon \lhd [p \mapsto 1]\rangle \langle\rangle) & \| & (!q)\,;\,(!p)\Big) \overset{*}{\hookrightarrow}_{\varepsilon} \\
\Big(\{p \mapsto 0, q \mapsto 1\}, & \langle \epsilon \lhd [p \mapsto 1]\rangle \langle\rangle & \| & (!q)\,;\,(!p)\Big) \xhookrightarrow[\curvearrowright]{\mathsf{rd}_{q,1}} \\
\Big(\{p \mapsto 0, q \mapsto 1\}, & \langle \epsilon \lhd [p \mapsto 1]\rangle \langle\rangle & \| & 1\,;\,(!p)\Big) \xhookrightarrow[\curvearrowright]{\mathsf{rd}_{p,0}} \\
\Big(\{p \mapsto 0, q \mapsto 1\}, & \langle \epsilon \lhd [p \mapsto 1]\rangle \langle\rangle & \| & 0\Big) \overset{*}{\hookrightarrow} \\
\Big(\{p \mapsto 1, q \mapsto 1\}, & \langle\rangle & \| & 0\Big)
\end{array}
$$

We can see here that the update to q is propagated to the store before the one of p, thus the reading thread can obtain a value of 1 for the read of q and a value 0 for the read of p.

Let us consider another classical example of relaxed memory models; namely, the IRIW example of [Boehm and Adve, 2008]:

Example 2.24.

$$
\left(\ [p := 1] \,\|\, \left[\begin{array}{l} r_0 := (!p)\,; \\ r_1 := (!q) \end{array} \right] \ \right) \| \left(\ [q := 1] \,\|\, \left[\begin{array}{l} r_2 := (!q)\,; \\ r_3 := (!p) \end{array} \right] \ \right)
$$

It is easy to check that with this configuration we could obtain as a result $r_0 = r_2 = 1$ and $r_1 = r_3 = 0$, which is impossible if we analyze sequential consistent behaviors of the program. However, notice that the threads systems we consider in this work are somewhat *rigid*, in the sense that if the threads where placed differently in the thread system, the set of behaviors are different too. For instance, one might naively consider that the previous thread system is equivalent to:

$$
\left(\ [p := 1] \,\|\, [q := 1] \ \right) \| \left(\ \left[\begin{array}{l} r_0 := (!p)\,; \\ r_1 := (!q) \end{array} \right] \,\|\, \left[\begin{array}{l} r_2 := (!q)\,; \\ r_3 := (!p) \end{array} \right] \ \right)
$$

but in this last one the behavior in question is no longer feasible. These two thread systems are actually reachable from different programs; the first one is reachable from the expression:

$$
\begin{array}{rcl}
e & = & (\mathsf{thread}\,\big((\mathsf{thread}\,p := 1)\,;\ r_0 := (!p)\,;\ r_1 := (!q)\big))\,; \\
 & & (\mathsf{thread}\,q := 1)\,; \\
 & & r_2 := (!p)\,; \\
 & & r_3 := (!q)
\end{array}
$$

whereas the second one is only reachable from the expression:

$$\begin{array}{rcl} e' & = & (\mathsf{thread}\,\big((\mathsf{thread}\,p := 1)\,;q := 1\big))\,; \\ & & (\mathsf{thread}\,\big(r_0 := (!p)\,;r_1 := (!q)\big))\,; \\ & & r_2 := (!q)\,; \\ & & r_3 := (!p) \end{array}$$

It is only in that sense that that our formalization allows the IRIW example. Therefore we cannot say that IRIW is allowed in general. Actually this is due to the semantics of thread creation. To preserve the dependencies of the sequential program, when a thread is spawned it must necessarily be aware of all previous writes of the parent thread. This means that, in some sense, thread creation implies some synchronization between the parent and the child threads. A possibility we considered, but we did not pursue, for modeling thread creation is to flush the buffers (actually require that they be empty) at the time of creating a new thread. It should be straightforward to see that this semantics would be more restrictive than the one we are considering here. On the other hand to have the two threads, the parent and the child, sharing buffers created up to the thread creation, but not necessarily the updates that will be subsequently buffered, requires a hierarchical structure as the one we have considered. Indeed, this explains why our thread systems are not associative. In Chapter 4 we will consider a formalization, by means of the semantics this chapter, of the memory models of Sparc [SPARC, 1994], and the constraint regarding the lack of associativity and commutativity will be removed. It is left as an excercise to the reader to verify that if we remove dynamic thread creation, and have a single (nonshared) buffer per processor, we obtain a semantics that does not allow the IRIW example. We will consider such semantics in Chapter 4 when modelling the PSO and TSO memory models. This observation is exploited in [Owens et al., 2009; Sewell et al., 2010] to model the semantics of x86 .

It should be intuitively clear that by means of the buffer update rules one can reduce weak configurations to standard ones by first updating every buffer, and then removing the empty ones. This is what we now prove. To this end let us denote by $(S, L, \Theta) \twoheadrightarrow (S', L', \Theta')$ the transition relation defined as follows:

$$C \twoheadrightarrow C' \iff \exists o \in \mathcal{O}cc,\ C \underset{o}{\hookrightarrow} C'$$

and let $\Theta /\!\!/ occ$ denote the buffer present at the node whose index is occ in Θ, if such buffer exists:

$$\begin{array}{rcl} \langle B\rangle\Theta /\!\!/ \varepsilon & = & B \\ \langle B\rangle\Theta /\!\!/ \downarrow \cdot occ & = & \Theta /\!\!/ occ \\ (\Theta \| \Theta') /\!\!/ \,\lrcorner \cdot occ & = & \Theta /\!\!/ occ \\ (\Theta \| \Theta') /\!\!/ \,\ulcorner \cdot occ & = & \Theta' /\!\!/ occ \end{array}$$

To prove that buffer updates eventually converge to a buffer free (i.e. strong) configuration it is useful to define the *size* $|\Theta|$ of the buffered thread systems Θ as follows:

$$\begin{array}{rcl} |T| & = & 0 \\ |\langle B\rangle\Theta| & = & 2 + |\Theta| + \|\Theta\| + |B| \\ |(\Theta_0 \| \Theta_1)| & = & 2 + \dfrac{(|\Theta_0| + |\Theta_1|)(|\Theta_0| + |\Theta_1| + 3)}{2} \end{array}$$

where $|B|$ denotes the number of pending updates in buffer B and similarly $\|\Theta\|$ for the thread system Θ. They are defined by:

$$\begin{array}{rcl} |B| & = & \sum_{p \in \mathsf{W}(B)} |B(p)| \\ \|\Theta\| & = & \sum_{\{\, o \mid \exists B.\ \Theta /\!\!/ o = B \,\}} |\Theta /\!\!/ o| \end{array}$$

A property that should be obvious is that the number of buffer update rules required to flush the buffers in a thread system is greater than the number of pending writes in it:

Lemma 2.25. *For all thread system with buffers Θ we have $\|\Theta\| \leq |\Theta|$*

Proof. The proof is trivial by induction on Θ. □

Another property that is useful at this point is that if the measure of a certain thread system Θ_0 is lower than that of another thread system Θ_1, then any thread system context composed with Θ_0 also has a lower measure than the same context composed Θ_1.

Lemma 2.26. *For all buffered thread context $\mathbf{\Theta}$ and thread systems Θ_0 and Θ_1 where $|\Theta_0| \leq |\Theta_1|$ we have $|\mathbf{\Theta}[\Theta_0]| \leq |\mathbf{\Theta}[\Theta_1]|$.*

Proof. Again the proof is trivial by induction on $\mathbf{\Theta}$. □

Then we can prove that performing a buffer update decreases the number of updates required to reach a strong configuration:

Lemma 2.27 (Termination)**.** $(S, L, \Theta) \twoheadrightarrow (S', L, \Theta') \;\Rightarrow\; |\Theta| > |\Theta'|$.

Proof. Provided with Lemma 2.26 we need to perform case analysis only in the thread sub-system where the buffer update rule takes place, i.e. in $\Theta /\!\!/ o$ if the rule considered is $(S, L, \Theta) \underset{o}{\hookrightarrow} (S', L, \Theta')$. Let us then proceed by cases:

- Consider the case where $\Theta = \mathbf{\Theta}[\langle\epsilon\rangle\Theta_0]$ and $\Theta' = \mathbf{\Theta}[\Theta_0]$. We need to see that $|\langle\epsilon\rangle\Theta_0| > |\Theta_0|$. Unfolding the definition of $|\langle\epsilon\rangle\Theta_0|$ we have that:

$$|\langle\epsilon\rangle\Theta_0| = 2 + |\epsilon| + |\Theta_0| + \|\Theta_0\| = 2 + |\Theta_0| + \|\Theta_0\| > |\Theta_0|$$

- Let us now consider the case where $\Theta = \mathbf{\Theta}[\langle B_0\rangle\langle B_1\rangle\Theta_0]$ and $\Theta' = \mathbf{\Theta}[\langle B_0 \lhd [p \mapsto v]\rangle\langle B_1 \downarrow p\rangle\Theta_0]$. Again, applying the definitions and the induction hypothesis we have:

$$\begin{array}{rl} |\langle B_0\rangle\langle B_1\rangle\Theta_0| & = 2 + |B_0| + \|\langle B_1\rangle\Theta_0\| + |\langle B_1\rangle\Theta_0| \\ & = 2 + |B_0| + |B_1| + \|\Theta_0\| + |\langle B_1\rangle\Theta_0| \\ & = 2 + |B_0 \lhd [p \mapsto v]| + |B_1{\downarrow}p| + \|\Theta_0\| + |\langle B_1\rangle\Theta_0| \\ & = 2 + |B_0 \lhd [p \mapsto v]| + \|\langle B_1{\downarrow}p\rangle\Theta_0\| + |\langle B_1\rangle\Theta_0| \\ & = 2 + |B_0 \lhd [p \mapsto v]| + \|\langle B_1{\downarrow}p\rangle\Theta_0\| + 2 + |B_1| + |\Theta_0| + \|\Theta_0\| \\ & = 2 + |B_0 \lhd [p \mapsto v]| + \|\langle B_1{\downarrow}p\rangle\Theta_0\| \\ & \qquad + 2 + |B_1{\downarrow}p| + |\Theta_0| + \|\Theta_0\| + 1 \\ & = 2 + |B_0 \lhd [p \mapsto v]| + \|\langle B_1{\downarrow}p\rangle\Theta_0\| + |\langle B_1{\downarrow}p\rangle\Theta_0| + 1 \\ & = |\langle B_0 \lhd [p \mapsto v]\rangle\langle B_1{\downarrow}p\rangle\Theta_0| + 1 \\ & > |\langle B_0 \lhd [p \mapsto v]\rangle\langle B_1{\downarrow}p\rangle\Theta_0| \end{array}$$

- Then we have the case of $\Theta = \mathbf{\Theta}[(\Theta_0 \| \langle B \rangle \Theta_1)]$ and $\Theta' = \mathbf{\Theta}[\langle \epsilon \triangleleft [p \mapsto v] \rangle (\Theta_0 \| \langle B \downarrow p \rangle \Theta_1)]$. Using Lemma 2.25 we have that:

$$\begin{aligned}
|(\Theta_0 \| \langle B \rangle \Theta_1)| &= 2 + \frac{(|\Theta_0| + |\langle B \rangle \Theta_1|)(|\Theta_0| + |\langle B \rangle \Theta_1| + 3)}{2} \\
&= 2 + \frac{(|\Theta_0| + |\langle B{\downarrow}p \rangle \Theta_1| + 1)(|\Theta_0| + |\langle B{\downarrow}p \rangle \Theta_1| + 3 + 1)}{2} \\
&= 2 + |(\Theta_0 \| \langle B{\downarrow}p \rangle \Theta_1)| + |\Theta_0| + |\langle B{\downarrow}p \rangle \Theta_1| \\
&= 2 + |(\Theta_0 \| \langle B{\downarrow}p \rangle \Theta_1)| + |\Theta_0| + 2 + |B{\downarrow}p| + |\Theta_1| + \|\Theta_1\| \\
&= 2 + |(\Theta_0 \| \langle B{\downarrow}p \rangle \Theta_1)| + |\Theta_0| + 1 + |B| + |\Theta_1| + \|\Theta_1\| \\
&\geq 2 + |(\Theta_0 \| \langle B{\downarrow}p \rangle \Theta_1)| + \|\Theta_0\| + 1 + |B| + \|\Theta_1\| + \|\Theta_1\| \\
&> 2 + |(\Theta_0 \| \langle B{\downarrow}p \rangle \Theta_1)| + \|\Theta_0\| + |B| + \|\Theta_1\| \\
&= 2 + |\epsilon \triangleleft [p \mapsto v]| + |(\Theta_0 \| \langle B{\downarrow}p \rangle \Theta_1)| + \|(\Theta_0 \| \langle B{\downarrow}p \rangle \Theta_1)\| \\
&= |\langle \epsilon \triangleleft [p \mapsto v] \rangle (\Theta_0 \| \langle B{\downarrow}p \rangle \Theta_1)|
\end{aligned}$$

  The case where $\Theta = \mathbf{\Theta}[(\langle B \rangle \Theta_0 \| \Theta_1)]$ is symmetric.
- Finally the case where $\Theta = \langle B \rangle \Theta_0$ and $\Theta' = \langle B{\downarrow}p \rangle \Theta_0$ is trivial.

□

One can also check that if $|\Theta| > 0$ then there is (S', L, Θ') and o such that $(S, L, \Theta) \underset{o}{\hookrightarrow} (S', L, \Theta')$, and therefore we have:

Corollary 2.28. *For any weak configuration (S, L, Θ) there exists a strong configuration (S', L, T) such that $(S, L, \Theta) \overset{*}{\twoheadrightarrow} (S', L, T)$.*

A consequence is that unlock operations are actually never deadlocked, in the sense that buffers affecting the operation can always be flushed to reach a configuration where the unlock operation can succeed.

We can now define a relation between weak and strong configurations stating that the strong configuration can be reached from the weak one performing only buffer update rules:

Definition 2.29 (Flattening)**.**

$$(S, L, \Theta) \Downarrow (S', L, T) \;\Leftrightarrow_{\text{def}}\; (S, L, \Theta) \overset{*}{\twoheadrightarrow} (S', L, T)$$

Notice that the flattening relation ($\Downarrow$) is not deterministic in general. There are configurations $C = (S, L, \Theta)$ for which several possible stores S' satisfy $C \Downarrow (S', L, T)$. In particular this can result from configurations in which two branches contain pending writes to the same reference with different values. In this case, the order in which buffer updates are applied determines the final configuration. We will call this kind of configurations noncoherent in the next section.

2.5 Proof of the DRF Guarantee

In this section we will prove that for data-race free programs, the relaxed semantics introduced in the previous section coincides with the reference semantics.

Notice that the meaning of "coincides" here is that the flat configurations (once all the buffers have been updated in the relaxed semantics) are identical. To prove this property we will state a relation between configurations of the weak and strong semantics, which we will prove then to be a bisimulation.

Let us first define a simple property on weak configurations stating that no two branches of the same buffered thread system contain pending writes on the same reference.

Definition 2.30 (Coherence)**.** *A buffered thread system* Θ *is* coherent *if and only if*

$$o \smile o' \;\Rightarrow\; \forall p \in \mathcal{R}ef.\ \mathsf{W}(\Theta /\!\!/ o) \cap \mathsf{W}(\Theta /\!\!/ o') = \emptyset$$

In other words, Θ is coherent if for any given reference p the set

$$\{\, o \mid p \in \mathsf{W}(\Theta /\!\!/ o) \,\}$$

is totally ordered by the prefix order $\leq$. This property obviously holds for any standard configuration (S, L, T), hence in particular for initial configurations of the form $(\emptyset, \emptyset, e)$. It should be intuitively clear that the relation $\Downarrow$ is deterministic for coherent configurations. This is what we now prove. First we observe that silent transitions preserve coherence:

Lemma 2.31. *If* Θ *is coherent and* $(S, L, \Theta) \twoheadrightarrow (S', L', \Theta')$ *then* Θ' *is coherent.*

Proof. We will check that for any reference q, if $q \in \mathsf{W}(\Theta' /\!\!/ o)$ and $q \in \mathsf{W}(\Theta' /\!\!/ o')$ then $o \not\asymp o'$, or equivalently $o \leq o'$ or $o' \leq o$. The proof proceeds by cases on the transition $(S, L, \Theta) \underset{occ}{\hookrightarrow} (S', L', \Theta')$:

- Suppose that $\Theta = \mathbf{\Theta}[(\langle B\rangle\Theta_0 \| \Theta_1)]$ and $\Theta' = \mathbf{\Theta}[\langle \epsilon \lhd [p \mapsto v]\rangle(\langle B \downarrow p\rangle\Theta_0 \| \Theta_1)]$. The case where $p \neq q$ is uninteresting, since the properties of q would not change by this transition. Let us assume then, that $p = q$. Since Θ is coherent, by the hypotheses, we have that $o \leq @\mathbf{\Theta} = occ$, or $o = @\mathbf{\Theta} \cdot \downarrow \cdot o'$ where $p \in \mathsf{W}(\langle B\downarrow p\rangle\Theta_0 /\!\!/ o'$, and therefore the occurrences of writes for p in Θ' are totally ordered w.r.t. the prefix ordering.
- then we have the case where $\Theta = \mathbf{\Theta}[(\Theta_0 \| \langle B\rangle\Theta_1)]$ and $\Theta' = \mathbf{\Theta}[\langle \epsilon \lhd [p \mapsto v]\rangle(\Theta_0 \| \langle B\downarrow p\rangle\Theta_1)]$ which is similar to the previous one.
- Notice that for the other cases, the prefix ordering of the propagated write only decreases, which means that occurrences that where related continue to be so. We will not develop these cases since they are immediate.

□

Now we show that silent transitions on coherent configurations are locally confluent up to buffer equivalence. To that end we need to lift the definitions of equivalent buffers to weak thread systems, and weak configurations. Two weak thread systems Θ and Θ' are equivalent up to buffer equivalence (which abusing the notation we denote by $\Theta \equiv \Theta'$) if for all occurrence o such that $\Theta/o = \langle B\rangle\Theta_0$ then we have that $\Theta'/o = \langle B'\rangle\Theta'_0$ where $B \equiv B'$ and $\Theta_0 \equiv \Theta'_0$. The extension to configuration simply requires the thread systems to be equivalent. Not surprisingly we will denote $C \equiv C'$ the fact that C and C' are equivalent up to the equivalence of buffers.

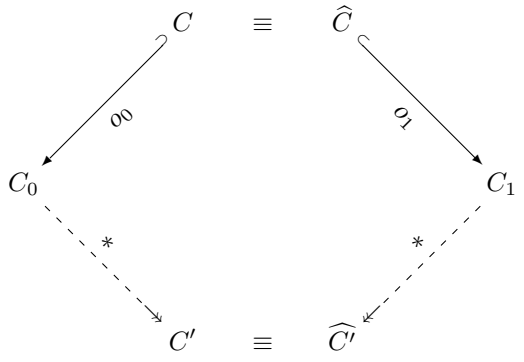


Figure 2.8: Diagram of Lemma 2.33

Notice that flat configurations contain no buffers, and therefore we have equality rather that the weaker equivalence of buffers.

Remark 2.32. *Given C and C' two strong configurations we have*

$$C \equiv C' \iff C = C'$$

We can now prove a confluence lemma similar to the asynchrony lemma 2.15 up to the buffer equivalence ($\equiv$). A diagram representing the lemma can be seen in Figure 2.8.

Lemma 2.33 (Local Confluence Modulo $\equiv$)**.** *If C and $\widehat{C}$ are equivalent coherent weak configurations (i.e. $C \equiv \widehat{C}$), and $C \underset{o_0}{\hookrightarrow} C_0$ and $\widehat{C} \underset{o_1}{\hookrightarrow} C_1$ then either $C_0 \equiv C_1$ or there are C' and $\widehat{C'}$ such that $C_0 \overset{*}{\twoheadrightarrow} C'$ and $C_1 \overset{*}{\twoheadrightarrow} \widehat{C'}$ with $C' \equiv \widehat{C'}$.*

Proof. There are many cases to consider, which are all easy, and therefore we only examine a few of them. For instance, we may have, if the transitions $C \underset{o_0}{\hookrightarrow} C_0$ and $C \underset{o_1}{\hookrightarrow} C_1$ are performed at disjoint occurrences, $C = (S, L, \mathbf{\Theta}[(\langle B_0\rangle\Theta_0 \| \langle B_1\rangle\Theta_1)]$ and

$$\begin{array}{rcl} C_0 & = & (S, L, \mathbf{\Theta}[\langle \epsilon \triangleleft [p \mapsto v]\rangle(\langle B{\downarrow}p\rangle\Theta_0 \| \langle B_1\rangle\Theta_1)]) \\ C_1 & = & (S, L, \mathbf{\Theta}[\langle \epsilon \triangleleft [q \mapsto v']\rangle(\langle B_0\rangle\Theta_0 \| \langle B_1{\downarrow}q\rangle\Theta_1)]) \end{array}$$

Since C is coherent, we have $p \neq q$, and in this case we let

$$C' = (S, L, \mathbf{\Theta}[\langle \epsilon \triangleleft [p \mapsto v] \triangleleft [q \mapsto v']\rangle(\langle B_0{\downarrow}p\rangle\Theta_0 \| \langle B_1{\downarrow}q\rangle\Theta_1)])$$

and we have $C_0 \overset{*}{\twoheadrightarrow} C'$ and $C_1 \overset{*}{\twoheadrightarrow} C'$ in three steps. Notice that here we consider buffers up to the $\equiv$ equivalence (or to put it differently, as functions).

When the two transitions $C \underset{o_0}{\hookrightarrow} C_0$ and $C \underset{o_1}{\hookrightarrow} C_1$ are performed at occurrences that are related by the prefix order, we may have for instance $C = (S, L, \mathbf{\Theta}[\langle B_0\rangle\langle B_1\rangle\langle B_2\rangle\Theta])$ with

$$\begin{array}{rcl} C_0 & = & (S, L, \mathbf{\Theta}[\langle B_0 \triangleleft [p \mapsto v]\rangle\langle B_1{\downarrow}p\rangle\langle B_2\rangle\Theta]) \\ C_1 & = & (S, L, \mathbf{\Theta}[\langle B_0\rangle\langle B_2 \triangleleft [q \mapsto v']\rangle\langle B_2{\downarrow}q\rangle\Theta]) \end{array}$$

(where possibly $p = q$). In this case we let

$$C' = (S, L, \mathbf{\Theta}[\langle B_0 \lhd [p \mapsto v]\rangle\langle (B_1{\downarrow}p) \lhd [q \mapsto v']\rangle\langle B_2{\downarrow}q\rangle\Theta])$$

and we have $C_0 \overset{*}{\twoheadrightarrow} C'$ and $C_1 \overset{*}{\twoheadrightarrow} C'$ in one step. □

It has been proved in [Huet, 1980] that as a consequence of the lemmas 2.27, 2.31, 2.33 and the remark 2.32, the relation $\twoheadrightarrow$ is confluent up to the buffer equivalence relation ($\equiv$) on coherent configurations, and therefore we have the following corollary.

Corollary 2.34. *If C is a coherent configuration and*

$$C \Downarrow C_0 \;\&\; C \Downarrow C_1 \;\Rightarrow\; C_0 = C_1$$

To establish our main result, that is the "fundamental property" for our weak memory model, we need a technical definition. We denote by $\pi(o)$ the *projection* of the (weak) occurrence o, given as follows:

$$\begin{array}{rcl} \pi(\varepsilon) & = & \varepsilon \\ \pi(\downarrow \cdot o) & = & \pi(o) \\ \pi(\mathcal{L} \cdot o) & = & \mathcal{L} \cdot \pi(o) \\ \pi(\mathcal{R} \cdot o) & = & \mathcal{R} \cdot \pi(o) \end{array}$$

Definition 2.35 (The Bisimulation Relation)**.** *For any given (strong) configuration C, we define the relation $\mathcal{R}(C)$ between weak and strong configurations as follows: $C'\,\mathcal{R}(C)\,C''$ if and only if there exists a sequence of weak transitions*

$$C_0 = C \overset{*}{\twoheadrightarrow} \underset{o_0}{\overset{a_0}{\hookrightarrow}} C_1 \cdots \overset{*}{\twoheadrightarrow} \underset{o_n}{\overset{a_n}{\hookrightarrow}} C_n = C'$$

such that

$$C'_0 = C \xrightarrow[\pi(o_0)]{a_0} C'_1 \cdots \xrightarrow[\pi(o_n)]{a_n} C'_n = C''$$

is a valid sequence of (strong) transitions, with $C_i \Downarrow C'_i$ for all i.

(Notice that since C is a standard configuration, we actually have, with the notations of the definition, $C \underset{o_0}{\overset{a_0}{\hookrightarrow}} C_1$.) We show that, if C is a DRF configuration, the relation $\mathcal{R}(C)$ is indeed a bisimulation. First, we observe that the weak semantics simulates the reference one:

Proposition 2.36. *If $C'\,\mathcal{R}(C)\,C''$ and $C'' \underset{o}{\overset{a}{\hookrightarrow}} \overline{C''}$ then there exist $\overline{C'}$ and o' such that $C' \overset{*}{\twoheadrightarrow} \underset{o'}{\overset{a}{\hookrightarrow}} \overline{C'}$ with $o = \pi(o')$ and $\overline{C'}\,\mathcal{R}(C)\,\overline{C''}$.*

Proof. This is immediate, because if

$$C_0 = C \overset{*}{\twoheadrightarrow} \underset{o_0}{\overset{a_0}{\hookrightarrow}} C_1 \cdots \overset{*}{\twoheadrightarrow} \underset{o_n}{\overset{a_n}{\hookrightarrow}} C_n = C'$$

is such that

$$C \xrightarrow[\pi(o_0)]{a_0} C'_1 \cdots \xrightarrow[\pi(o_n)]{a_n} C'_n = C''$$

with $C_i \Downarrow C'_i$ for all i, hence in particular $C' \Downarrow C''$, then we have $C' \overset{*}{\twoheadrightarrow} C'' \underset{o}{\overset{a}{\hookrightarrow}} C'''$, and in all cases except $a = \mathsf{wr}_{p,v}$ we have $C''' = \overline{C''}$, hence obviously $C''' \Downarrow \overline{C''}$. It is easy to see that $C''' \Downarrow \overline{C''}$ also holds in the case where $a = \mathsf{wr}_{p,v}$, since there is only one buffered write (on p) in C'''. □

To prove that conversely, the weak semantics does not deviate from the reference semantics as regards data-race free programs, we need to introduce the following notation that generalizes the definition of pending writes on buffers to thread systems in the obvious way:

$$\mathsf{W}(\Theta) \triangleq \begin{cases} \emptyset & \text{if } \Theta = T \\ \mathsf{W}(B) \cup \mathsf{W}(\Theta') & \text{if } \Theta = \langle B \rangle \Theta' \\ \mathsf{W}(\Theta') \cup \mathsf{W}(\Theta'') & \text{if } \Theta = (\Theta' \| \Theta'') \end{cases}$$

And we can now prove the following lemma.

Lemma 2.37. *Let C be a strong regular configuration such that*

$$C = C_0 \xrightarrow[o_0]{*} C_1 \cdots \xrightarrow[o_n]{*} C_n = (S, L, \Theta)$$

with $p \in \mathsf{W}(\Theta /\!\!/ o)$. Then there exists i such that $a_i = \mathsf{wr}_{p,v}$ with $\pi(o) \leq \pi(o_i)$, and for all $i < j \leqslant n$ if $\pi(o_i) \leq \pi(o_j)$ then $a_j \neq \overset{\curvearrowleft}{\ell}$.

Proof. The proof is by induction on n. First we observe that, due to the hypothesis $\mathsf{W}(\Theta) \neq \emptyset$, we must have $n \neq 0$, since C is a standard configuration. If $n = 1$, then it is easy to see that the only possibility, in order to have $\mathsf{W}(\Theta) \neq \emptyset$, is $a_1 = \mathsf{wr}_{p,v}$ with $\mathsf{W}(\Theta) = \{p\}$ (and $o = o_1$).

Otherwise ($n > 1$), we proceed by cases on a_n. We notice that if $a_n = \overset{\curvearrowleft}{\ell}$ then $o \not\leq o_n$. The lemma is obvious in the case where $a_n = \mathsf{wr}_{p,v}$ and $o = o_n$. Otherwise, we have $C_{n-1} \xrightarrow[o_n]{*} C_n$, and if $C_{n-1} = (S', L', \Theta')$ we have $p \in \mathsf{W}(\Theta')$ with

$$p \in \mathsf{W}(\Theta /\!\!/ o) \;\Rightarrow\; \exists o'.\ \pi(o') \leq \pi(o) \;\&\; p \in \mathsf{W}(\Theta' /\!\!/ o')$$

and we conclude using the induction hypothesis. □

Now we show that, for data-race free regular configurations, the second half of our bisimulation result holds. Moreover, we show that in the bisimulation scenario, the coherence property is preserved by the weak semantics (not just the silent transitions as in Lemma 2.31):

Proposition 2.38. *If C is a DRF regular configuration, $C' \,\mathcal{R}(C)\, C''$ where C' is coherent and $C' \xrightarrow[o]{*} \overline{C'}$ then $\overline{C'}$ is coherent and there exists $\overline{C''}$ such that $C'' \xrightarrow[\pi(o)]{a} \overline{C''}$ and $\overline{C'} \,\mathcal{R}(C)\, \overline{C''}$.*

Proof. We have

$$C_0 = C \xrightarrow[o_0]{*} C_1 \cdots \xrightarrow[o_n]{*} C_n = C'$$

and

$$C \xrightarrow[\pi(o_0)]{a_0} C'_1 \cdots \xrightarrow[\pi(o_n)]{a_n} C'_n = C''$$

with $C_i \Downarrow C'_i$ for all i. Let D be such that $C' \xrightarrow{*} D \xrightarrow[o]{a} \overline{C'}$. Then D is coherent by Lemma 2.31. By Corollary 2.28 there exists $\overline{D}$ such that $D \Downarrow \overline{D}$, hence $C' \Downarrow \overline{D}$,

and therefore $\overline{D} = C''$ by Corollary 2.34. We proceed by induction on the length of the sequence of $\twoheadrightarrow$ -transitions from D to C''. If this length is 0, that is, $D = C''$, we have $\pi(o) = o$ since C'' is a strong configuration, and either $a \neq \mathsf{wr}_{p,v}$ and $D \xrightarrow[o]{a} \overline{C'}$, or $a = \mathsf{wr}_{p,v}$. In the first case, we may let $\overline{C''} = \overline{C'}$. In the second case there obviously exists $\overline{C''}$ such that $D \xrightarrow[o]{a} \overline{C''}$ and $\overline{C'} \Downarrow \overline{C''}$. Since there is exactly one write buffered in $\overline{C'}$, this configuration is coherent, and clearly $\overline{C'}\,\mathcal{R}(C)\,\overline{C''}$.

Otherwise let D' be such that $D \underset{o'}{\hookrightarrow} D' \overset{*}{\twoheadrightarrow} C''$. We show that there exist $\overline{D}$ and u such that $\overline{D}$ is coherent and $D' \xrightarrow[u]{a} \overline{D}$ with $\pi(u) = \pi(o)$ (we shall then conclude using the induction hypothesis regarding D'). We proceed by cases on the transitions $D \xrightarrow[o]{a} \overline{C'}$ and $D \underset{o'}{\hookrightarrow} D'$. There are many cases to consider, most of which are immediate. We only examine the ones where $a = \mathsf{wr}_{p,v}$ or $\mathsf{rd}_{p,v}$, that is $D = (S, L, \Theta)$ with $\Theta = \mathbf{\Theta}[\mathbf{E}[r]]$ where $r = (p := v)$ or $r = (!\,p)$ and $o = @\mathbf{\Theta}$.

- $r = (p := v)$. We have $\overline{C'} = (S, L, \mathbf{\Theta}[\langle \epsilon \triangleleft [p \mapsto v]\rangle \mathbf{E}[()]])$. If $o \smile o'$, let us consider the case where $\mathbf{\Theta} = \mathbf{\Theta}'[\mathbf{\Theta}_0[\langle B_0\rangle\langle B_1\rangle\Theta']\|\mathbf{\Theta}_1]$ with

  $$D' = (S, L, \mathbf{\Theta}'[\mathbf{\Theta}_0[\langle B_0 \triangleleft [q \mapsto v']\rangle\langle B_1{\downarrow}q\rangle\Theta']\|\mathbf{\Theta}_1[\mathbf{E}[r]]])$$

  and $o = @\mathbf{\Theta}'\cdot \curvearrowright \cdot @\mathbf{\Theta}_1$. Assume that $q = p$. Then $p \in \mathsf{W}(\Theta /\!\!/ o'\cdot\downarrow)$ with

  $$o' = @\mathbf{\Theta}'\cdot \curvearrowleft \cdot @\mathbf{\Theta}_0$$

  and by Lemma 2.37 there exists i such that $a_i = \mathsf{wr}_{p,v}$ and $\pi(o'\cdot\downarrow) \leq \pi(o_i)$, with $a_j \neq \overset{\curvearrowleft}{\ell}$ for $i < j \leqslant n$ if $\pi(o_i) \leq \pi(o_j)$. Then $\pi(o_i) \smile \pi(o)$, but this contradicts Proposition 2.22 since $a_i \# a$ and C is data-race free and regular. Then it must be the case that $q \neq p$, and if we let $\overline{D} = (S, L, \Theta'')$ where

  $$\Theta'' = \mathbf{\Theta}'[\mathbf{\Theta}_0[\langle B_0 \triangleleft [q \mapsto v']\rangle\langle B_1{\downarrow}q\rangle\Theta']\|\mathbf{\Theta}_1[\langle \epsilon \triangleleft [p \mapsto v]\rangle \mathbf{E}[()]]]$$

  then we have $D' \xrightarrow[o]{a} \overline{D}$. It remains to see that $\overline{D}$ is coherent. Assume that $p \in \Theta''/\!/o''$ with $o'' \smile o$. Then by Lemma 2.37 there exists i such that $a_i = \mathsf{wr}_{p,v}$ and $\pi(o'') \leq \pi(o_i)$, with $a_j \neq \overset{\curvearrowright}{\ell}$ for $i < j \leqslant n$ if $\pi(o_i) \leq \pi(o_j)$, but, as above, this contradicts Proposition 2.22.

  Still assuming $o \smile o'$, let us consider the case where $\mathbf{\Theta} = \mathbf{\Theta}'[(\langle B\rangle\Theta'\|\mathbf{\Theta}_1)]$ with $o = @\mathbf{\Theta}'\cdot \curvearrowright \cdot @\mathbf{\Theta}_1$ and

  $$D' = (S, L, \mathbf{\Theta}'[\langle \epsilon \triangleleft [p \mapsto v']\rangle(\langle B{\downarrow}q\rangle\Theta'\|\mathbf{\Theta}_1[\mathbf{E}[r]])])$$

  and $o' = @\mathbf{\Theta}'\cdot \curvearrowleft$. Since $q \in \mathsf{W}(\Theta /\!\!/ o')$, we can show, using as in the previous case Lemma 2.37 and Proposition 2.22, that $q \neq p$ (since otherwise this would contradict the assumption that C is DRF). Then we let in this case $\overline{D} = (S, L, \Theta'')$ where

  $$\Theta'' = \mathbf{\Theta}'[\langle \epsilon \triangleleft [q \mapsto v']\rangle(\langle B{\downarrow}q\rangle\Theta'\|\mathbf{\Theta}_1[\langle \epsilon \triangleleft [p \mapsto v]\rangle \mathbf{E}[()]])]$$

We have $D \xrightarrow[u]{a} \overline{D}$ where $u = @\mathbf{\Theta}' \cdot \downarrow \cdot \Rsh \cdot @\mathbf{\Theta}_1$, and we conclude as in the previous case. All the other cases (with $o \smile o'$ or $o' \leq o$) are easy. As indicated above, we conclude the proof of the Proposition in the case where $r = (p := v)$ using the induction hypothesis regarding D'.

- $r = (!\, p)$. We have $\overline{C'} = (S, L, \mathbf{\Theta}[\mathbf{E}[v]])$ where $v = (S, \mathbf{\Theta})(p)$. We only examine the case where $\mathbf{\Theta} = \mathbf{\Theta}'[(\langle B\rangle \Theta' \| \mathbf{\Theta}_0)]$ with $o = @\mathbf{\Theta}' \cdot \Rsh \cdot @\mathbf{\Theta}_0$, $o' = @\mathbf{\Theta}' \cdot \Lsh$ and

$$D' = (S, L, \mathbf{\Theta}'[\langle \epsilon \lhd [q \mapsto v']\rangle(\langle B{\downarrow}q\rangle \Theta' \| \mathbf{\Theta}_0[\mathbf{E}[r]])]$$

(all the other cases are easy). Assume that $q = p$. Then $p \in \mathsf{W}(\Theta /\!\!/ o')$, and therefore by Lemma 2.37 there exists i such that $a_i = \mathsf{wr}_{p,w}$ with $\mathrm{D}\pi(o_i) \smile \pi(o)$ and $a_i \# a$. Then we must have $q \neq p$ in this case, and it is easy to see that we then have $(S, \mathbf{\Theta}'')(p) = v = (S, \mathbf{\Theta})(p)$ where

$$\mathbf{\Theta}'' = \mathbf{\Theta}'[\langle \epsilon \lhd [q \mapsto v']\rangle(\langle B{\downarrow}q\rangle \Theta' \| \mathbf{\Theta}_0)]$$

Therefore if we let $u = @\mathbf{\Theta}''$ and $\overline{D} = (S, L, \mathbf{\Theta}''[\mathbf{E}[v]])$ we have $D' \xhookrightarrow[u]{a} \overline{D}$ and $\pi(u) = \pi(o)$. By Lemma 2.31 D' is coherent, hence so is $\overline{D}$. We conclude the proof, as above, using the induction hypothesis for D'.

□

As an obvious consequence of the propositions 2.36 and 2.38 (and of Corollary 2.34), we finally obtain the correctness result:

Theorem 2.39 (Correctness)**.** *The weak memory model implements the reference semantics for data-race free programs. More precisely, the strong configurations reachable from a (strong) DRF regular configuration C in the weak semantics coincide with the configurations reachable from the same configuration C in the reference semantics.*

Notice that in particular the weak semantics correctly implements sequential programs, that do not use the $(\mathsf{thread}\, e)$ construct.

2.6 Conclusion

In this chapter we have considered an operational formalization of the semantics of write-buffering architectures. Since most relaxed memory architectures provide this kind of relaxation, we believe our formalization can be adapted to describe the memory models present in several existing commercial specifications. We will develop the case of the TSO and PSO memory models of Sparc [SPARC, 1994] in Chapter 4. Sewell et. al. consider in [Owens et al., 2009; Sewell et al., 2010] a formalization that is very similar to ours for the x86 architecture, where the hierarchy of the thread systems is flat (since there is no dynamic thread creation).

Since our formalization is sustained by standard programming languages techniques it is on one hand, simple to understand, and on the other hand adequate for developing language-based techniques. Interesting research directions are providing thread-safe programming models that are robust with respect

to write-buffering architectures, and also considering the problem of program transformations, like in the works [Ševčík, 2009; Saraswat et al., 2007] from the point of view of this semantics.

To summarize, from the point of view of relaxed memory models, the formalization of this chapter is sufficient to model a thread "reading its own writes early" as shown in example 2.3, and the effects of reordering a write followed by a read on a different location ($\mathbf{W} \to \mathbf{R}$), and of a write followed by another write on a different location ($\mathbf{W} \to \mathbf{R}$), as described in [Adve and Gharachorloo, 1996]. The attentive reader might have observed that we did not consider relaxations such as the reordering of a read followed by a write ($\mathbf{R} \to \mathbf{W}$) from example 1.4, or the reordering of a read followed by a read ($\mathbf{R} \to \mathbf{R}$) which can be observed in the following example, a slight variation of Example 1.3, where a barrier (denoted by $\langle \mathtt{wr}|\mathtt{wr} \rangle$) is added between the writes to avoid their reordering.

Example 2.40 (Read Read Reordering)**.**

$$\left[\begin{array}{l} p := 1\,; \\ \langle \mathtt{wr}|\mathtt{wr} \rangle\,; \\ q := 1 \end{array}\right] \quad \| \quad \left[\begin{array}{l} r_0 := (!q)\,; \\ r_1 := (!p) \end{array}\right]$$

In the program above, assuming that the initial stores satisfies that $p = q = 0$, we can only obtain $r_0 = 1$ and $r_1 = 0$ as a final result if the reads of the thread on the right are performed in a reverse order, or "delayed". This reordering, however, is not possible with the write buffers we consider here since read actions are performed immediately (as opposed to the buffered writes). Indeed, adding this kind of behaviors will be the subject of the following chapter, in which we consider an operational formalization of speculative computation, which among other effects, allows these reorderings.

Chapter 3

A Formalization of Speculative Computation

Speculative computation [Burton, 1985; Knight, 1986] is an implementation technique that leverages the high cost of accessing the memory by computing pieces of sequential code in advance, or in parallel. The formal semantics of *speculative computations* is the main topic of this chapter. Some of these speculative techniques are: pipelining [Hennessy and Patterson, 1996], instruction level parallelism [Fisher, 1981], out-of-order execution, branch prediction [Smith, 1981] and thread level speculation [Burton, 1985] just to mention a few. In this work we will define a general framework for speculative computation, rather than considering each of these particular mechanisms in detail. In some sense, our investigations are independent of any implementation technique. More precisely we will only focus on the effects that the reordering of memory actions can have in the execution of parallel programs disregarding other important effects like performance.

As with the optimizations discussed in the previous chapter, most speculative computation techniques preserve the data and control dependencies present in each of the sequential components of a parallel programs. This means that for purely sequential programs, speculative computations are indistinguishable from the normal – that is program order preserving – computations. For parallel programs however, reordering the execution of individual instructions of a thread can have visible effects for other threads, and thus, executions that do not correspond with the standard *interleaving semantics* of the program are to be expected.

We said before that the goal of this chapter is to provide a semantic framework to deal with speculative computations, but actually one of the main motivations to formalize speculative computations is that they also provide a framework to describe many of the behaviors present in relaxed memory models. We will see that by means of speculative computations we are able to express many of the examples of possible "noninterleaving" behaviors of parallel programs in the literature of relaxed memory models – the so called *litmus tests*. We will show that our framework of speculative computations enables to express a wide range of execution relaxations – in particular extending the relaxations allowed by the semantics with write buffers of the previous chapter. This claim will be

later justified by the formalization of some existing commercial specifications of memory models in the following chapter (4). In particular we will consider the family of memory models of the Sparc [SPARC, 1994] architecture, and we will see that the framework of the previous chapter is enough to characterize the TSO and PSO flavors of Sparc, but it is somewhat restrictive to capture the behaviors of RMO; whereas the framework of this chapter is also capable of modeling the RMO memory model. Notwithstanding, the developments of this chapter are independent of any memory model.

An interesting consequence of our formalization of speculative computation is that it allows us to characterize which speculations are "intuitively" desirable and which are not. In a nutshell, computing speculatively consists in making a certain *prediction*; for example the return value of a future read, or the result of a complex condition in a branching construct; computing according to this prediction and finally *validating* the prediction. Clearly, not all predictions will be correct and, possibly, a speculated execution should not be considered as a valid execution of the program. We will provide a formal definition of what we consider to be a *valid* speculation. The intuition here is that a speculation can be regarded as valid if there is an "equivalent" sequential execution for each of the parallel components of the program. We shall use a very precise definition of *equivalent* execution, which is a variant of the *equivalence by permutations* of Lévy and Berry [Berry and Lévy, 1979] that we have already used in the previous chapter. This intuition seems to be the common interpretation of what is considered to be a desirable valid speculations in the literature. It might be important to clarify here that we consider a certain type of speculative techniques and their validity conditions. We do not claim to provide an exhaustive definition of speculative computation, nor the ultimate definition of what should be considered valid or not. More modestly, ours in an attempt to formalize the intuitive notions commonly found in the literature on speculative computation and relaxed memory models.

3.1 The Language & Semantics

In this chapter we will consider a slight variation of the language of Chapter 2. Specifically, we will devote the first part of this chapter to results regarding a language with locks similar to the ones presented in the previous chapter, and we will devote a second part of the chapter to a language without locks, but with barriers as the only mechanism to restrict the reorderings allowed by the underlying speculative semantics. To simplify the developments, factoring many common results, we will introduce the language with both locks and barriers, and we will consider two sublanguages for the main results of this chapter.

Let us concentrate now on the technical aspects of the language. The source language that we discuss here varies from the one in Chapter 2 in that it is given in *Administrative Normal Form* [Flanagan et al., 1993] (ANF). This means that only values can be applied as functions to an argument. In fact, the ANF of our language is not exactly the same as in [Flanagan et al., 1993] since the language in that work contains a primitive let construct, whereas in our language it is just a syntactic sugar form. An important remark regarding the use of an ANF instead of the standard syntax is that it does not impose constraints on the expressive power of the language. Since there is a trivial translation of

programs in the language of Chapter 2 to the language we are considering here, the use of this ANF does not impose restrictions regarding the syntax either. Obviously this translation – which we will present shortly – preserves the exact semantics of the original program, and allows us to develop our technical results by considering this simpler and technically more convenient syntax.

Let us now present the concrete syntax of the language:

$$
\begin{array}{rcll}
e & ::= & v \mid (ve) \mid (\mathsf{if}\ v\ \mathsf{then}\ e_0\ \mathsf{else}\ e_1) & \textit{expressions} \\
 & \mid & (\mathsf{ref}\, v) \mid (\mathsf{!}\, v) \mid (v_0 := v_1) & \\
 & \mid & (\mathsf{thread}\, e) \mid (\mathsf{with}\ \ell\ \mathsf{do}\ e) \mid (\mathsf{cas}\ v) & \\
 & \mid & \langle \mathtt{rd}|\mathtt{rd}\rangle \mid \langle \mathtt{rd}|\mathtt{wr}\rangle \mid \langle \mathtt{wr}|\mathtt{rd}\rangle \mid \langle \mathtt{wr}|\mathtt{wr}\rangle & \textit{barriers} \\
v & ::= & x \mid \lambda x e \mid \mathit{tt} \mid \mathit{ff} \mid () & \textit{values}
\end{array}
$$

where we make the same assumptions of the previous chapter; that is: variables are sampled from the set $\mathcal{V}ar$ of variable names, constants tt and ff correspond to the boolean values, λxe stands for a function with parameter x bound in e; expressions are considered up to α-conversion, and the notation $\{x \mapsto e'\}e$ stands for the capture avoiding substitution of the free occurrences of variable x by the expression e' in the expression e. Again we adopt the standard syntactic forms $(\mathsf{let}\ x = e_0\ \mathsf{in}\ e_1)$ and $(e_0\,; e_1)$ to denote $(\lambda x e_1 e_0)$, where x is not free in e_1 for the latter one.

Most of the constructs of this language have already been introduced in Chapter 2. Indeed, the only difference is that here we have incorporated barriers and a compare-and-swap $(\mathsf{cas}\ v)$ instruction to the language. Barriers will be used later to restrict the reordering of actions according to their kind. Clearly the kind of actions we consider here are reads (denoted by $\mathtt{rd}$) and writes (denoted by $\mathtt{wr}$); thus a barrier with the shape $\langle \mu|\mu'\rangle$ prevents actions of kind μ' from being executed before previous actions (in the sense of the program text) of kind μ. For instance, the barrier $\langle \mathtt{wr}|\mathtt{rd}\rangle$ requires that all writes that come before the barrier in the program text be performed before any read that follows the barrier (again in the program order) be performed. Let us postpone the treatment of barriers for the moment. The $(\mathsf{cas}\ p)$ instruction is provided for atomicity purposes only. This expression checks a memory location p and if it contains the value ff it atomically sets it to tt while returning tt, whereas if it contains tt it does nothing, returning ff. We say that $(\mathsf{cas}\ p)$ is only provided for atomicity because, unlike the read-modify-write instructions of the x86 architecture [Intel Corporation, 2007; Owens et al., 2009], our compare-and-swap construct does not provide barrier semantics. This approach is in accordance with architectures like Sparc [SPARC, 1994] for example. For explanations on the other constructs of the language the reader is invited to review the discussion of the language of Chapter 2.

Two Sublanguages The main results of this chapter are robustness guarantees stating that for programs satisfying a certain property – one for each sublanguage – the speculative semantics and the interleaving semantics coincide. Not surprisingly, these properties involve the use of the synchronization mechanisms present in the sublanguages to which the property corresponds.

As we said before we will consider two sublanguages of the language we have just introduced. The motivation for having these two sublanguages is based

on the level of abstraction at which synchronization is considered. On the one hand, for a high-level programming language a synchronization mechanism such as locks (or monitors, etc.) is to be expected, and in their presence, barriers and compare-and-swap instructions are not a requirement[1]. On the other hand, a low-level programming language, as it could be the instruction set architecture (ISA) of a machine architecture, is unlikely to have sophisticated synchronization mechanisms such as locks. At this programming level it is more common to have only barriers and some atomic read/write instruction such as compare-and-swap. This difference is the main motivation for the two different languages we will consider here. However, most of the intermediate results (in particular the formalization framework) are common to both these sublanguages, and hence, we state them for the richer language including both locks and barriers. It should be clear that the synchronization discipline (or synchronization model according to [Adve and Hill, 1990]) varies significantly between these two sublanguages.

It is useful to have names for the sublanguages we consider in the chapter, so let us now present the two sublanguages which we call λ-lock for the high-level language with locks, as indicated by the name, and λ-barrier for the low-level language with barriers and the compare-and-swap construct.

Definition 3.1 (Two Sub-languages)**.** *The high-level sublanguage λ-lock is given by the following syntax:*

$$\begin{array}{rcll} e & ::= & v \mid (ve) \mid (\mathsf{if}\ v\ \mathsf{then}\ e_0\ \mathsf{else}\ e_1) & \lambda\text{-lock} \\ & \mid & (\mathsf{ref}\, v) \mid (\mathsf{!}\, v) \mid (v_0 := v_1) & \\ & \mid & (\mathsf{thread}\, e) \mid (\mathsf{with}\ \ell\ \mathsf{do}\ e) & \end{array}$$

And the low-level sublanguage λ-barrier by the following one:

$$\begin{array}{rcll} e & ::= & v \mid (ve) \mid (\mathsf{if}\ v\ \mathsf{then}\ e_0\ \mathsf{else}\ e_1) & \lambda\text{-barrier} \\ & \mid & (\mathsf{ref}\, v) \mid (\mathsf{!}\, v) \mid (v_0 := v_1) & \\ & \mid & (\mathsf{thread}\, e) \mid (\mathsf{cas}\ v) & \\ & \mid & \langle \mathtt{rd}|\mathtt{rd}\rangle \mid \langle \mathtt{rd}|\mathtt{wr}\rangle \mid \langle \mathtt{wr}|\mathtt{rd}\rangle \mid \langle \mathtt{wr}|\mathtt{wr}\rangle & \textit{barriers} \end{array}$$

As we said before, most of the definitions and intermediate results of the chapter are general (sometimes parametric over the synchronization mechanism) and not for a particular sublanguage. Whenever a particular sublanguage is to be considered we will acknowledge this fact carefully. Whenever we do not clarify the sublanguage we are considering it must be assumed that the most general language, with both barriers and locks, is targeted.

As we mentioned earlier, there is a direct translation, which we denote here by $\mathcal{T}$, from the language of Chapter 2 to the high-level sublanguage, λ-lock of this chapter.

We present the translation in Figure 3.1. This translation is very simple, and it should be easy to see that the semantics of expressions on the left is preserved by the transformation. For perspicuity in the rest of this chapter we will state most of our examples in their nonANF form, since it makes them clearer and more concise; thus the translation will be left implicit in general. We

[1] Actually from the point of view of the synchronization model (as in [Adve and Hill, 1990]) these are almost undesirable since they complicate the definition.

$$
\begin{array}{rcll}
\mathcal{T}(\lambda x e) & \Rightarrow & (\lambda x\ \mathcal{T}(e)) & \\
\mathcal{T}(v) & \Rightarrow & v & v \neq (\lambda x e) \\
\mathcal{T}(e_0 e_1) & \Rightarrow & (\mathsf{let}\ x = \mathcal{T}(e_0)\ \mathsf{in} & x \notin FV(e_1) \\
 & & \quad (x\ (\mathcal{T}(e_1)))) & \\
\mathcal{T}(\mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2) & \Rightarrow & (\mathsf{let}\ x = \mathcal{T}(e_0)\ \mathsf{in} & \\
 & & \quad (\mathsf{if}\ x\ \mathsf{then}\ \mathcal{T}(e_1)\ \mathsf{else}\ \mathcal{T}(e_2))) & \\
 & & & x \notin FV(e_1) \cup FV(e_2) \\
\mathcal{T}(\mathsf{ref}\ e) & \Rightarrow & (\mathsf{let}\ x = e\ \mathsf{in}\ (\mathsf{ref}\ x)) & \\
\mathcal{T}(\mathbf{!}\ e) & \Rightarrow & (\mathsf{let}\ x = e\ \mathsf{in}\ (\mathbf{!}\ x)) & \\
\mathcal{T}(e_0 := e_1) & \Rightarrow & (\mathsf{let}\ x = \mathcal{T}(e_0)\ \mathsf{in} & \\
 & & \quad (\mathsf{let}\ y = \mathcal{T}(e_1)\ \mathsf{in}\ x := y) & \\
 & & & x, y \notin FV(e_0) \cup FV(e_1) \\
\mathcal{T}(\mathsf{thread}\, e) & \Rightarrow & (\mathsf{thread}\, (\mathcal{T}(e))) & \\
\mathcal{T}(\mathsf{with}\ \ell\ \mathsf{do}\ e) & \Rightarrow & (\mathsf{with}\ \ell\ \mathsf{do}\ \mathcal{T}(e)) &
\end{array}
$$

Figure 3.1: Administrative Normal Form transformation $\mathcal{T}$

will formally justify the use of the translation by a simple lemma establishing the correspondence between the semantics of expressions before and after applying $\mathcal{T}$; but to do so, we need to introduce the semantics of the language first.

A few aspects of the semantics are different from the one presented previously in Figure 2.3 of Chapter 2. We will include in a box the elements that are novel to the formalization of this chapter (and also with respect to [Boudol and Petri, 2010]), these boxes are not part of the syntax of the language. The runtime language includes:

$$
\begin{array}{rcll}
e & ::= & \ldots \mid \boxed{\lambda v^{?} e_0\ e_1} \mid (e \backslash \ell) & \textit{expressions} \\
v & ::= & \ldots \mid p \mid \boxed{\lambda v^{?} e} & \textit{values}
\end{array}
$$

with the following evaluation contexts and redexes:

$$
\begin{array}{rcll}
\mathbf{E} & ::= & [] \mid (v\mathbf{E}) \mid (\mathbf{E}\backslash\ell) & \textit{evaluation contexts} \\
r & ::= & (\lambda x e v) \mid \boxed{(\lambda v^{?} e v)} \mid (\mathsf{if}\ v\ \mathsf{then}\ e_0\ \mathsf{else}\ e_1) & \textit{redexes} \\
 & \mid & (\mathsf{ref}\ v) \mid (\mathbf{!}\ p) \mid (p := v_1) & \\
 & \mid & (\mathsf{thread}\, e) \mid (\mathsf{with}\ \ell\ \mathsf{do}\ e) \mid (v\backslash\ell) \mid (\mathsf{cas}\ p) & \\
 & \mid & \langle \mathtt{rd} | \mathtt{rd} \rangle \mid \langle \mathtt{rd} | \mathtt{wr} \rangle \mid \langle \mathtt{wr} | \mathtt{rd} \rangle \mid \langle \mathtt{wr} | \mathtt{wr} \rangle &
\end{array}
$$

Notice that we have incorporated a new kind of λ-expressions in the runtime expressions, namely $(\lambda v^{?} e_0 e_1)$ which is marked inside a box, and accordingly a new kind of value $\lambda v^{?} e$ which corresponds to the functional element of the application. These expressions contain a tagged value $v^{?}$ in the place of the binder of the *lambda* expression; its intention is to signal that the function argument has been *predicted* (or *speculated*) to be v and this value has already

been substituted in the body of the λ-abstraction for the corresponding variable. The idea here is to decouple the substitution resulting from an application, from the actual application that happens once a value is produced for the argument. This decoupling will later allow us to avoid some artificial dependencies (due to redex creation) that are inherent to the λ-calculus that we use. We will shortly come back to this issue once we introduce the speculative semantics. The exact usage of this technique will be given with the semantics, but we anticipate that the usual β-reduction of the standard λ-calculus takes two steps here. For example, if we consider the expression (λxev) we have to first reduce it by predicting the argument:

$$(\lambda xev) \xrightarrow{\beta_v} (\lambda v^?\{x \mapsto v\}ev) = (\lambda v^? e' v)$$

and only after this step we can reduce it once more to obtain the result of a normal β reduction:

$$(\lambda v^? e' v) \xrightarrow{\beta} e'$$

We will refer to this second step as the *validation* or *commitment* of the first one.

Notice that since values of the form $\lambda v^? e$ do not appear in the source language they can only result after reducing an application of the form $(\lambda x e_0 e_1)$, which guarantees that the resulting value $\lambda v^? e$ is placed in a functional position of an application, and the only possible use for that value is to apply it. Therefore, a value of this kind can never appear as an argument, and cannot appear in the store either. Simply said, this new value acts as a function that is applied to an argument, and thus it cannot constitute a value to be manipulated by other constructs of the language, for which it deserves no special attention.

As usual, we denote by $\mathbf{E}[e]$ the expression resulting from filling the hole in $\mathbf{E}$ by e. We can establish a standard lemma, which justifies the use of evaluation contexts, stating that every runtime expression can be decomposed into a unique evaluation context and redex, or it is faulty. The notion of faulty expression changes slightly from that presented in Chapter 2 to account for $(\lambda v^? e w)$ expressions. The extended definition of faulty expressions becomes:

Definition 3.2 (Faulty expression)**.** *We say an expression e is faulty if it satisfies any of the conditions of Definition 2.4 of Chapter 2 or:*

- $e = (\lambda v^? e w)$ *and* $v \neq w$.

We now establish the standard property for the decomposition of runtime expressions into a unique evaluation contexts and redex:

Lemma 3.3. *For any expression e of the run-time language, either e is a value, or there are unique* $\mathbf{E}$*, an evaluation context, and* e'*, a redex or a faulty expression, such that* $e = \mathbf{E}[e']$.

As we did previously we introduce the semantics in two stages by considering first the semantics of single expressions, or threads, with a label indicating the corresponding action; and then, composing threads together with the store and the state of locks in a single configuration. Let us present the actions for

individual reductions:

$$
\begin{array}{rcl}
a \in \mathcal{A}ct & ::= & \beta \mid \boxed{\beta_v} \mid \swarrow \mid \searrow \mid \nu_{p,v} \mid \mathsf{wr}_{p,v} \mid \mathsf{rd}_{p,v} \mid \boxed{\mathsf{rd}^{\circ}_{p,v}} \\
& \mid & \boxed{\mathsf{cas}_{p,v}} \mid \boxed{\mathsf{cas}^{\circ}_{p,v}} \mid \overset{\frown}{\ell} \mid s \mid \boxed{b} \\
s \in \mathcal{S}ync & ::= & \mathsf{spw}_e \mid \overset{\curvearrowleft}{\ell} \\
b \in \mathcal{B}ar & ::= & \mathtt{rr} \mid \mathtt{rw} \mid \mathtt{wr} \mid \mathtt{ww}
\end{array}
$$

where v is a closed value, e is a closed expression and $p \in \mathcal{R}ef$ is a reference. The actions are almost the same as in the previous chapter, the new ones are marked inside a box and these are β_v, the barrier actions (`rr`, `wr`, `wr` and `ww`), the $\mathsf{cas}_{p,v}$ and $\mathsf{cas}^{\circ}_{p,v}$ actions for compare-and-swap, and $\mathsf{rd}^{\circ}_{p,v}$. We remark that the actions β_v, $\mathsf{rd}^{\circ}_{p,v}$ and $\mathsf{cas}^{\circ}_{p,v}$ and barriers where not considered in our previous work [Boudol and Petri, 2010]. The β_v action stands for the first reduction in the example above, by which the anticipated argument of a function application is substituted in the body of the λ-abstraction without consuming the argument. Later, in the speculative semantics, the value of the argument will be "predicted", justifying the fact that the β reduction needs to validate the prediction. Barrier actions do not deserve much introduction, since their only effect is to constraint the reordering of actions in the computation of the thread. The $\mathsf{cas}_{p,v}$ action attempts to set to *tt* the reference p in case this one contains a *ff* value and signals in the action whether it succeeded in doing so or not. A $\mathsf{cas}_{p,tt}$ action represents a successful compare-and-swap, whereas a $\mathsf{cas}_{p,ff}$ stands for a failure.

Perhaps the most intriguing new actions are $\mathsf{rd}^{\circ}_{p,v}$ and $\mathsf{cas}^{\circ}_{p,v}$ which we will use as a read, and respectively a compare-and-swap, with a different semantics to that of the standard read $\mathsf{rd}_{p,v}$ and compare-and-swap $\mathsf{cas}_{p,v}$. Indeed, the $\mathsf{rd}^{\circ}_{p,v}$ action is intended to model the behavior of a read that possibly sees a past or future value, as it is in the case of a thread that "reads its own writes early" in the previous chapter. The same reasoning applies to $\mathsf{cas}^{\circ}_{p,v}$, except that it is only the read that is speculated, the compare-and-swap succeeds or fails according to the predicted reading. Behaviors other than reading own writes early are possible, for instance reading outdated (or *old*) values as could result from adding a noncoherent cache structure in the architecture. But let us now move on with the semantics to better understand these actions.

We present the semantics of single expressions in Figure 3.2. These rules bear a great resemblance to those of Figure 2.2 of the previous chapter. Indeed, only a few rules are different: the rule for β-reduction which simply consumes the argument that has already been substituted by means of the new rule β_v; obviously the β_v rule which we have already discussed; the rules for the new language constructs: barriers and the compare-and-swap; and finally an extra rule for dereferencing ($\mathsf{rd}^{\circ}_{p,v}$). Notice that both rules for dereferencing have the exact same semantics at this point. However, they are not the same when considering execution of parallel programs. The difference in the semantics of these two read actions will become apparent in the parallel semantics that we will present shortly.

Notice that in the Figure 3.2 we have separated the transitions of the two sublanguages we will consider later. In the part 3.3a we have the rules of the common part of both sublanguages. The synchronization rules for the high-level

$$\begin{array}{rcl}
\mathbf{E}[(\lambda x e v)] & \xrightarrow{\beta_v} & \mathbf{E}[(\lambda v^{?}\ \{x \mapsto v\}e\ v)] \\
\mathbf{E}[(\lambda v^{?} e v)] & \xrightarrow{\beta} & \mathbf{E}[e] \\
\mathbf{E}[(\mathsf{if}\ tt\ \mathsf{then}\ e_0\ \mathsf{else}\ e_1)] & \xrightarrow{\swarrow} & \mathbf{E}[e_0] \\
\mathbf{E}[(\mathsf{if}\ ff\ \mathsf{then}\ e_0\ \mathsf{else}\ e_1)] & \xrightarrow{\searrow} & \mathbf{E}[e_1] \\
\mathbf{E}[(\mathsf{ref}\, v)] & \xrightarrow{\nu_{p,v}} & \mathbf{E}[p] \\
\mathbf{E}[(p := v)] & \xrightarrow{\mathsf{wr}_{p,v}} & \mathbf{E}[()] \\
\mathbf{E}[(!\, p)] & \xrightarrow{\mathsf{rd}_{p,v}} & \mathbf{E}[v] \\
\mathbf{E}[(!\, p)] & \xrightarrow{\mathsf{rd}^{\circ}_{p,v}} & \mathbf{E}[v] \\
\mathbf{E}[(\mathsf{thread}\, e)] & \xrightarrow{\mathsf{spw}_e} & \mathbf{E}[()]
\end{array}$$

(a) Common Sublanguage

$$\begin{array}{rcl}
\mathbf{E}[(v\backslash\ell)] & \xrightarrow{\widehat{\ell}} & \mathbf{E}[v] \\
\mathbf{E}[(\mathsf{with}\ \ell\ \mathsf{do}\ e)] & \xrightarrow{\widehat{\ell}} & \mathbf{E}[(e\backslash\ell)]
\end{array}$$

(b) High-level Language Synchronization

$$\begin{array}{rcl}
\mathbf{E}[(\mathsf{cas}\ p)] & \xrightarrow{\mathsf{cas}_{p,v}} & \mathbf{E}[v] \\
\mathbf{E}[(\mathsf{cas}\ p)] & \xrightarrow{\mathsf{cas}^{\circ}_{p,v}} & \mathbf{E}[v] \\
\mathbf{E}[\langle \mathtt{rd}|\mathtt{rd}\rangle] & \xrightarrow{\mathtt{rr}} & \mathbf{E}[()] \\
\mathbf{E}[\langle \mathtt{rd}|\mathtt{wr}\rangle] & \xrightarrow{\mathtt{rw}} & \mathbf{E}[()] \\
\mathbf{E}[\langle \mathtt{wr}|\mathtt{rd}\rangle] & \xrightarrow{\mathtt{wr}} & \mathbf{E}[()] \\
\mathbf{E}[\langle \mathtt{wr}|\mathtt{wr}\rangle] & \xrightarrow{\mathtt{ww}} & \mathbf{E}[()]
\end{array}$$

(c) Low-level Language Synchronization

Figure 3.2: Semantics of Single Expressions: ANF-Language

language with locks are presented in subfigure 3.3b and the ones for the low-level language with barriers and compare-and-swap are presented in subfigure 3.3c.

A property that should be intuitively obvious is that an expression in ANF when reduced by means of these rules (of Figure 3.2) produces an ANF expression as well. The following remark provides a formal statement for that observation.

Remark 3.4. *If e is an expression of the language (that is, in ANF), and $e \to e'$ by the rules of Figure 3.2, then e' is well-defined, that is, e' also in ANF.*

We can now justify the fact that the transformation $\mathcal{T}$ does not affect the semantics of the expression language in the following lemma.

Lemma 3.5. *Let e be an expression of the language of Chapter 2, and suppose that $e \xrightarrow{*} e'$ by the semantics of Figure 2.2, then we have that $\mathcal{T}(e) \xrightarrow{*} \mathcal{T}(e')$ by the semantics of Figure 3.2 (where we disregard $\mathsf{rd}^{\circ}_{p,v}$ actions).*

We spare the reader of the proof of this evident fact, and we simply observe that the equivalent of a β reduction on the calculus of Figure 2.2 is composed of two reductions in the transitions of Figure 3.2; namely a β_v reduction immediately

followed by a β reduction. For instance, coming back to the previous example for the expression $(\lambda x e v)$, where the full reduction was:

$$(\lambda x e v) \xrightarrow{\beta_v} (\lambda v^?\ \{x \mapsto v\}e\ v) \xrightarrow{\beta} \{x \mapsto v\}e$$

we can clearly see that this is equivalence to a single β reduction in the standard λ-calculus.

3.1.1 Speculative Semantics

The key ingredient of the semantics of speculations is the *speculation context*, which is a generalization of the traditional evaluation context:

$$\begin{array}{rcll} \mathbf{\Sigma} & ::= & [] \mid (v\mathbf{\Sigma}) \mid (\mathbf{\Sigma}\backslash\ell) & \textit{speculation contexts} \\ & \mid & \boxed{(\lambda x \mathbf{\Sigma} e)} \mid \boxed{(\lambda v^? \mathbf{\Sigma} e)} & \end{array}$$

It is straightforward to see that speculation contexts are more general than the typical evaluation contexts, since there are two extra productions here – included inside a box. The idea of these additional contexts is that they allow us to compute within the body of a function before the actual application takes place. Notice that these contexts require the function to be applied, otherwise a λ-abstraction has to be considered as a value, and computing inside it should be avoided as we seek to model a call-by-value imperative λ-calculus. Let us now see how we achieve computing in advance by means of these new contexts. For instance consider the expression $(!p)\,;(q := tt)$ where p and q are arbitrary references. As we said earlier this is syntactic sugar for the expression $(\lambda x\,(q := tt)\,(!p))$. Using the speculative context $(\lambda x[](!p))$ we can see that the redex $(q := tt)$ is capable of being reduced, achieving thus an effective reordering of the reading of p and the writing of q. In particular with an expression of the form $(\mathsf{let}\ x = e_0\ \mathsf{in}\ e_1)$ we can start by reducing within e_1, before or in parallel with the reduction of e_0. This is in some sense similar to the construct $(\mathsf{let}\ x = \mathsf{future}\, e_0\ \mathsf{in}\ e_1)$ of [Flanagan and Felleisen, 1995].

Despite the simplicity of this technique, we will shortly see how the interaction of speculation contexts with the prediction of arguments in applications renders powerful speculation behaviors, like computing inside a branching construct $(\mathsf{let}\ x = e_0\ \mathsf{in}\ (\mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2))$ before actually evaluating its condition for example. A similar formalization of speculative contexts was presented in [Boudol and Petri, 2010], in which additional speculation contexts where added for speculating within a conditional branch. Here we will use a different, and perhaps simpler, approach to achieve the same result. The main difference between our work and that of [Boudol and Petri, 2010], is the novel β_v reduction jointly with the adoption of a language in ANF, which proved to be crucial for the results of Chapter 4.

As we anticipated in the introduction we will define which speculations are *valid*, and which are not. To that end, we need not only to know which action is being performed (which we already included in the semantics by means of the action label) but also *where* in the expression the action is being produced; in other words, the exact position of the redex being reduced in the "global" expression. To achieve it, we will also annotate the transitions rules with that information by incorporating the notion of *occurrences* – which shall not be

confused with the occurrences of the previous chapter. Occurrences here are sequences over a set of symbols denoting a path from the root of the expression to the redex that is to be evaluated in the next step. Since the occurrence of an event is derived from the context (either evaluation context or speculation context) in which it is produced, it should be intuitive that we will have two sets of occurrence symbols as well. Symbols generated by evaluation contexts are sampled from the set $\mathcal{O}cc$, and symbols generated by speculative contexts are sampled from the set $\mathcal{SO}cc$, both of them being defined below. Naturally $\mathcal{SO}cc$ includes $\mathcal{O}cc$. Then an occurrence is a sequence occ over the set $\mathcal{SO}cc$:

$$\begin{array}{rcl} \mathcal{O}cc & = & \{(\_[]), ([]\backslash\ell)\} \\ \mathcal{SO}cc & = & \mathcal{O}cc \cup \{(\lambda\_[]\_)\} \end{array}$$

Notice that these symbols correspond with the contexts defined above. In particular we can recover the path $@\mathbf{\Sigma}$ to the hole in a speculation context $\mathbf{\Sigma}$ by means of the following inductive definition:

$$\begin{array}{rcl} @[] & = & \varepsilon \\ @(v\mathbf{\Sigma}) & = & (\_[]) \cdot @\mathbf{\Sigma} \\ @(\lambda x\mathbf{\Sigma}e) & = & (\lambda\_[]\_) \cdot @\mathbf{\Sigma} \\ @(\lambda v^?\mathbf{\Sigma}e) & = & (\lambda\_[]\_) \cdot @\mathbf{\Sigma} \\ @(\mathbf{\Sigma}\backslash\ell) & = & ([]\backslash\ell) \cdot @\mathbf{\Sigma} \end{array}$$

The occurrences $occ \in \mathcal{O}cc^*$ are called *normal* in contrast with occurrences that strictly belong to the set $\mathcal{SO}cc^*$ that we will name *speculative*. Normal computations – that is computations that do not speculate – involve only normal occurrences in their labels. We will use the symbols o and occ to range over the set of occurrences (i.e. $\mathcal{SO}cc^*$).

Before presenting the full semantics we have to augment the definition of redexes to include a new case of function application generalizing (λxev). Since we allow the speculation of argument values – we recall that this is the purpose of the β_v reduction – we will let a function application perform the substitution of the bound variable with a predicted value at any point in the computation. Later the β reduction will only succeed if the predicted value for the argument coincides with the value resulting from reducing the argument. Importantly, to reduce any function application we have to extend the notion of redex to include applications that do not necessarily have a value in the argument position as follows:

$$r \quad ::= \quad \ldots \mid (\lambda x e_0 e_1)$$

We have now all the notions required to present local speculations in Figure 3.3. Local speculations are defined as a small step semantics, as we did many times already, but we add to the transitions $(e \xrightarrow[o]{a} e')$ a label o representing the occurrence where the reduction takes place. We need not give detailed explanations on these transitions as they are the similar to the ones of Figure 3.2.

We can then establish the following standard property:

Lemma 3.6. *If* $e \xrightarrow[o]{a} e'$ *then* $\{x \mapsto v\}e \xrightarrow[o]{a} \{x \mapsto v\}e'$ *for any* v.

$$\begin{array}{rcl}
\mathbf{\Sigma}[(\lambda x e_0 e_1)] & \xrightarrow[@\mathbf{\Sigma}]{\beta_v} & \mathbf{\Sigma}[(\lambda v^?\ \{x \mapsto v\} e_0\ e_1)] \\
\mathbf{\Sigma}[(\lambda v^? e v)] & \xrightarrow[@\mathbf{\Sigma}]{\beta} & \mathbf{\Sigma}[e] \\
\mathbf{\Sigma}[(\mathsf{if}\ tt\ \mathsf{then}\ e_0\ \mathsf{else}\ e_1)] & \xrightarrow[@\mathbf{\Sigma}]{\swarrow} & \mathbf{\Sigma}[e_0] \\
\mathbf{\Sigma}[(\mathsf{if}\ ff\ \mathsf{then}\ e_0\ \mathsf{else}\ e_1)] & \xrightarrow[@\mathbf{\Sigma}]{\searrow} & \mathbf{\Sigma}[e_1] \\
\mathbf{\Sigma}[(\mathsf{ref}\ v)] & \xrightarrow[@\mathbf{\Sigma}]{\nu_{p,v}} & \mathbf{\Sigma}[p] \\
\mathbf{\Sigma}[(p := v)] & \xrightarrow[@\mathbf{\Sigma}]{\mathsf{wr}_{p,v}} & \mathbf{\Sigma}[()] \\
\mathbf{\Sigma}[(!\, p)] & \xrightarrow[@\mathbf{\Sigma}]{\mathsf{rd}_{p,v}} & \mathbf{\Sigma}[v] \\
\mathbf{\Sigma}[(!\, p)] & \xrightarrow[@\mathbf{\Sigma}]{\mathsf{rd}^{\mathsf{o}}_{p,v}} & \mathbf{\Sigma}[v] \\
\mathbf{\Sigma}[(\mathsf{thread}\, e)] & \xrightarrow[@\mathbf{\Sigma}]{\mathsf{spw}_e} & \mathbf{\Sigma}[()]
\end{array}$$

(a) Common Sublanguage

$$\begin{array}{rcl}
\mathbf{\Sigma}[(v \backslash \ell)] & \xrightarrow[@\mathbf{\Sigma}]{\widehat{\ell}} & \mathbf{\Sigma}[v] \\
\mathbf{\Sigma}[(\mathsf{with}\ \ell\ \mathsf{do}\ e)] & \xrightarrow[@\mathbf{\Sigma}]{\widehat{\ell}} & \mathbf{\Sigma}[(e \backslash \ell)]
\end{array}$$

(b) High-level Language Synchronization

$$\begin{array}{rcl}
\mathbf{\Sigma}[(\mathsf{cas}\ p)] & \xrightarrow[@\mathbf{\Sigma}]{\mathsf{cas}_{p,v}} & \mathbf{\Sigma}[v] \\
\mathbf{\Sigma}[(\mathsf{cas}\ p)] & \xrightarrow[@\mathbf{\Sigma}]{\mathsf{cas}^{\mathsf{o}}_{p,v}} & \mathbf{\Sigma}[v] \\
\mathbf{\Sigma}[\langle \mathtt{rd} | \mathtt{rd} \rangle] & \xrightarrow[@\mathbf{\Sigma}]{\mathtt{rr}} & \mathbf{\Sigma}[()] \\
\mathbf{\Sigma}[\langle \mathtt{rd} | \mathtt{wr} \rangle] & \xrightarrow[@\mathbf{\Sigma}]{\mathtt{rw}} & \mathbf{\Sigma}[()] \\
\mathbf{\Sigma}[\langle \mathtt{wr} | \mathtt{rd} \rangle] & \xrightarrow[@\mathbf{\Sigma}]{\mathtt{wr}} & \mathbf{\Sigma}[()] \\
\mathbf{\Sigma}[\langle \mathtt{wr} | \mathtt{wr} \rangle] & \xrightarrow[@\mathbf{\Sigma}]{\mathtt{ww}} & \mathbf{\Sigma}[()]
\end{array}$$

(c) Low-level Language Synchronization

Figure 3.3: Speculative Semantics of Single Expressions

Sequences of speculative computations are going to be named *speculations* hereafter, as defined below.

Definition 3.7 (Speculations)**.** *A* speculation *from an expression e to an expression e' is a (possibly empty) sequence $\sigma = \left(e_i \xrightarrow[a_i]{o_i} e_{i+1}\right)_{0 \leqslant i \leqslant n}$ of speculation steps such that $e_0 = e$ and $e_n = e'$. This is written $\sigma : e \xrightarrow{*} e'$. The empty speculation (with $e' = e$) is denoted ε. The sequence σ is* normal *iff for all i the occurrence o_i is normal. The* concatenation *$\sigma \cdot \sigma' : e \xrightarrow{*} e'$ of σ and σ' is only defined (in the obvious way) if σ ends on the expression e'' where σ' originates.*

Notice that a normal speculation proceeds in program order up to the reordering of β_v actions (which is of no consequence), evaluating redexes inside evaluation contexts – not speculation contexts. Let us see two examples of speculations – omitting some labels, just mentioning the actions[2]:

Example 3.8.

$$r := (!\, p)\,;q := tt \xrightarrow{\mathsf{wr}_{q,tt}} r := (!\, p)\,;() \xrightarrow{\mathsf{rd}_{p,tt}} r := tt\,;() \xrightarrow{\mathsf{wr}_{r,tt}} ()\,;() \xrightarrow{\beta} ()$$

Here we speculate by reducing first the assignment $q := tt$, which would normally take place after reading p and updating r; we can say that the assignment is issued out of order. Moreover, even though the second step is normal, we guess the value read from memory location p, since at this stage of the semantics there is no store to retrieve the value of the read.

An interesting consequence of the added speculative contexts and the prediction of argument values is that we can now "fabricate" redexes, reduce them, and later validate them. This would be typically the case of a branch prediction [Smith, 1981]. To illustrate this we consider the expression:

$$p := tt\,;(\mathsf{let}\ x = (!q)\ \mathsf{in}\ (\mathsf{if}\ x\ \mathsf{then}\ (r := ff)\ \mathsf{else}\ (r := tt)))$$

where, assuming that every reference is initially set to ff, we want to perform the assignment $r := ff$, before evaluating the condition and before reducing the assignment to p. Then we can apply the following reduction to obtain the desired result:

Example 3.9.

$$\begin{array}{r}
p := tt\,;(\mathsf{let}\ x = (!q)\ \mathsf{in}\ (\mathsf{if}\ x\ \mathsf{then}\ (r := ff)\ \mathsf{else}\ (r := tt))) \xrightarrow{\beta_{ff}} \\
p := tt\,;(\mathsf{let}\ ff^{?} = (!q)\ \mathsf{in}\ (\mathsf{if}\ ff\ \mathsf{then}\ (r := ff)\ \mathsf{else}\ (r := tt))) \xrightarrow{\searrow} \\
p := tt\,;(\mathsf{let}\ ff^{?} = (!q)\ \mathsf{in}\ (r := tt)) \xrightarrow{\mathsf{wr}_{r,tt}} \\
p := tt\,;(\mathsf{let}\ ff^{?} = (!q)\ \mathsf{in}\ ()) \xrightarrow{\mathsf{rd}_{q,ff}} \\
p := tt\,;(\mathsf{let}\ ff^{?} = ff\ \mathsf{in}\ ()) \xrightarrow{\beta} \\
p := tt\,;() \xrightarrow{\mathsf{wr}_{p,tt}} ();() \xrightarrow{*} ()
\end{array}$$

[2]We recall that the translation to ANF is implicit here.

Importantly, the combination of a language in ANF, with argument speculation enables us to eliminate some dependencies present in the λ-calculus related to redex creation. Let us consider the following simple example to clarify this issue:

$$(\lambda x\,(!x)\,(q := \mathit{ff}\,;p)$$

where we have a function that expects a memory location as its argument to dereference it, and this function is applied to an expression that firstly updates the reference q with value ff and then simply returns the reference p (to be consumed by the function). It is clear that in the normal semantics this expression first assigns to the reference q and then dereferences p. However we would like to reorder these memory accesses – which is a common relaxation present in many relaxed memory models – since p and q are different references. If we consider the standard λ-calculus, it is not possible perform the dereferencing of p first since $(!x)$ is not a redex. On the other hand, by means of the β_v action we are allowed to "predict" that the reference that the λ-abstraction receives as its argument will be p and we can substitute it in the body of the function. Then later we can compute inside the function, dereferencing p and finally validating the prediction. The transitions taken, where we omit the occurrences, are:

$$\begin{aligned}\lambda x\,(!x)\,(q := \mathit{ff}\,;p) \;&\xrightarrow{\beta_p}\; \lambda p^?\,(!p)\,(q := \mathit{ff}\,;p) \;\xrightarrow{\mathsf{rd}_{p,\mathit{ff}}}\\ &\lambda p^?\,\mathit{ff}\,(q := \mathit{ff}\,;p) \;\xrightarrow{\mathsf{wr}_{q,\mathit{ff}}}\; \lambda p^?\,\mathit{ff}\,(()\,;p) \;\xrightarrow{*}\; \lambda p^?\,\mathit{ff}\,p \;\xrightarrow{\beta}\; \mathit{ff}\end{aligned}$$

Notice that by means of this simple technique we have actually achieved reordering the actions. This behavior can be compared to the effects of write-buffering discussed in the previous chapter. In the case of a write-buffering architecture, one can think that the write to q is actually performed first, but its contents remain in the buffer until after the read of p has finished. Thus, one could imagine that the actions of writing q and reading p have been reordered in the sense of the speculations of this chapter. We will see in Chapter 4 that this technique turns out to be fundamental to capture the specific behavior of some existing relaxed memory models.

To conclude our presentation of the single thread semantics let us briefly discuss the semantics of the locking construct. One might observe that locks here do not enforce an ordering among the actions that precede the locking construct and those that follow it[3]. In fact we can observe that in an expression of the form $(\mathsf{with}\ \ell\ \mathsf{do}\ (!p))\,;(!q)$ one can reduce the first the redex $(!q)$ and then continue by acquiring the lock and so on and so forth. This kind of "speculation" can be surprising to common programmers, but it is not new in the literature of relaxed memory models. The term *roach motel semantics* for synchronization has been used for Java in [Manson et al., 2005; Ševčík and Aspinall, 2008] to describe this kind of behavior, and a similar account for C++ with Pthreads is discussed in [Boehm, 2007]. Notice, however, in the example above that the redex $(!p)$ cannot be reduced before acquiring the lock ℓ since there is no speculation or evaluation context that inspects the contents inside of a mutual exclusion construct. Indeed, this guarantees that these locks implement the correct mutual exclusion of the critical sections using the same lock.

[3]However they impose an order among the actions included in the critical section and the acquisition and release of the lock.

Perhaps it would be more natural, from a sequentially consistent point of view, to disallow this kind of speculation but it would make our calculus less general with no added benefit. It is easy to see that all the results that we will present in the sequel still hold if we add this extra restriction. We will discuss this choice again when considering the semantics of parallel threads in the next subsection.

3.1.2 The Global Semantics

Once more, we will begin by presenting the *configurations* that amalgamate threads and allow them to communicate. These configurations have the following shape:

$$C = (S, L, T)$$

where the store S and the lock context L are similar to those of the presented previously in Section 2.2. Obviously the lock context L is of no use for λ-barrier programs and could be omitted in that case. We assume an infinite set $\mathcal{T}id$ of thread identifiers. Then, the thread system T is a mapping from a finite set $\mathsf{dom}(T)$ of thread names (or thread identifiers), a proper subset of $\mathcal{T}id$, to expressions representing the threads. If $\mathsf{dom}(T) = \{t_1, \ldots, t_n\}$ and $T(t_i) = e_i$ we also write T as

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

That is, in contrast to the thread systems of the previous chapter, here we consider the standard assumption that thread systems are commutative and associative.

As usual, we shall consider only *well-formed* configurations, meaning that any reference that occurs somewhere in the configuration belongs to the domain of the store, that is, it is bound to a value in the memory – we shall not define this property, which is preserved in the operational semantics, more formally. For instance, if e is an expression of the source language, any initial configuration $(\emptyset, \emptyset, (t, e))$ is well-formed. The speculative computations are made of transitions that have the form

$$C \xrightarrow[t,o]{a} C'$$

indicating the action $a \in \mathcal{A}ct$ that is performed, the thread $t \in \mathcal{T}id$ that performs it, and the occurrence $o \in \mathcal{SO}cc$ signaling where it is performed within the thread (again, these labels are just annotations, introduced for technical convenience, but they do not entail any constraint on the semantics). At each step, a speculation issued by one thread is recorded, provided that *the global state agrees* with the action that is performed. For instance the value guessed by a read for a reference must be the value on the store for that reference, and similarly acquiring a lock can only be done if the lock is free. In the semantics of the global configuration we use the notation $\mathsf{FRef}(e)$ that stands for the set of references appearing in the expression e.

The semantics is presented in Figure 3.4 where we distinguish two cases, depending on whether the action spawns a new thread or not. The condition $(*)$ of the upper rule is given below, where the omission of the new store S' and/or the new lock pool L' means that these remain unchanged (i.e. $S' = S$

$$\frac{e \xrightarrow[o]{a} e'}{(S, L, (t, e)\|T) \xrightarrow[t,o]{a} (S', L', (t, e')\|T)}\ (*)$$

$$\frac{e \xrightarrow[o]{\mathsf{spw}_{e''}} e'}{(S, L, (t, e)\|T) \xrightarrow[t,o]{a} (S, L, (t, e')\|(t'', e'')\|T)}\ t'' \notin \mathsf{dom}(T)$$

Figure 3.4: Global Speculative Semantics

and/or $L' = L$):

$$(*)\begin{cases} a \in \left\{ \begin{array}{c} \beta, \swarrow, \searrow, \mathsf{cas}^{\circ}_{p,ff} \\ \mathtt{rr}, \mathtt{rw}, \mathtt{wr}, \mathtt{ww} \end{array} \right\} & \Rightarrow \quad S' = S \ \&\ L' = L \\ a \in \{\beta_v, \mathsf{rd}^{\circ}_{p,v}\} & \Rightarrow \quad \mathsf{FRef}(v) \subseteq \mathsf{dom}(S) \\ a = \nu_{p,v} & \Rightarrow \quad p \notin \mathsf{dom}(S) \ \&\ S' = S \cup \{p \mapsto v\} \\ a = \mathsf{rd}_{p,v} & \Rightarrow \quad v = S(p) \\ a = \mathsf{wr}_{p,v} & \Rightarrow \quad p \in \mathsf{dom}(S) \ \&\ S' = S\{p \leftarrow v\} \\ a = \overset{\frown}{\ell} & \Rightarrow \quad L' = L \cup \{\ell\} \\ a = \overset{\frown}{\ell} & \Rightarrow \quad L' = L - \{\ell\} \\ a = \mathsf{cas}_{p,tt} & \Rightarrow \quad S(p) = ff \ \&\ S' = S\{p \leftarrow tt\} \\ a = \mathsf{cas}_{p,ff} & \Rightarrow \quad S(p) = tt \\ a = \mathsf{cas}^{\circ}_{p,tt} & \Rightarrow \quad S' = S\{p \leftarrow tt\} \end{cases}$$

Up to now the predictions performed by the β_v reductions were unrestricted. However some of them should not be considered as valid predictions. Consider for instance the following expression:

$$(\lambda x\,(!x)\,(q := ff\ ;(\mathsf{ref}\ ff))$$

Indeed, we cannot speculate the value of the reference to be created before actually creating it, by the condition $\mathsf{FRef}(v) \subseteq \mathsf{dom}(S)$ in the condition $(*)$. This is a reasonable assumption, since it does not affect the set of behaviors of the semantics, and the *well-formedness* of configurations would be compromised if we did not adopt it here.

An important feature of this global semantics is that the value obtained for a read of the form $\mathsf{rd}^{\circ}_{p,v}$ does not need to coincide with the value in the store for that reference. In that sense it is not constrainted by the configuration. Indeed, as it is now, any value (that does not contain new references) can be returned by these "speculative" reads. Some constraints regarding these actions will be added in the definition of validity later. Consequently, different validity criteria can provide different semantics for these reads; for instance, we could consider reads that obtain their value from a write buffer, as we will do in the next chapter, or reads that obtain their value in a noncoherent cache, etc.

We can finally provide a formal definition of *speculative computations* that includes all the threads in coordination with the store and the lock pool in a single configuration.

Definition 3.10 (Speculative Computations)**.** *A* speculative computation *from a configuration C to a configuration C' is a (possibly empty) sequence γ of steps $\left(C_i \xrightarrow[a_i]{t_i,o_i} C_{i+1}\right)_{0\leqslant i\leqslant n}$ in the speculative operational semantics such that $C_0 = C$ and $C_n = C'$. This is written $\gamma : C \xrightarrow{*} C'$. The empty computation is denoted ε. The* concatenation *$\gamma \cdot \gamma' : C \xrightarrow{*} C'$ is only defined (in the obvious way) if γ ends on the configuration C'' where γ' originates, that is $\gamma : C \xrightarrow{*} C''$ and $\gamma' : C'' \xrightarrow{*} C'$. The computation $\gamma = \left(C_i \xrightarrow[t_i,o_i]{a_i} C_{i+1}\right)_{0\leqslant i\leqslant n}$ is* normal *if for all i the occurrence o_i is normal.*

In particular we will be interested in discussing about speculative computations whose behaviors correspond to sequentially consistent executions of the program, that is the speculative semantics coincides with the interleaving semantics. To achieve that we need to define coherent computations in which all read actions return the latest value written to the same reference in the computation (i.e. the value in the store).

Definition 3.11 (Coherent Speculative Computation)**.** *A speculative computation γ is qualified as* coherent *if whenever $\gamma = \gamma' \cdot (C \xrightarrow[t,o]{a} C') \cdot \gamma''$ with $C = (S, L, T)$ then $a = \mathsf{rd}^{\circ}_{p,v} \Rightarrow S(p) = v$, $a = \mathsf{cas}^{\circ}_{p,tt} \Rightarrow S(p) = ff$ and $a = \mathsf{cas}^{\circ}_{p,ff} \Rightarrow S(p) = tt$.*

As an obvious consequence of the above definition, we can replace every $\mathsf{rd}^{\circ}_{p,v}$ and $\mathsf{cas}^{\circ}_{p,v}$ action in a coherent speculative computation with its $\mathsf{rd}_{p,v}$ or $\mathsf{cas}_{p,v}$ counterpart to get an equally legitimate computation.

Notice that for normal coherent computations of the λ-lock language if we replace the speculation contexts $\mathbf{\Sigma}$ by standard evaluation contexts $\mathbf{E}$ (recall that we are considering normal computations) in the semantics of Figure 3.4 we obtain the *interleaving semantics* of the previous Chapter 2.3 considering β_v steps as a silent steps. Once more, we regard semantics of *normal coherent computations* as the reference semantics from the programmer's view point.

Even though our definition of speculative computations ensures that the values read from the memory are correctly guessed, some speculation sequences are still wrong, like – omitting the occurrences:

$$(\{p \mapsto ff\}, \emptyset, (!p)\,;(p := tt)) \xrightarrow{\mathsf{wr}_{p,tt}} (\{p \mapsto tt\}, \emptyset, (!p)\,;()) \xrightarrow{\mathsf{rd}_{p,tt}} (\{p \mapsto tt\}, \emptyset, tt\,;()) \xrightarrow{*} ()$$

where the read and write actions are clearly conflicting, since they access the same reference p. Obviously, this speculation violates the sequential semantics of the program, since in the normal execution of this program the read always obtains the initial value of p in the store, in this case ff. It is obvious that this reordering of conflicting actions should be disallowed to preserve the sequential semantics of the program (and of each thread in the case of a parallel program).

Precluding the kind of behavior considered above is the purpose of the definition of *validity* that we present in the next section. To that end, we shall need the following technical definition, which formalizes the contribution of each thread to a speculative computation:

Definition 3.12 (Projection)**.** *Given a thread identifier t, the* projection $\gamma|_t$ *of a speculative computation γ on thread t is defined as follows, by induction on γ:*

$$\gamma|_t \triangleq \begin{cases} \varepsilon & \text{if } \gamma = \varepsilon \\ e \xrightarrow[o]{a} e' \cdot (\gamma'|_t) & \text{if } \gamma = (C \xrightarrow[t',o]{a} C') \cdot \gamma' \ \&\ t' = t \\ & \qquad \&\ C = (S, L, (t,e) \| T) \ \&\ C' = (S', L', (t,e') \| T) \\ \gamma'|_t & \text{if } \gamma = (C \xrightarrow[t',o]{a} C') \cdot \gamma' \ \&\ t' \neq t \end{cases}$$

It is easy to check that this is indeed well-defined, that is:

Remark 3.13. *For any speculative computation γ and name t, the projection $\gamma|_t$ is a speculation.*

3.1.3 Valid Speculations

We shall qualify a speculative computation as *valid* in the case where each of its projections is *equivalent* in some sense to a normal evaluation. That is, a speculative computation is valid if it only involves thread speculations that correctly predict the values read from the memory (for normal reads, i.e. $\mathsf{rd}_{p,v}$ actions), and preserves, up to some equivalence, the normal program order. Moreover, for the case of $\mathsf{rd}^{\mathsf{o}}_{p,v}$ actions (self-fulfilled reads) we will require an extra condition on the speculation, guaranteeing that actually there is no synchronization in the speculation that could hinder that kind of read. In other words, the validity criterion is *purely local* to each thread, namely, each thread's speculation should be "equivalent" to a sequential execution of the thread[4]. We will use a relation that is similar to the *permutation of transitions equivalence* introduced by Berry and Lévy [Berry and Lévy, 1979; Lévy, 1980] that we have already discussed in the previous chapter. Intuitively, this relation says that permuting independent steps in a speculation results in "the same" speculation, and that such independent steps could actually be performed in parallel. It is clear, for instance, that actions performed at disjoint occurrences can be done in any order, provided that they are not conflicting accesses to the same memory location (the conflict relation will be defined below). This applies for instance to

$$r := (!p)\,;q := tt \quad \xrightarrow{\mathsf{wr}_{q,tt}} \quad r := (!p)\,;() \quad \xrightarrow{\mathsf{rd}_{p,tt}} \quad r := tt\,;() \quad \xrightarrow{*} \quad ()$$

from Example 3.8, which as we will see is equivalent to the computation

$$r := (!p)\,;q := tt \quad \xrightarrow{\mathsf{rd}_{p,tt}} \quad r := tt\,;q := tt \quad \xrightarrow{\mathsf{wr}_{q,tt}} \quad r := tt\,;() \quad \xrightarrow{*} \quad ()$$

However, in order to do so we need to "identify" the first step in the first speculation with the second one in the latter, and vice versa. To this end, given a speculation step $e \xrightarrow[o]{a} e'$ and an occurrence o' in e, we define the *residual* of o' after this step, that is the occurrence, if any, that points to the same subterm (if any) as o' pointed to in e. The notion of a residual here is much simpler than in the λ-calculus (see [Lévy, 1980]), because an occurrence is never duplicated, since we do not compute inside a value (except in a function applied to an

[4]This appears to be the standard – though implicit – validity criterion in the literature on speculative execution of sequential programs.

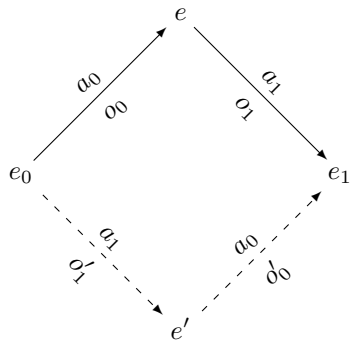


Figure 3.5: Diagram of Lemma 3.15

argument). Here the residual of an occurrence after a speculation step will be either undefined, or a single occurrence.

We actually need to know the action a that is performed, the occurrence o where it is performed and the expression e to which the occurrences belong in order to define the residual of o' after such a step. Notice that in [Boudol and Petri, 2010] we did not parameterize the relation with the expression e. The reason why we need it here, is that speculating the argument of functions can create new redexes (precisely its intention) in the body of the function, that are not present in the original expression. For example in the transition $(\lambda x(!x)(!q)) \xrightarrow{\beta_p} (\lambda p^?(!p)(!q))$ the redex $(!p)$ is created by the action β_p.

We will use the notation $e@o$ to retrieve the subexpression of e at location o. This is defined in the obvious way:

$$e@o \triangleq \begin{cases} e & \text{if } o = [] \\ e_1@o' & \text{if } e = (e_0 e_1) \ \&\ o = (\_[]) \cdot o' \\ e'@o' & \text{if } e = (e'\backslash\ell) \ \&\ o = ([]\backslash\ell) \cdot o' \\ e_0@o' & \text{if } e \in \{(\lambda x e_0 e_1), (\lambda v^? e_0 e_1)\} \ \&\ o = (\lambda\_[]\_) \cdot o' \end{cases}$$

And we can then define $o'/_e(a, o)$ as follows:

Definition 3.14 (Residual of an occurrence after a step)**.**

$$o'/_e(a,o) \triangleq \begin{cases} o' & \textit{if } o \not\leq o', \textit{ or} \\ & \quad o' = o \cdot (\lambda\_[]\_) \cdot o'' \ \&\ a = \beta_v \ \&\ e@o' \textit{ is a redex, or} \\ & \quad o' = o \cdot (\_[]) \cdot o'' \ \&\ a = \beta_v \\ o \cdot o'' & \textit{if } o' = o \cdot (\lambda\_[]\_) \cdot o'' \ \&\ a = \beta \\ \texttt{undef} & \textit{otherwise} \end{cases}$$

In the following we write $o'/_e(a, o) \equiv o''$ to mean that the residual of o' after (a, o) is defined, and is o''. Notice that if $o'/_e(a, o) \equiv o''$ with $o' \in \mathcal{O}cc^*$ then $o'' = o'$ and $o \not\leq o'$.

We can now prove a property that is the core of the reordering relation (cf. the equivalence by permutations of Chapter 2) that we will use later to

define *valid* computations. The following lemma states that if two consecutive actions are not related by redex creation (meaning that they have corresponding residuals), then reordering the reductions in the computation leads to the same result as shown in Figure 3.5:

Lemma 3.15 (Reordering Lemma)**.** *If* $e_0 \xrightarrow[o_0]{a_0} e \xrightarrow[o_1]{a_1} e_1$ *with* $o_1 \equiv o_1'/_{e_0}(a_0, o_0)$ *and* $o_0' \equiv o_0/_{e_0}(a_1, o_1')$*, then there exists* e' *(unique up to* α*-conversion) such that*

$$e_0 \xrightarrow[o_1']{a_1} e' \xrightarrow[o_0']{a_0} e_1$$

Proof. By cases on the respective positions of o_0 and o_1'. Notice first that if $o_0 \not\leq o_1'$ and $o_1' \not\leq o_0$ (that is the occurrences are disjoint), then $o_1 = o_1' \equiv o_1/_{e_0}(a_0, o_0)$ and $o_0' = o_0 \equiv o_0/_{e_0}(a_1, o_1')$, and it is easy to see that the two speculations can be done in any order.

Let us assume that $o_0 < o_1'$. Since o_1 is well defined, according to the definition of $o_1'/_e(a_0, o_0)$ we have three possibilities:

- $o_1' = o_0 \cdot (\lambda\_[]\_) \cdot o_1''$ and $a_0 = \beta_v$ and $e@o_1'$ is a redex. In this case we have $e_0 = \mathbf{\Sigma}_0[(\lambda x e_0' \bar{e})]$ and $e = \mathbf{\Sigma}_0[(\lambda v^?(\{x \mapsto v\}e_0')\bar{e})]$. From the hypotheses we have $e_0' \xrightarrow[o_1'']{a_1} e_0''$, and using Lemma 3.6 we have that $\{x \mapsto v\}e_0' \xrightarrow[o_1'']{a_1} \{x \mapsto v\}e_0''$. We conclude then with $e' = \mathbf{\Sigma}_0[(\lambda x(\{x \mapsto v\}e_0'')\bar{e})]$ where verifying that the steps can be commuted is trivial.
- $o_1' = o_0 \cdot (\lambda\_[]\_) \cdot o_1''$ and $a_0 = \beta$. Then we have that $e_0 = \mathbf{\Sigma}_0[(\lambda v^? e_0' v)]$ and $e = \mathbf{\Sigma}_0[e_0']$. But from the hypotheses we also have $e_0' \xrightarrow[o_1'']{a_1} e_0''$. Then we have $e' = \mathbf{\Sigma}_0[(\lambda v^? e_0'' v)]$ and the conclusion is obvious.
- $o_1' = o_0 \cdot (\_[]) \cdot o_1''$ and $a_0 = \beta_v$. Then we know that $e_0 = \mathbf{\Sigma}_0[(\lambda x e_0' \bar{e})]$ and from the hypotheses $\bar{e} \xrightarrow[o_1'']{a_1} \bar{e}'$. The candidate for e' is then $\mathbf{\Sigma}_0[(\lambda x e_0' \bar{e}')]$ and the verification that the transitions commute is immediate.

The case of $o_1' < o_0$ is symmetric, and $o_0 = o_1'$ is impossible since o_1 is well defined. □

This commutation property is the basis for the definition of the reordering relation between computing steps: with the hypotheses of the Reordering Lemma, we shall regard the two speculations

$$e_0 \xrightarrow[o_0]{a_0} e \xrightarrow[o_1']{a_1} e_1 \quad \text{and} \quad e_0 \xrightarrow[o_1]{a_1} e' \xrightarrow[o_0']{a_0} e_1$$

as equivalent. However, this will not be so simple, since we have to ensure that the program order is preserved as regards accesses to a given memory location (unless these accesses are all reads). For instance, the speculation – again, omitting the occurrences:

$$p := tt\,; r := (!p) \xrightarrow{\mathsf{rd}_{p,ff}} p := tt\,; r := ff \xrightarrow{\mathsf{wr}_{p,tt}} ()\,; r := ff \to \ldots$$

should not be considered as valid, because it breaks the data dependency between the write and the read on p. As in the previous chapter, we need a

definition of conflicting actions. However, unlike the definition of the previous chapter, here we consider only conflicts that relate to read and write actions, disregarding synchronization related actions (i.e. locks). The constraints on reorderings imposed by synchronization actions will be considered in the definition of *dependency* which was not needed in the previous chapter, but will play a fundamental role in this one.

Let us define the set $\mathcal{MR}d_p$ of actions on reference p with read semantics that act on the memory (thus the $\mathcal{M}$):

$$\mathcal{MR}d_p \triangleq \{\mathsf{rd}_{p,v}, \mathsf{cas}_{p,v} \mid v \in \mathcal{V}al\}$$

and a similar definition for the actions with write semantics on p:

$$\mathcal{MW}r_p \triangleq \{\mathsf{wr}_{p,v} \mid v \in \mathcal{V}al\} \cup \{\mathsf{cas}_{p,tt}, \mathsf{cas}^{\circ}_{p,tt}\}$$

We repeat that this definition regards actions that read or write the contents of the store. We can see for example that the action $\mathsf{rd}^{\circ}_{p,v}$ even if it is a read action, does not concern the memory, and so it is not included in $\mathcal{MR}d_p$. The same applies to $\mathsf{cas}^{\circ}_{p,v}$ for the case of reads. That is because as regards the conflict relation, events that do not affect or are not affected by the contents of the store cannot be involved in conflicts.

And now we can present the definition of conflict

Definition 3.16 (Conflicting Actions)**.** *We define the conflict relation, denoted by #, to be the following binary relation on actions:*

$$\# \triangleq \bigcup_{p \in \mathcal{R}ef} (\mathcal{MW}r_p \times \mathcal{MW}r_p) \cup (\mathcal{MW}r_p \times \mathcal{MR}d_p) \cup (\mathcal{MR}d_p \times \mathcal{MW}r_p)$$

In this definition we can see that the actions $\mathsf{rd}^{\circ}_{p,v}$ and $\mathsf{cas}^{\circ}_{p,ff}$ are not involved, since their values are "guessed", and they do not access the store.

To cope with the particular reorderings of relaxed memory models we need to add constraints stating which actions cannot be reordered w.r.t. other actions. This is the purpose of the *dependency* relation.

We have seen in previous examples that reordering conflicting actions in a speculation in general violates the sequential semantics of the program. Indeed, we can also observe, from the definition of *Data-Race* 2.9 of Chapter 2 and the Asynchrony Lemma (2.15) of the same chapter, that conflicting actions also have implications on the global reordering of actions of different threads – similar definitions and results will be given for the speculative calculus of this chapter. On the other hand, the dependency relation only poses local restrictions on the reorderings of speculations. The moral difference between conflict and dependency resides in the fact that conflicting actions have not only a local meaning, but also an intrinsic global one, precisely captured by the hypotheses of the Asynchrony Lemma below, stating that conflicting actions of different threads when commuted generally lead to different results. The dependency relation, however, imposes restrictions that are only local to a thread, and it can greatly vary without breaking the soundness of the framework.

In fact, rather than defining a concrete dependency relation for the speculative calculus, let us leave it abstract by parameterizing our definitions with an arbitrary dependency relation. Actually the dependency relation cannot be totally arbitrary, some restrictions have to be imposed to it. It suffices to say,

for the moment that the conflict relation has to be included in the dependency relation. Let us make this intuition more formal by providing a definition of acceptable dependency relations. Let us denote by $\mathcal{C}as_p$ the set of compare-and-swap actions on reference p and by $\mathcal{M}em_p$ the set of memory accesses to reference p:

$$\mathcal{C}as_p \triangleq \{\mathsf{cas}_{p,v}, \mathsf{cas}^{\circ}_{p,v} \mid v \in \mathcal{V}al\}$$

$$\mathcal{M}em_p \triangleq \mathcal{C}as_p \cup \{\mathsf{rd}_{p,v}, \mathsf{wr}_{p,v} \mid v \in \mathcal{V}al\}$$

Definition 3.17 (Dependency Relation)**.** *A* dependency *relation $\mathcal{D}$ is a binary relation on actions satisfying*

$$\bowtie \subseteq \mathcal{D}$$

where

$$\bowtie \triangleq \# \cup \bigcup_{p \in \mathcal{R}ef} (\mathcal{C}as_p \times \mathcal{M}em_p) \cup (\mathcal{M}em_p \times \mathcal{C}as_p)$$

Intuitively $x \mathcal{D} y$ means that if the action x precedes the action y in a normal speculation, this execution should not be rearranged into a speculative one in which y precedes x. If $\mathcal{D} \subseteq \mathcal{D}'$ we say that $\mathcal{D}$ is weaker than $\mathcal{D}'$. Notice that we do not require the dependency relation to be symmetric. In particular, when considering the language with λ-barrier, we will see that the dependency relation is the key semantical aspect to model the behavior of barriers. We will soon provide further restrictions imposed to acceptable dependency relations for the two versions of the language we are considering, but let us now move on to the definition of validity using such a general notion of dependency.

We based the results of the previous chapter on the permutation equivalence between computations. Likewise, we will define a *reordering relation* between speculative computations here to capture the same intuition. However, we need to incorporate the concept of dependency that we have just introduced in our relation. Notice that since $\mathcal{D}$ is not assumed to be symmetric, we can no longer use an equivalence, we will have to content ourselves with a preorder, which fortunately is enough to draw all of our subsequent results.

Definition 3.18 (Reordering Relation)**.** *Given a dependency relation $\mathcal{D}$ we define a reordering relation between speculations, called* $\mathcal{D}$-reordering*, to be the least preorder $\propto^{\mathcal{D}}$ such that if $e_0 \xrightarrow[o_0]{a_0} e \xrightarrow[o_1]{a_1} e_1$ with $o'_0 \equiv o_0/_e(a_1, o'_1)$ and $o'_1/_e(a_0, o_0) \equiv o_1$, and $\neg(a_0 \mathcal{D} a_1)$, then*

$$\sigma_0 \cdot e_0 \xrightarrow[o'_1]{a_1} e' \xrightarrow[o'_0]{a_0} e_1 \cdot \sigma_1 \;\; \propto^{\mathcal{D}} \;\; \sigma_0 \cdot e_0 \xrightarrow[o_0]{a_0} e \xrightarrow[o_1]{a_1} e_1 \cdot \sigma_1$$

where e' is determined as in the Reordering Lemma 3.15.

Notice that the important clause of this definition is $\neg(a_0 \mathcal{D} a_1)$. The moral here is that the computation to the right of the $\propto^{\mathcal{D}}$ relation is more *normal* than the one on the left, since the one on the left can be obtained by reordering nondependent (according to $\mathcal{D}$) actions. Let us consider an example to illustrate why this relation is not an equivalence, as it was the case in the previous chapter. To that end let us anticipate that barrier actions are not symmetric as regards their dependencies. Then, recalling that `wr` is the action that prevents a write action from being delayed past a read action, or conversely, prevents a read action from being speculated before a write action, we might have for example

that $(\mathsf{wr}_{p,v}\ \mathcal{D}\ \mathtt{wr})$ but $\neg(\mathtt{wr}\ \mathcal{D}\ \mathsf{wr}_{p,v})$. We have according to the reordering definition above

$$e_0 \xrightarrow[o_1']{\mathsf{wr}_{p,v}} e' \xrightarrow[o_1']{\mathsf{wr}_{p,v}} e_1 \propto^{\mathcal{D}} e_0 \xrightarrow[o_0]{\mathtt{wr}} e \xrightarrow[o_1]{\mathsf{wr}_{p,v}} e_1$$

but on the other hand we cannot reestablish the left-hand side computation from the right-hand side one, since we have $(\mathsf{wr}_{p,v}\ \mathcal{D}\ \mathtt{wr})$.

If $\sigma' \propto^{\mathcal{D}} \sigma$ we say that σ' is a $\mathcal{D}$-reordering of σ. We adopt the notation $\infty^{\mathcal{D}}$ for the symmetric closure of the $\propto^{\mathcal{D}}$ relation. That is:

$$\sigma \infty^{\mathcal{D}} \sigma' \iff \sigma \propto^{\mathcal{D}} \sigma' \text{ or } \sigma' \propto^{\mathcal{D}} \sigma$$

Notice that two speculations related by reordering have the same length. One can now verify that actually the speculation of Example 3.8 is a $\bowtie$-reordering of the "more normal" speculation:

$$r := \mathbf{!}\,p\,;q := tt \quad \xrightarrow{\mathsf{rd}_{p,tt}} \quad r := tt\,;q := tt \quad \xrightarrow{\mathsf{wr}_{q,tt}} \quad r := tt\,;() \quad \xrightarrow{*} \quad ()$$

Indeed, to have an entirely normal computation we should perform the write of r between the read of p and the following write of q. Similarly the speculation of Example 3.9 is a $\bowtie$-reordering of the normal speculation:

$$p := tt\,;(\mathsf{let}\ x = (\mathbf{!}q)\ \mathsf{in}\ (\mathsf{if}\ x\ \mathsf{then}\ (r := ff)\ \mathsf{else}\ (r := tt))) \xrightarrow{\mathsf{wr}_{p,tt}} \xrightarrow{*}$$
$$(\mathsf{let}\ x = (\mathbf{!}q)\ \mathsf{in}\ (\mathsf{if}\ x\ \mathsf{then}\ (r := ff)\ \mathsf{else}\ (r := tt))) \xrightarrow{\mathsf{rd}_{q,ff}} \xrightarrow{*}$$
$$(\mathsf{if}\ ff\ \mathsf{then}\ (r := ff)\ \mathsf{else}\ (r := tt)) \xrightarrow{\searrow} (r := tt) \xrightarrow{\mathsf{wr}_{r,tt}} ()$$

Before presenting the definition of the validity condition let us briefly consider the $\mathsf{rd}^{\circ}_{p,v}$ and $\mathsf{cas}^{\circ}_{p,v}$ actions in more detail. We give a particular interpretation to these actions in our calculus; namely that of a read that obtains the latest written value to the same reference by the same thread. Indeed, this is similar to reading the contents of a buffer in the semantics of the previous chapter. In particular, this interpretation stands for the common relaxation present in many relaxed memory models sometimes known as the capability of a thread to *read its own writes early* [Adve and Gharachorloo, 1996]. Notice that this kind of relaxation actually adds behaviors that are not possible by simply considering the reordering of normal actions (that is, if we disallow $\mathsf{rd}^{\circ}_{p,v}$ and $\mathsf{cas}^{\circ}_{p,v}$ actions, relying only on normal the $\mathsf{rd}_{p,v}$ and $\mathsf{cas}_{p,v}$ actions). To see this consider the following example (cf. Example 2.3 enhanced with barriers), where we assume a dependency relation $\mathcal{D}$ such that $(\mathsf{rd}_{p,v}\ \mathcal{D}\ \mathtt{rr})$ and $(\mathtt{rr}\ \mathcal{D}\ \mathsf{rd}_{p,v})$ as it is the case of the Sparc memory models (see Chapter 4).

Example 3.19.

$$\begin{bmatrix} p := tt\,; \\ r_0 := (\mathbf{!}p)\,; \\ \langle \mathtt{rd} | \mathtt{rd} \rangle\,; \\ r_1 := (\mathbf{!}q) \end{bmatrix} \quad \| \quad \begin{bmatrix} q := tt\,; \\ r_2 := (\mathbf{!}q)\,; \\ \langle \mathtt{rd} | \mathtt{rd} \rangle\,; \\ r_3 := (\mathbf{!}p) \end{bmatrix}$$

Assuming the initial store contains the value ff for both references p and q it might be surprising to learn that a possible final result for this program is $r_0 = r_2 = tt$ and $r_1 = r_3 = ff$. One should observe, however, that this is a possible outcome with the buffering techniques of Chapter 2. Notice that the first and second instruction of each thread are conflicting, and thus cannot be reordered, as it would violate a data dependency. Notice as well that the reads cannot be reordered either, since we have separated them by a $\langle \texttt{rd}|\texttt{rd}\rangle$ barrier for exactly that purpose. One can easily verify that if we do not consider the possibility of self-fulfilled reads ($\mathsf{rd}^{\circ}_{p,v}$ actions) this behavior is not possible. On the other hand, when considering them we notice that a action $\mathsf{rd}^{\circ}_{p,v}$ is not in conflict with the $\mathsf{wr}_{p,v}$ action that precedes it in the program order, and so they can be reordered in the speculation. This reordering opens the possibility to the above final result – where we notice that the barrier $\langle \texttt{rd}|\texttt{rd}\rangle$ does not prevent the reordering of the read of q and the write of p in the leftmost thread once the read of p has been permuted to the front of the speculation. However, the question now is: which are the values a $\mathsf{rd}^{\circ}_{p,v}$ action can possibly return? Indeed, this will be answered in the validity condition later, but let us anticipate that the only possible value a read of that kind can obtain is the latest write of the same reference performed by the same thread according to the order of the program text.

But then: is it always possible for a read to see its "own write early"? The answer to that question must necessarily be on the negative. Otherwise it would be impossible to write programs such that their speculative semantics coincides with the interleaving one – precisely the result we are seeking. To avoid this kind of "self-fulfilling" reads the language has to provide synchronization mechanisms – in particular λ-lock provides locks and λ-barrier provides barriers for that purpose. Indeed we have seen in the previous chapter that the unlock action required buffers to be empty before continuing, and a similar condition will be imposed here for λ-lock.

We can now give a precise definition for what we consider to be *valid* speculations.

Definition 3.20 (Valid Speculative Computations)**.** *For a dependency relation $\mathcal{D}$, a speculation σ is $\mathcal{D}$-valid if it is a $\mathcal{D}$-reordering of a normal computation σ' which satisfies the following condition: if $\sigma' = \sigma'_0 \cdot \xrightarrow[o]{a} \cdot \sigma'_1$ where $a \in \{\mathsf{rd}^{\circ}_{p,v}, \mathsf{cas}^{\circ}_{p,v} \mid v \in \mathcal{V}al\}$ then there exist δ_0 and δ_1 such that $\sigma'_0 = \delta_0 \cdot \xrightarrow[o']{a'} \cdot \delta_1$ with $a' \in \{\mathsf{wr}_{p,v}, \mathsf{cas}_{p,tt}, \mathsf{cas}^{\circ}_{p,tt}\}$ (where if $a' = \mathsf{cas}_{p,tt}$ or $a' = \mathsf{cas}^{\circ}_{p,tt}$ then $a = \mathsf{rd}^{\circ}_{p,tt}$ or $a = \mathsf{cas}^{\circ}_{p,ff}$), and such that if there is an action a'' in δ_1 satisfying $(a' \ \mathcal{D} \ a'')$ and $(a'' \ \mathcal{D} \ a)$ then $a'' = \mathsf{cas}^{\circ}_{p,ff}$. We say that the pair $[\delta_0, (a', o')]$ is the matching write of the pair $[\sigma'_0, (a, o)]$ and we denote it by $\mathsf{match}([\sigma'_0, (a, o)]) = [\delta_0, (a', o')]$.*

A speculative computation γ is $\mathcal{D}$-valid if all its projections $\gamma|_t$ are $\mathcal{D}$-valid speculations.

As a side note, we will actually consider the function match up to the step equivalence $\sim$ that we should define shortly.

It is only here that we add a constraint on the possible values returned by $\mathsf{rd}^{\circ}_{p,v}$ and $\mathsf{cas}^{\circ}_{p,v}$ actions. The requirement is that the value returned must be exactly the last value written to that reference by the same thread, where last is w.r.t. the program order. In particular, there must be one such write, otherwise

the speculation, and thus the entire speculative computation, cannot be qualified as valid. Moreover, we add constraint that there should be no synchronization action between the write and the speculated read; if there is a synchronization action in between them, we require the write action to be globally visible when the read is performed, and thus the read is necessarily from the memory, and not local, or "early". Through this definition we make formal our claim that $\mathsf{rd}^{\circ}_{p,v}$ actions were placed to allow a thread to read *its own writes early*.

It is clear for instance that the speculations given above that do not preserve the normal data dependencies are not *valid* according to this definition. (Notice that obviously a normal speculation that uses no $\mathsf{rd}^{\circ}_{p,v}$ actions is valid.) Then the reader can observe that from the thread system (from Example 1.4)– where we omit the thread identifiers

$$\left[\begin{array}{l} r := (!p)\,; \\ q := tt \end{array}\right] \;\|\; \left[\begin{array}{l} r' := (!q)\,; \\ p := tt \end{array}\right]$$

and an initial store S such that $S(p) = ff = S(q)$, we can, by a $\bowtie$-valid speculative computation, get as an outcome a state where the memory S' is such that $S'(r) = tt = S'(r')$, something that cannot be obtained with the standard, non-speculative interleaving semantics. This is a typical behavior of relaxed memory models where the reads can be reordered with respect to subsequent memory operations – a property symbolically called $\mathbf{R}{\rightarrow}\mathbf{RW}$, according to the terminology of [Adve and Gharachorloo, 1996]. We could not model this behavior by means of the semantics of Chapter 2. Indeed, we will see in the following chapter that for static thread systems the operational model for speculative computations is more general than the operational model for write-buffers of the previous chapter, in the sense that for any configuration, there are more outcomes following (valid) speculative computations than with write buffering. We also believe, although this would have to be more formally stated, that speculative computations are more general than most hardware memory models, which deal with memory accesses, but do not transform programs using semantical reasoning as optimizing compilers do. For instance, let us examine the case of the IRIW example (given in Example 2.24 and taken from [Adve and Boehm, 2010]), that is:

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

where $r_0 := (!p)$ stands for $(\mathsf{let}\ x = (!p)\ \mathsf{in}\ r_0 := x)$ and so on. If we start from a configuration where the memory S is such that $S(p) = ff = S(q)$, we may speculate in the third thread that $!\,q$ returns ff (which is indeed the initial value of q), and similarly in the fourth thread that $!\,p$ returns ff, and then proceed with the assignments $p := tt$ and $q := tt$, and continue to the end. Then we can reach, by a $\bowtie$-valid speculative computation, a state where the memory S' is such that $S'(r_0) = tt = S'(r_2)$ and $S'(r_1) = ff = S'(r_3)$, an outcome which cannot be obtained with the interleaving semantics.

Another unusual example, which is based on [Manson et al., 2005] where it is used to illustrate an "out of thin air" read, it is given by the following system made of two threads (with the obvious abbreviations).

Example 3.21.

$$\left[\begin{array}{l} p := \mathit{ff}\,; \\ (\mathsf{if}\ !p\ \mathsf{then}\ q := \mathit{tt}\ \mathsf{else}\ ()) \end{array}\right] \parallel \left[\begin{array}{l} q := \mathit{ff}\,; \\ (\mathsf{if}\ !q\ \mathsf{then}\ p := \mathit{tt}\ \mathsf{else}\ ()) \end{array}\right]$$

Then by a valid speculative computation we can reach, after having performed the two initial assignments, a state where $S(p) = \mathit{tt} = S(q)$. What is unusual with this example, with respect to what is generally expected from relaxed memory models for instance [Adve and Hill, 1990; Gharachorloo et al., 1990], is that this is, with respect to the interleaving semantics, a data race free thread system, which still has an "unwanted" outcome in the optimizing framework of speculative computations (see [Boehm and Adve, 2008] for a similar example). This indicates that we have to assume a stronger property than DRF (data-race freeness) to ensure that a program is "robust" with respect to speculations.

The speculative semantics of robust expressions coincides with their interleaving semantics. In other words the robust programs are the ones for which the speculative semantics is correct (with respect to the interleaving semantics).

Definition 3.22 (Robust Programs)**.** *A closed expression e is* $\mathcal{D}$-robust *iff for any t and γ such that $\gamma : (\emptyset, \emptyset, (t, e)) \xrightarrow{*} (S, L, T)$ is a* $\mathcal{D}$-valid *computation there exists a normal coherent computation $\overline{\gamma}$ such that $\overline{\gamma} : (\emptyset, \emptyset, (t, e)) \xrightarrow{*} (S, L, T)$.*

We shall give sufficient conditions for robustness in the following sections, but we first establish general results regarding the speculative semantics.

3.1.4 Properties of Speculations

First, we extend the notion of residual by defining $o/_e\sigma$ where o is an occurrence and σ a speculation. This is defined by induction on the length of σ, where the notation $o' \equiv o/_e\sigma$ means that $o/_e\sigma$ is defined in e and is o'.

$$o/_e\varepsilon = o$$
$$o/_e(e \xrightarrow[o']{a} e') \cdot \sigma = (o/_e(a, o'))/_{e'}\sigma$$

The following lemma states that the residual of a given occurrence along equivalent speculations are the same. This property which is depicted in Figure 3.6 was called the "Cube Lemma" in [Boudol, 1986] and was introduced in the "parallel moves lemma" of Lévy [1980].

Lemma 3.23 (The Cube Lemma)**.** *Let $\sigma = e_0 \xrightarrow[o_0]{a_0} e \xrightarrow[o_1]{a_1} e_1$ and $\sigma' = e_0 \xrightarrow[o'_1]{a_1} e' \xrightarrow[o'_0]{a_0} e_1$ and $\sigma \infty^{\mathcal{D}} \sigma'$, then for all o we have $o/_e\sigma \equiv o/_e\sigma'$.*

Proof. Straightforward (but tedious) case analysis. Refer to the proof of Lemma 3.15 for a similar proof.

□

The Figure 3.6 presents three cases of the lemma (one for each nondashed square) and we include as well the expression e' which exists as justified by lemma 2.15.

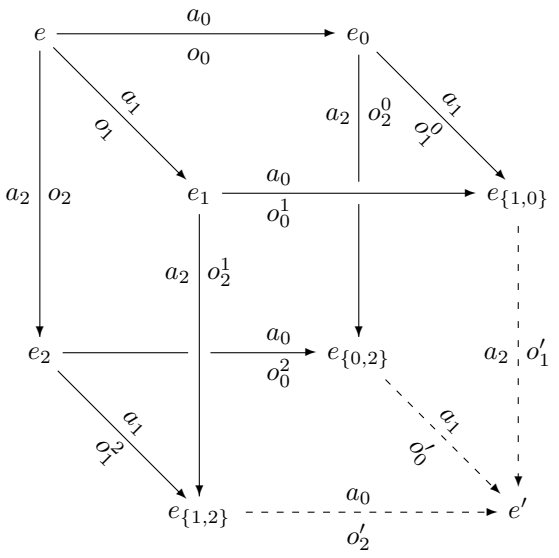


Figure 3.6: Diagram of the Cube Lemma 3.23: the occurrences o_i^j stand for the residual of o_i after the action a_j, that is $o_i^j \equiv o_i/_e(a_j, o_j)$. The occurrences o'_i are given by the statement of the lemma.

In the following we shall often omit the expressions in a speculation, writing $\sigma_0 \cdot \xrightarrow[o]{a} \cdot \sigma_1$ instead of $\sigma_0 \cdot (e_0 \xrightarrow[o]{a} e_1) \cdot \sigma_1$ and similarly for the residuals, written $o/(a, o')$ instead of $o/_e(a, o')$. Indeed, e_0 is determined by σ_0, and, given e_0, the expression e_1 is determined by the pair (a, o). Now we introduce the notion of a *step*, called "redex-with-history" in [Berry and Lévy, 1979; Lévy, 1980], and of steps being in the same *family*, a property introduced in [Berry and Lévy, 1979].

Definition 3.24 (Steps)**.** *A* step *is a pair* $[\sigma, (a, o)]$ *of a speculation* $\sigma : e \xrightarrow{*} e'$ *and an action* a *at occurrence* o *such that* $e' \xrightarrow[o]{a} e''$ *for some expression* e''*. Given a speculation* σ*, the set* $\mathsf{Step}(\sigma)$ *is the set of steps* $[\varsigma, (a, o)]$ *such that* $\varsigma \cdot \xrightarrow[o]{a} \leq \sigma$*. The binary relation* $\sim^{\mathcal{D}}$ *on steps, meaning that two steps are in the* same family*, is the equivalence relation generated by the rule*

$$\frac{\exists \sigma''.\ \sigma' \propto^{\mathcal{D}} \sigma \cdot \sigma'' \ \&\ o' \equiv o/\sigma''}{[\sigma, (a, o)] \sim^{\mathcal{D}} [\sigma', (a, o')]}$$

Speculations that are related by the reordering relation have similar steps:

Lemma 3.25. *If* $[\varsigma, (a, o)] \in \mathsf{Step}(\sigma)$ *and* $\sigma \propto^{\mathcal{D}} \sigma'$*, then there exists* $[\varsigma', (a, o')] \in \mathsf{Step}(\sigma')$ *with* $[\varsigma, (a, o)] \sim^{\mathcal{D}} [\varsigma', (a, o')]$

Proof. The proof is obvious by induction on the definition of $\propto^{\mathcal{D}}$. □

A property that should be intuitively clear is that if a step in a speculation is in the same family as the initial step of an equivalent speculation, then it can be commuted with all the steps that precede it:

Lemma 3.26. *Let* $\sigma = \varsigma_0 \cdot \xrightarrow[\bar{o}]{\bar{a}} \cdot \varsigma_1 \cdot \xrightarrow[o]{a} \cdot \varsigma_2$ *and* $\sigma \propto^{\mathcal{D}} \xrightarrow[o']{a} \cdot \sigma'$ *with* $[\varsigma_0 \cdot \xrightarrow[\bar{o}]{\bar{a}} \cdot \varsigma_1, (a, o)] \sim^{\mathcal{D}} [\varepsilon, (a, o')]$*, then there exists* o''*,* e''*,* $\bar{o'}$ *and* σ'' *such that* $\sigma \propto^{\mathcal{D}} \varsigma_0 \cdot (e \xrightarrow[o'']{a} e'' \xrightarrow[\bar{o'}]{\bar{a}} \bar{e}) \cdot \sigma''$ *where* $o \equiv o''/(\xrightarrow[\bar{o'}]{\bar{a}} \cdot \sigma'')$ *and* $\bar{o'} \equiv \bar{o}/(a, o'')$*.*

Proof. The proof is by induction on the inference of $\sigma \propto^{\mathcal{D}} \xrightarrow[o']{a} \cdot \sigma'$. Let us consider the case where $\sigma \propto^{\mathcal{D}} \xrightarrow[o']{a} \cdot \sigma'$. The case where $\sigma = \xrightarrow[o']{a} \cdot \sigma'$, meaning that $\sigma_0 = \varepsilon$, is trivial. Otherwise there exists a speculation $\widehat{\sigma}$ with $\widehat{\sigma} \propto^{\mathcal{D}} \xrightarrow[o']{a} \cdot \sigma'$ (with a shorter inference) and $\widehat{\sigma}$ results from σ by replacing two consecutive steps. If the transposition occurs in ς_2 or commutes the last step of $\sigma_0 \cdot \xrightarrow[o]{a}$ with the first step of ς_2 we apply the induction hypothesis. If the transposition commutes the last step of σ_0 with the step $[\sigma_0, (a, o)]$ we have the required conclusion for the case where $\varsigma_1 = \varepsilon$, and we have the conclusion by the induction hypothesis otherwise. Finally, if the transposition occurs within σ_0 we use the Cube Lemma 3.23 and the induction hypothesis to conclude.

The proof in the case where $\xrightarrow[o']{a} \cdot \sigma' \propto^{\mathcal{D}} \sigma$ is similar to the one just considered taking $\xrightarrow[\hat{o'}]{\hat{a}} \cdot \sigma' \propto^{\mathcal{D}} \widehat{\sigma}$ with $\widehat{\sigma}$ reaching σ by permuting only two steps. □

3.1.5 Properties of Speculative Computations

The properties of speculative computations that we develop here state that if actions of different threads are not conflicting, then reordering them is of no consequence in the overall computation. Moreover, if the actions are of the same thread, and they are not dependent, they can also be reordered reaching the same final configuration. We will present these results now.

From now on, we shall consider *regular* configurations, where at most one thread can hold a given lock, and where a lock held by some thread is indeed in the lock context. This is defined as follows:

Definition 3.27 (Regular Configuration)**.** *A configuration* $C = (S, L, T)$ *is* regular *if and only if it satisfies*

i) *if* $T = (t_i, \mathbf{\Sigma}_i[(e_i \backslash \ell)]) \| T_i$ *for* $i = 0, 1$ *then* $t_0 = t_1$ & $\mathbf{\Sigma}_0 = \mathbf{\Sigma}_1$ & $e_0 = e_1$ & $T_0 = T_1$
ii) $T = (t, \mathbf{\Sigma}[(e \backslash \ell)]) \| T' \Rightarrow \ell \in L$

For instance, any configuration of the form $(\emptyset, \emptyset, (t, e))$ where e is an expression is regular. The following should be obvious:

Remark 3.28. *If* C *is regular and* $C \xrightarrow[t,o]{a} C'$ *then* C' *is regular.*

The following lemma (for a different notion of computation) was called the "Asynchrony Lemma" in the previous chapter. Previously we used it as the basis to define the equivalence by permutation of computations. We could also introduce a similar relation here, generalizing the reordering of speculations, but this is actually not necessary.

Lemma 3.29. *Let C be a (well-formed) regular configuration. If $C \xrightarrow[t_0,o_0]{a_0} C_0 \xrightarrow[t_1,o_1]{a_1} C'$ with $t_0 \neq t_1$, $\neg(a_0 \# a_1)$ and $a_0 = \overset{\curvearrowleft}{\ell} \Rightarrow a_1 \neq \overset{\curvearrowright}{\ell}$ and $a_0 = \mathsf{spw}_{e'} \Rightarrow t_1 \in \mathsf{dom}(T)$ if $C = (S, L, T)$, then there exists C_1 such that $C \xrightarrow[t_1,o_1]{a_1} C_1 \xrightarrow[t_0,o_0]{a_0} C'$.*

Proof (Sketch). The proof is by a case analysis on the actions a_0 and a_1:

- If $a_0 \in \{\beta, \beta_v, \swarrow, \searrow, \mathsf{rd}^{\mathsf{o}}_{p,v}, \mathsf{cas}^{\mathsf{o}}_{p,ff}, \mathtt{ww}, \mathtt{wr}, \mathtt{rw}, \mathtt{rr}\}$ then this action has no side effect (i.e., it does not modify the components S, L and T of the configuration), and therefore such an action commutes with any other action a_1.
- If $a_0 = \nu_{p,v}$ then it cannot be the case that a_1 is $\mathsf{rd}_{p,w}$ or $\mathsf{wr}_{p,w}$, by the well-formedness of the configurations. Also, $a_1 \neq \nu_{p,w}$, and it is therefore easy to see that a_1 commutes with a_0 in this case.
- If $a_0 = \mathsf{rd}_{p,v}$ then we have $a_1 \notin \{\mathsf{wr}_{p,w}, \mathsf{cas}_{p,tt}, \mathsf{cas}^{\mathsf{o}}_{p,tt}\}$ (otherwise $a_0 \# a_1$) and $a_1 \neq \nu_{p,w}$ by well-formedness. Again it is easy to see that in any possible case for a_1, the two actions commute, producing the same resulting configuration.
- If $a_0 = \mathsf{cas}_{p,ff}$ then the case is identical to the previous one.
- If $a_0 = \mathsf{wr}_{p,v}$ then we have $a_1 \notin \{\mathsf{rd}_{p,w}, \mathsf{wr}_{p,w}, \mathsf{cas}_{p,w}, \mathsf{cas}^{\mathsf{o}}_{p,tt}\}$ and $a_1 \neq \nu_{p,w}$. As in the previous case, we easily conclude.
- If $a_0 = \mathsf{cas}_{p,tt}$ or $a_0 = \mathsf{cas}^{\mathsf{o}}_{p,tt}$ then the case is identical to the previous one.
- If $a_0 = \overset{\curvearrowleft}{\ell}$ or $a = \mu$ then $a_1 \neq \overset{\curvearrowright}{\ell}$ since two different threads cannot acquire the same lock (we are using the regularity of C when $a_0 = \mu$). Also, $a_1 \neq \overset{\curvearrowleft}{\ell}$ because this would mean that C_0 is not regular. If $a_0 = \overset{\curvearrowleft}{\ell}$ then $a_1 \neq \overset{\curvearrowleft}{\ell}$, since otherwise C_0 would not be regular, and $a_1 \neq \overset{\curvearrowright}{\ell}$ by hypothesis. Again in these cases it is easy to conclude that the lemma holds.
- The case where $a_0 = \mathsf{spw}_e$ is immediate.

□

And a similar lemma holds for transitions of the same thread:

Lemma 3.30. *Let C be a (well-formed) regular configuration. If $C \xrightarrow[t,o_0]{a_0} C_0 \xrightarrow[t,o'_1]{a_1} C'$ with $C = (S, L, (t, e) \| T)$, $C_0 = (S_0, L_0, (t, e_0) \| T_0)$, $C' = (S', L', (t, e') \| T')$ and $e \xrightarrow[o_0]{a_0} e_0 \xrightarrow[o'_1]{a_1} e' \propto^{\mathcal{D}} e \xrightarrow[o_1]{a_1} e_1 \xrightarrow[o'_o]{a_0} e'$ then $C \xrightarrow[t,o_1]{a_1} (S_1, L_1, (t, e_1) \| T_1) \xrightarrow[t,o'_0]{a_0} C'$ for some S_1, L_1 and T_1.*

Proof (Sketch). We distinguish three cases, according to the respective position of the occurrences o_0 and o_1.

- If $o_0 \leq o_1$, we can only have $a_0 \in \{\beta, \beta_v\}$, and therefore $S_0 = S$, $L_0 = L$ and $T_0 = T$, and it is easy to conclude with $S_1 = S'$, $L_1 = L'$ and $T_1 = T'$.
- If $o_1 < o_0$, then $a_1 \in \{\beta, \beta_v\}$, hence $S' = S_0$, $L' = L_0$ and $T' = T_0$, and we conclude, as in the previous case, with $S_1 = S$, $L_1 = L$ and $T_1 = T$.
- If o_0 and o_1 are disjoint, that is $o_0 \not\leq o_1$ and $o_1 \not\leq o_0$, we proceed by cases on (a_0, o_0) and (a_1, o_1), as in the previous proof. The hypothesis

$e \xrightarrow[o_0]{a_0} e_0 \xrightarrow[o'_1]{a_1} e' \propto^{\mathcal{D}} e \xrightarrow[o_1]{a_1} e_1 \xrightarrow[o'_o]{a_0} e'$ requires that $\neg(a_0 \mathcal{D} a_1)$ which in particular implies that $\neg(a_0 \# a_1)$. Notice also that $a_0 \notin \mathcal{S}ync$, because it would contradict this hypothesis.

□

3.2 Robustness for λ-lock

In this section we consider the problem of providing a property to guarantee the robustness for the high-level language with locks. Let us begin our discussion of this high-level language by defining the actual constraints of the dependency relation.

We define a dependency relation $\ltimes_L$ that contains the minimal set of dependencies that are enough to guarantee our robustness result for λ-lock. In fact, the only additional requirement is that actions that come before (in the program order) synchronization actions must not be reordered with respect to these. Notice the similarity with the unlock action of the previous chapter. Let us recall the definition of $\mathcal{S}ync$:

$$b \in \mathcal{S}ync \quad ::= \quad \mathsf{spw}_e \mid \overset{\frown}{\ell}$$

and let us give the formal definition for feasible dependency relations for λ-lock.

Definition 3.31 (Dependency Relation for λ-lock)**.** *An L-dependency relation $\mathcal{D}$ for the λ-lock language, is a binary relation on actions that satisfies $\ltimes_L \subseteq \mathcal{D}$ where:*

$$\ltimes_L \quad \triangleq \quad \bowtie \cup (\mathcal{A}ct \times \mathcal{S}ync)$$

Notice here, that many of the requirements of the definition 3.17 are trivially satisfied by λ-lock, since there is no compare-and-swap construct in it. Until the end of this section we will completely disregard compare-and-swap and barriers.

We have chosen a rather minimal set of dependencies for the $\ltimes_L$ relation. In particular the only additional constraint to those included in the dependency relation ($\bowtie$) is that every action that comes in the program order "before" a synchronization action has to happen before the synchronization action in the speculation as well. In that sense, performing the action of an unlock or a spawn depends on any action that precedes it in the program order. In relation with the previous chapter, we can also think that synchronization actions require the pending buffers to be empty, since they constraint the use of $\mathsf{rd}^{\circ}_{p,v}$ actions. One can make a parallel with a *flush* action in an architecture with buffers. This approach is also slightly different from the one taken in [Boudol and Petri, 2010] where instead of considering this dependency relation, these actions were only allowed to happen within evaluation contexts (rather than speculation contexts). In essence, this has the same effect as disallowing the execution of a thread creation and unlock actions if there are pending actions that should come before them in the program order.

The motivation for having such liberal definition of dependency is that it is sufficient for proving our safety results, while being very general. In particular, we are interested here in providing a framework to describe different speculative techniques, which we shall relate to relaxed memory models as well.

Thus we prefer to have the minimal set of restrictions that are enough to prove our results. Further restrictions only strengthen the results derived here. We will later comment on how variations of the dependency relation provide different, and perhaps more intuitive, semantics for locks corresponding to common programming practice.

With this instantiation of the dependency relation we have a sufficient validity definition to prove the main result in this section, that is that *speculatively* data-race free programs are robust.

3.2.1 Robustness Condition: SDRF

The following definition, which should be familiar from the previous chapter, is the lifting of the Data Race Free definition to the speculative calculus.

Definition 3.32 (Speculative DRF Program)**.** *A configuration C* has a speculative data race *iff there exist $t_i,\ o_i,\ a_i$ and C_i $(i = 0, 1)$ such that $C \xrightarrow[t_0,o_0]{a_0} C_0$ and $C \xrightarrow[t_1,o_1]{a_1} C_1$ with $t_0 \neq t_1$ & $a_0 \# a_1$. A $\mathcal{D}$-valid speculative computation $\left(C_i \xrightarrow[t_i,o_i]{a_i} C_{i+1}\right)_{0 \leqslant i \leqslant n}$ is* speculatively date race free *iff for all i, C_i has no speculative data race. A configuration C is* speculatively date race free *(speculatively DRF, or SDRF) w.r.t. the dependency relation $\mathcal{D}$ iff any $\mathcal{D}$-valid speculative computation originating in C is data race free. An expression e is speculatively DRF w.r.t. the dependency relation $\mathcal{D}$ iff for any t the configuration $(\emptyset, \emptyset, (t, e))$ is speculatively DRF w.r.t. the relation $\mathcal{D}$.*

It is obvious that this is a safety property, in the sense that if C is speculatively DRF and C' is reachable from C by a normal computation, then C' is speculatively DRF w.r.t. the $\ltimes_L$ dependency relation. We could have formulated this property directly, without resorting to the conflict relation, saying that there are no reachable concurrent accesses to the same location in the memory. In this way we could deal with optimizing architectures (such as the Alpha memory model, see [Compaq, 2002]) that allow to reorder such accesses, by including the case where these concurrent accesses can occur (in the speculative semantics) from within the same thread, like for instance in $p := \mathit{ff}\,; r := !p$. We do not follow this way here, since such a model requires unnatural synchronizations from the programmer.
Notice that if we have two consecutive conflicting steps from different threads in a computation, then the computation is not free of data races:

Remark 3.33. *Let $C \xrightarrow[t_0,o_0]{a_0} C_0 \xrightarrow[t_1,o_1]{a_1} C'$ with $t_0 \neq t_1$ and $a_0 \# a_1$, then C contains a data race. That is, there exists C'' such that $C \xrightarrow[t_1,o_1]{a_1} C''$.*

In order to establish our main result, we need some preliminary lemmas, regarding both speculations and speculative computations.

Provided with Lemma 3.26 we now show that in a speculation, unlock and spawning actions act as *barriers* with respect to other actions that occur in an evaluation context: these actions cannot be permuted with unlock (or spawn) actions. This is expressed by the following lemma:

Lemma 3.34. *If σ is a $\mathcal{D}$-valid speculation, justified by the normal execution $\widehat{\sigma}$, that is $\sigma \propto^{\mathcal{D}} \widehat{\sigma}$; where $\widehat{\sigma} = \widehat{\sigma_0} \cdot \xrightarrow[o]{a} \cdot \widehat{\sigma_1}$ with $a \in \mathcal{S}ync$, then $\sigma = \sigma_0 \cdot \xrightarrow[\bar{o}]{a} \cdot \sigma_1$ with $[\widehat{\sigma_0}, (a, o)] \sim^{\mathcal{D}} [\sigma_0, (a, \bar{o})]$, and for any step $[\widehat{\varsigma}, (a', o')] \in \mathsf{Step}(\widehat{\sigma_0})$ there is a step $[\varsigma, (a', o'')] \in \mathsf{Step}(\sigma_0)$ such that $[\widehat{\varsigma}, (a', o')] \sim^{\mathcal{D}} [\varsigma, (a', o'')]$.*

Proof. The proof is by induction on the inference of $\sigma \propto^{\mathcal{D}} \widehat{\sigma}$. Let us consider the case where $\widehat{\sigma}$ is obtained from σ by a single transposition; and let $\sigma = \sigma_0 \cdot (e \xrightarrow[o_1]{a_1} e_0 \xrightarrow[o_0]{a_0} e') \cdot \sigma_1$ with $\neg(a_0 \mathcal{D} a_1)$ and $\widehat{\sigma} = \sigma_0 \cdot (e \xrightarrow[o_0']{a_0} e_1 \xrightarrow[o_1']{a_1} e') \cdot \sigma_1$. Clearly in this case $a_1 \notin \mathcal{S}ync$, for $a_1 \in \mathcal{S}ync$ implies that $a_0 \ltimes_L a_1$ and then $a_0 \mathcal{D} a_1$. Then $a_1 \neq a$ and the conclusion is obvious.

Let us consider now the case where $\sigma \propto^{\mathcal{D}} \sigma' \propto^{\mathcal{D}} \widehat{\sigma}$ where σ' is obtained from σ by a single transposition and $\sigma' \propto^{\mathcal{D}} \widehat{\sigma}$ is a shorter inference. Suppose none of the steps being permuted is in the family of $[\widehat{\sigma_0}, (a, o)]$, then we can simply conclude by the induction hypothesis. Otherwise suppose that $\sigma = \sigma_0 \cdot (e \xrightarrow[o_1]{a_1} e_0 \xrightarrow[o_0]{a} e') \cdot \sigma_1$ and $\sigma' = \sigma_0 \cdot (e \xrightarrow[o_0']{a} e_1 \xrightarrow[o_1']{a_1} e') \cdot \sigma_1$ with $\neg(a_1 \mathcal{D} a)$. Then we conclude by the induction hypothesis, since the steps in $\mathsf{Step}(\sigma_0)$ remain unchanged. Finally, the case where $\sigma = \sigma_0 \cdot (e \xrightarrow[o_1]{a} e_0 \xrightarrow[o_0]{a_0} e') \cdot \sigma_1$ needs not be considered since we necessarily have that $a_0 \ltimes_L a$ for $a \in \mathcal{S}ync$; otherwise we would have a contradiction to the validity hypothesis. □

In order for a speculation to be valid, all the operations that normally (i.e. in the program order) precede a $\mathcal{S}ync$ action, and in particular an unlock, must be performed before this action in the speculation. In some sense, $\mathcal{S}ync$ actions considered here act as a barrier disallowing preceding actions from being delayed across them.

We now prove that given a speculatively data race free configuration C every two conflicting actions by different threads in any $\ltimes_L$-valid computation starting from C must be separated by an unlock action performed by the first thread.

Lemma 3.35. *Let C be a well-formed, closed regular configuration such that C is SDRF. If $\gamma : C \xrightarrow{*} C'$ is a $\ltimes_L$-valid speculative computation such that $\gamma = \gamma_0 \cdot \xrightarrow[t_0, o_0]{a_0} \cdot \gamma_1 \cdot \xrightarrow[t_1, o_1]{\mathsf{wr}_{p,v}} \cdot \gamma_2$ with $t_0 \neq t_1$ and $a_0 \# \mathsf{wr}_{p,v}$ we can conclude that $\gamma_1 = \gamma_1' \cdot \xrightarrow[t_0, o]{a} \cdot \gamma_1''$ for some γ_1', γ_1'', o and a where $a \in \mathcal{S}ync$.*

Proof. The proof proceeds by induction on the length of γ_1. The base case is trivial, since the hypothesis $a_0 \# \mathsf{wr}_{p,v}$ implies that we have a configuration with a data race, contradicting the hypothesis stating that C is SDRF. Let us concentrate on the inductive case now.

Let us assume by contradiction that there are no $\overset{\curvearrowleft}{\ell}$ or spw_e actions by thread t_0 in γ_1. Let us now consider the following cases for a_0:

- if $a_0 = \mathsf{wr}_{p,w}$ we have the following cases: If $\gamma_1|_t = \sigma_1 \cdot \xrightarrow[t_2, o_2]{a_2} \cdot \sigma_1'$ with $a_0 \# a_2$ then $a_2 \# \mathsf{wr}_{p,v}$ as well, and we can apply the induction hypothesis with a_2 in the place of a_0. In the case that there is no such conflicting event in $\gamma_1|t$ it must be the case that for all action a' appearing in $\gamma_1|_t$ it holds $\neg(a_0 \ltimes_L a')$. Assuming now that $\gamma_1 = \xrightarrow[t_3, o_3]{a_3} \cdot \widehat{\gamma_1}$ it must be that $t_0 \neq t_3$ and then $\neg(a_0 \# a_3)$, otherwise we would have a data race contradicting the

SDRF hypothesis, and in this case we conclude applying the lemma 3.29 and the induction hypothesis; otherwise if $t_0 = t_3$ we have $\neg(a_0 \ltimes_L a_3)$ and we conclude applying lemma 3.30 and the induction hypothesis. This contradicts the assumption that there is no $\overset{\curvearrowleft}{\ell}$ or spw_e action by t_0 in γ_1.

- if $a_0 = \mathsf{rd}_{p,w}$ then, as previously, all actions in $\gamma_1|_t$ are nonconflicting with $\mathsf{rd}_{p,w}$ or we conclude by the induction hypothesis using any conflicting action occurring in a later position. So again, if $\gamma_1 = \xrightarrow[t_3,o_3]{a_3} \cdot \widehat{\gamma_1}$ we know that if $t_0 \neq t_3$ then $\neg(a_0 \# a_3)$ and conclude as before. On the other hand $t_0 \neq t_3$ implies that $\neg(a_0 \ltimes_L a_3)$ and we conclude applying lemma 3.30 and the induction hypothesis.

□

Notice that this particular proof is not given in its most general form. This is so because we assume that if actions are not dependent according to the $\ltimes_L$ relation, then we can reorder them as provided by the lemma 3.30. This needs not be the case for dependency relations that extend the $\ltimes_L$ relation (i.e. any relation $\mathcal{D}$ such that $\ltimes_L \subset \mathcal{D}$). Indeed, the claim remains true if we take any dependency relation that extends this one, but the argument is more sophisticated, so we prefer to present this simpler lemma, and continue with our results for this particular dependency relation.

Proposition 3.36. *Let C be a well-formed, closed, regular configuration. If $\gamma : C \xrightarrow{*} C'$ is a $\ltimes_L$-valid speculatively data race free computation, then there exists a normal computation $\overline{\gamma}$ from C to C'.*

Proof. We proceed by induction on the length of γ. This is trivial if $\gamma = \varepsilon$. Otherwise, let $\gamma = \left(C_i \xrightarrow[t_i,o_i]{a_i} C_{i+1}\right)_{0 \leqslant i \leqslant n}$ with $n > 0$. Notice that for any i, the configuration C_i is well-formed, regular and has no data race. The set $\{\, t \mid \gamma|_t \neq \varepsilon \,\}$ is nonempty. For any t there exists a normal (up to $\mathsf{rd}^{\circ}_{p,v}$ actions) $\ltimes_L$-valid speculation σ^t such that $\gamma|_t \propto^{\ltimes_L} \sigma^t$. Let j be the first index ($0 \leqslant j < n$) such that $\gamma|_{t_j} = \sigma_0 \cdot \xrightarrow[o_j]{a_j} \cdot \sigma_1$ and $\sigma^{t_j} = \xrightarrow[o]{a_j} \cdot \sigma'$ with $[\varepsilon, (o, a_j)] \sim^{\ltimes_L} [\sigma_0, (a_j, o_j)]$. Now we proceed by induction on j. If $j = 0$ then $o = o_j \in \mathcal{O}cc^*$, and we use the induction hypothesis (on the length n) to conclude. Otherwise, we have $C_{j-1} \xrightarrow[t_{j-1},o_{j-1}]{a_{j-1}} C_j \xrightarrow[t_j,o_j]{a_j} C_{j+1}$. We distinguish two cases.

- If $t_{j-1} \neq t_j$ then we have $\neg(a_{j-1} \# a_j)$ by remark 3.33 and since γ is speculatively data-race free. We show that $i < j \;\Rightarrow\; a_i \notin \mathcal{S}ync$. Assume the contrary, that is $a_i \in \mathcal{S}ync$ for some $i < j$. Then $\gamma|_{t_i} = \varsigma_0 \cdot \xrightarrow[o_i]{a_i} \cdot \varsigma_1$, and by Lemma 3.25 we have $\sigma^{t_i} = \overline{\varsigma_0} \cdot \xrightarrow[o']{a_i} \cdot \overline{\varsigma_1}$ with $[\varsigma_0, (o_i, a_i)] \sim [\overline{\varsigma_0}, (o', a_i)]$. Then by Corollary 3.34 the first step of $\overline{\varsigma_0} \cdot \xrightarrow[o']{a_i}$ is in the family of a step in $\varsigma_0 \cdot \xrightarrow[o_i]{a_i}$, contradicting the minimality of j. We therefore have $a_{j-1} \neq \overset{\curvearrowleft}{\ell}$ in particular. By Lemma 3.29 we can commute the two steps $\xrightarrow[o_{j-1}]{a_{j-1}}$ and $\xrightarrow[o_j]{a_j}$, and we conclude using the induction hypothesis (on j).
- If $t_{j-1} = t_j$, we have $\sigma_0 = \varsigma_0 \cdot \xrightarrow[o_{j-1}]{a_{j-1}}$, and by Lemma 3.26 there exist o', o'' and σ'_1 such that $\gamma|_{t_j} \propto^{\ltimes_L} \varsigma_0 \cdot \xrightarrow[o']{a_j} \cdot \xrightarrow[o'']{a_{j-1}} \cdot \sigma'_1$ (where we recall that $\propto^{\ltimes_L}$ is

the symmetric closure of the $\propto^{\ltimes_L}$ relation) with $o \equiv o'/(a_{j-1}, o_{j-1})$. We conclude using Lemma 3.30 and the induction hypothesis (on j).

□

We have proved a property that is actually more precise than stated in the proposition, since the $\bar{\gamma}$ that is constructed is a reordering of γ – but we decided not to introduce explicitly this notion as regards speculative computations.

A final remark we need to establish our robustness result is that any speculative read (i.e. an action of the form $\mathsf{rd}^{\circ}_{p,v}$) sees exactly the value currently in the store in the computation $\bar{\gamma}$ as provided by the Proposition above.

Remark 3.37. *With the hypothesis of the Proposition 3.36, if $\bar{\gamma}$ is the resulting normal computation and $\bar{\gamma} = \gamma' \cdot (C \xrightarrow[t_0,o_0]{\mathsf{rd}^{\circ}_{p,v}} C') \cdot \gamma'$ where $C = (S, L, T)$, then $S(p) = v$.*

Proof. The proof considers the last write to p previous to the read ($\mathsf{rd}^{\circ}_{p,v}$). Suppose that $\bar{\gamma} = \gamma_0 \cdot \xrightarrow[t_0,o_0]{\mathsf{wr}_{p,v}} \cdot \gamma_1 \cdot \xrightarrow[t_1,o_1]{\mathsf{wr}_{p,w}} \cdot \gamma_2 \cdot (C \xrightarrow[t_0,o_0]{\mathsf{rd}^{\circ}_{p,v}} C') \cdot \gamma'$ and $t_0 \neq t_1$ and $(\gamma_1 \cdot \xrightarrow[t_1,o_1]{\mathsf{wr}_{p,w}} \cdot \gamma_2)|_t$ contains no $\mathsf{wr}_{p,v}$ action, meaning that according to the definition of speculation validity the write $\mathsf{wr}_{p,v}$ highlighted in the computation is the one satisfying the read $\mathsf{rd}^{\circ}_{p,v}$ in thread t_0. We have by lemma 3.35 that there must be an unlock or a thread spawn in $(\gamma_1 \cdot \xrightarrow[t_1,o_1]{\mathsf{wr}_{p,w}} \cdot \gamma_2)|_t$, which by the validity condition contradicts the fact that there is a $\mathsf{rd}^{\circ}_{p,v}$ action later.

□

An immediate consequence of the property 3.36 and the remark above (3.37) is the announced robustness result:

Theorem 3.38 (Robustness)**.** *Any speculatively data race free closed expression is $\ltimes_L$-robust*

We observe that if an expression is purely sequential, that is, it does not spawn any thread, then it is speculatively data race free, and therefore robust, that is, all the valid speculations for it are correct with respect to its standard semantics.

Let us conclude the treatment of the λ-lock language by reconsidering the synchronization mechanisms, and more precisely the relaxations allowed in it. We have mentioned that speculating bypassing synchronization can be surprising for the programmer. To see why, consider the following example, a lock proctected variation of Example 1.6, assuming that initially $flag = tt$ and where we assume a simple while construct.

Example 3.39.

$$\left[\begin{array}{l} p := tt\,; \\ (\mathsf{with}\ \ell\ \mathsf{do}\ flag := ff) \end{array}\right] \quad \| \quad \left[\begin{array}{l} \mathsf{while}\ (\mathsf{with}\ \ell\ \mathsf{do}\ (!flag))\ \mathsf{do}\ ()\,; \\ r_0 := (!p) \end{array}\right]$$

It is not hard to see that this example is free of data races in the interleaving semantics, since there is no data race on reference p, and the reference $flag$ is protected by the lock ℓ. However, when we consider $\ltimes_L$-valid speculative computations we have a data race in p, since we can speculate in the second

thread on the redex ($!p$) at any time. Importantly, this means that this program, commonly known as the safe publication idiom, is not safe under the relaxations we consider in this chapter. We could simply avoid this kind of speculations by adding the following condition on $\ltimes_L$:

$$\bigcup_{\ell \in \mathcal{L}ocks} \{\widehat{\ell}\} \times \mathcal{A}ct \subseteq \ltimes_L$$

in which case we get the standard *roach motel* semantics for locks. However, since our results hold for the more general case where this constraint is not added, we do not include them a priori. It is easy to see that this additional constraint does not invalidate our previous theorem, and furthermore, the same proofs are still valid under this extra assumption.

3.3 Robustness for λ-barrier

Part of the motivation for the formalization of speculative computations in this chapter is to describe, in an operational way, the semantics of relaxed memory models. However, λ-lock stands at a higher level than most of the languages that we find in low-level architectures. Indeed, the lock construct of the previous section is not common in instruction sets of a machine architectures[5]. On the other hand, one would typically find some kind of barrier instruction, whose effect is to prevent the reordering of instructions, or to guarantee the termination of instructions previous to the barrier before issuing the ones following it. But barriers alone are not enough to provide atomicity of memory accesses, needed to implement higher level synchronization mechanisms such as locks. Hence these architectures must provide some kind of atomicity construct, which they achieve by instructions like *compare-and-swap* or variants of it. Let us concentrate in this section on the language λ-barrier that we regard as a prototypical low-level language.

A natural question to ask here is: How do the standard DRF, or the previously discussed SDRF results apply in this case? Indeed, in the absence of locks, or other mutual exclusion mechanisms, it is unclear that the DRF guarantee can be applied. What is more, often times avoiding data races entirely is not possible at this low level. In particular, for implementing mutual exclusion algorithms – or any other synchronization mechanisms for that matter – with such a language one might have to resort to data races. So a question that follows is: How can one be sure that the implementation of a higher level synchronization mechanism in this architecture is sound (for the proper definition of soundness according to the mechanism)? And finally: can we identify properties that guarantee that programs written in this language are sequentially consistent? The first and last questions are highly related and we will provide an answer for them here. In particular, we will identify a property similar to the one identified in [Shasha and Snir, 1988], that guarantees that the behavior of speculative computations coincide with the behavior of sequentally consistent computations for programs satisfying it. We shall call this property *POSMA* (Preserving the Order of Shared Memory Accesses). To answer the

[5]Some architectures like x86 support locks, of a different kind, to provide atomicity for single instructions.

second question one would then reason about the POSMA property. We are currently conducting research in that direction.

This choice of the language constructs of λ-barrier is partly inspired by the Sparc memory models [SPARC, 1994] that we will consider in more detail in the following chapter.

Despite being at odds with much of the relaxed memory models literature [Ledgard, 1983; Adve and Boehm, 2010], we consider that providing guarantees for programs containing data races (that is, other than the DRF guarantee which does not consider them at all) is a requirement for low-level languages. We think this as a necessary step towards an end-to-end view of relaxed memory models [Gao and Sarkar, 1997], in which the guarantees provided at a higher programming language level are provably reflected by the intermediate stages of the compilation down to the execution of the program in the actual marchine architecture. The lack of such guarantees in lower level languages, or intermediate level compilations, requires an act of faith towards the implementation of these mechanisms. Actually, several recent works have pointed out that the current state of affairs as regards the specification of memory models is somewhat unsatisfactory [Sewell et al., 2010; Adve and Boehm, 2010; Ševčík and Aspinall, 2008] and therefore new, or perhaps more formal approaches need to be considered. We regard our formalization as a step towards more reliable guarantees for parallel programs running on machines with relaxed memory models.

Let us now characterize which are the speculative computations of this language that we shall consider as valid.

3.3.1 Valid Speculations

The most significant change, apart from the syntax, from the semantics of λ-lock is the definition of the dependency relation. Before showing the dependency relation let us introduce some abbreviations that will simplify our definitions. Let us define the set $\mathcal{R}d_p$ of actions on reference p with read semantics (Notice that this definition is different from $\mathcal{MR}dp$ in that $\mathsf{rd}^{\circ}_{p,v}$ actions are included, since they are reads):

$$\mathcal{R}d_p \triangleq \{\mathsf{rd}_{p,v}, \mathsf{rd}^{\circ}_{p,v}, \mathsf{cas}_{p,v}, \mathsf{cas}^{\circ}_{p,v} \mid v \in \mathcal{V}al\}$$

and a similar definition for the actions with write semantics on p:

$$\mathcal{W}r_p \triangleq \{\mathsf{wr}_{p,v}, \mathsf{cas}_{p,v}, \mathsf{cas}^{\circ}_{p,v} \mid v \in \mathcal{V}al\}$$

and their obvious generalizations to any reference:

$$\mathcal{R}d \triangleq \bigcup_{p \in \mathcal{R}ef} \mathcal{R}d_p \qquad \text{and} \qquad \mathcal{W}r \triangleq \bigcup_{p \in \mathcal{R}ef} \mathcal{W}r_p$$

Notice here that we consider $\mathsf{cas}_{p,v}$ actions to have write semantics whether they succeed or not. This is because we wish to disallow reordering actions that potentially depend on such actions. In particular this is in agreement with the semantics of the Sparc memory models which we will consider in the next chapter.

We can now define dependency relations for this language:

Definition 3.40 (Dependency Relation for λ-barrier)**.** *A B-dependency relation $\mathcal{D}$ for the λ-barrier language, is a binary relation on actions that satisfies $\ltimes_B \subseteq \mathcal{D}$ where:*

$$\begin{array}{rcl} \ltimes_B & \triangleq & \bowtie \cup\ \mathcal{A}ct \times \{\mathsf{spw}_e\} \\ & \cup & \bigcup_{p\in\mathcal{R}ef, v\in\mathcal{V}al} (\{(\mathsf{spw}_e\} \times \{\mathsf{rd}^{\mathsf{o}}_{p,v}, \mathsf{cas}^{\mathsf{o}}_{p,v}\}) \end{array}$$

We can observe that once more, we consider the action of spawning a new thread as imposing synchronization.

The reader might find surprising not to see barrier actions in the $\ltimes_B$ relation, after all these are the only way in which reorderings can be avoided (other than spawns) in this section. Actually, this is because we use the $\ltimes_B$ relation to prove our results, which are based on the $\ltimes_B$-POSMA property, and therefore assume that actions involved in data races happen as in the program order. Our theorems and intermediate results vacuously hold for computations not satisfying $\ltimes_B$-POSMA. However, we acknowledge that the task of writting $\ltimes_B$-POSMA programs is almost impossible (unless there is no communication whatsoever) if the expected dependencies of barriers are not included. So maybe a more reasonable dependency relation should be the $\ltimes'_B$ relation given below.

$$\begin{array}{rcl} \ltimes'_B & \triangleq & \ltimes_B \ \cup\ (\mathcal{R}d \times \{\mathtt{rr}, \mathtt{rw}\})\ \cup\ (\{\mathtt{rr}, \mathtt{wr}\} \times \mathcal{R}d) \\ & & \cup\ (\mathcal{W}r \times \{\mathtt{ww}, \mathtt{wr}\})\ \cup\ (\{\mathtt{ww}, \mathtt{rw}\} \times \mathcal{W}r) \end{array}$$

Importantly our results still hold for the $\ltimes'_B$ strengthening of $\ltimes_B$ in a trivial way.

Provided with the $\ltimes'_B$ dependency relation we notice that a trivial way to make speculations coherent – that means that all reads and compare-and-swap see the current value in the memory, as opposed to $\mathsf{rd}^{\mathsf{o}}_{p,v}$ and $\mathsf{cas}^{\mathsf{o}}_{p,v}$ actions – and moreover, to entirely disallow the use of $\mathsf{rd}^{\mathsf{o}}_{p,v}$ and $\mathsf{cas}^{\mathsf{o}}_{p,v}$ actions, is to require that writes and reads on the same reference be always separated by a $\langle \mathtt{wr}|\mathtt{rd}\rangle$ barrier in every thread.

A simple observation that we will use later is that there is a single normal speculation for any $\mathcal{D}$-valid speculation.

Remark 3.41. *If $\sigma_0 \propto^{\mathcal{D}} \sigma_1$ where σ_0 and σ_1 are both normal, then $\sigma_0 = \sigma_1$.*

If $\sigma = (e_i \xrightarrow[o_i]{a_i} e_{i+1})_{0\leqslant i<n}$ and $\sigma' = (e_i \xrightarrow[o_i]{a'_i} e_{i+1})_{0\leqslant i<n}$ with $\sigma \propto^{\mathcal{D}} \sigma'$, then there exists a permutation h of $\{0,\ldots,n-1\}$ such that $a_i = a'_{h(i)}$. We shall use explicitly this permutation, introducing the notation:

$$h : \sigma \propto^{\mathcal{D}} \sigma'$$

Notice that, with the previous notations, if $h(i) < h(j)$ and $a'_{h(i)}\,\mathcal{D}\,a'_{h(j)}$ then $i < j$ (this is easy to check, by induction on the definition of $\propto^{\mathcal{D}}$).

3.3.2 Robustness Condition: POSMA

Our robustness condition for λ-barrier states that if a thread engages in more than one conflict with another (one or many) threads, then the events that give rise to these conflicts should happen as prescribed by their order in the program:

Definition 3.42 (POSMA)**.** *A configuration C is* preserving the order of shared memory accesses *(POSMA) w.r.t. $\mathcal{D}$ ($\mathcal{D}-$posma for short), iff for any $\mathcal{D}$-valid speculative computation $\gamma : (C \xrightarrow{*} C')$ such that*

$$\gamma = \gamma_0 \cdot \xrightarrow[t,o]{a} \cdot \xrightarrow[t_0,o_0]{a_0} \cdot \gamma_1 \cdot \xrightarrow[t_1,o_1]{a_1} \cdot \xrightarrow[t,o']{a'} \cdot \gamma_2$$

(where $\xrightarrow[t_0,o_0]{a_0} \cdot \gamma_1 \cdot \xrightarrow[t_1,o_1]{a_1}$ is possibly of length one) with $t_0 \neq t \neq t_1$, $\gamma_1|_t = \varepsilon$, $a \neq a'$ and $a \# a_0$ and $a' \# a_1$, and if $\gamma_0|_t$ is of length j (i.e. a and a' are j^{th} and $j+1^{th}$ actions in $\gamma|_t$) and $h : \gamma|_t \propto^{\mathcal{D}} \sigma$ where σ is normal, then $h(j) < h(j+1)$.

Moreover, with the hypotheses above, whenever $\sigma = \sigma_0 \cdot \xrightarrow[\bar{o}]{a'} \cdot \sigma_1 \cdot \xrightarrow[\bar{o'}]{a} \cdot \sigma_2$ with $a \in \{\mathsf{rd}^{\mathsf{o}}_{p,v}, \mathsf{cas}^{\mathsf{o}}_{p,v}\}$ and $\mathsf{match}([\sigma_0 \cdot \xrightarrow[\bar{o}]{a'} \cdot \sigma_1, (a, \bar{o'})]) = [\sigma_0, (a', \bar{o})]$ and the length of σ_0 is n and the length of $\sigma_0 \cdot \xrightarrow[\bar{o}]{a'} \cdot \sigma_1$ is m then $n \leqslant h(j) \Rightarrow h(j+1) \not\leqslant m$.

The POSMA property is similar to the acyclicity condition discovered by Shasha and Snir in their pioneering work [Shasha and Snir, 1988]. However, their condition deals with graphs of memory accesses, whereas ours directly relies on the operational semantics of the programming language (our condition would be exactly the same with recursive programs).

Some examples might clarify the POSMA property. In the following program (the same as in the example 1.3) we consider a program that does not satisfy the POSMA property when assuming a dependency relation where every kind of memory access (reads or writes) on different references can be reordered, and we adopt the usual assumption about the initial memory, i.e. $p = q = ff$.

$$\begin{bmatrix} p := tt\,; \\ q := tt \end{bmatrix} \quad \| \quad \begin{bmatrix} r_0 := (!q)\,; \\ r_1 := (!p) \end{bmatrix}$$

A simple computation that proves that the POSMA property is not satisfied by this example is the following one, where we omit the occurrences, the configurations and we assume that the thread on the left is t_0 and the one on the right is t_1:

$$\gamma = \xrightarrow[t_0]{\mathsf{wr}_{q,tt}} \cdot \xrightarrow[t_1]{\mathsf{rd}_{q,tt}} \cdot \xrightarrow[t_1]{\mathsf{rd}_{p,ff}} \cdot \xrightarrow[t_0]{\mathsf{wr}_{p,tt}}$$

it is obvious that this computation is not sequentially consistent, but moreover, it does not respect the POSMA property, since the writes of thread t_0 are not in their program order, and are involved in data races. In fact, each thread is involved in two data races in this computation, but only t_0 can instantiate the thread t required by the POSMA property.

It is not hard to see that adding the appropriate barriers to the example we can make it satisfy the $\ltimes_{B'}$-POSMA property, and it is not hard to see as well that then we can only have sequentially consistent computations. The final program would be:

$$\begin{bmatrix} p := tt\,; \\ \langle \mathtt{wr} | \mathtt{wr} \rangle\,; \\ q := tt \end{bmatrix} \quad \| \quad \begin{bmatrix} r_0 := (!q)\,; \\ \langle \mathtt{rd} | \mathtt{rd} \rangle\,; \\ r_1 := (!p) \end{bmatrix}$$

Moreover, the example above illustrates a second interesting point about the POSMA property, and it is that we need to have at least two data races to deviate from sequentially consistent computations. But this fact is easier to illustrate with a simpler example.

$$\left[p := tt\,;\right] \quad \| \quad \begin{bmatrix}(!p)\,;\\ (!p)\end{bmatrix}$$

Here the only possible nonsequentially consistent computation is the one in which the reads of the second thread are reordered:

$$\gamma = \xrightarrow[t_1,((\lambda\_.[]\_)\cdot[])]{\mathsf{rd}_{p,ff}} \cdot \xrightarrow[t_0,[]]{\mathsf{wr}_{p,tt}} \cdot \xrightarrow[t_1,(\_.[])\cdot[]]{\mathsf{rd}_{p,tt}}$$

it is obvious in this case that if we had only one read in thread t_1 there is no possible nonsequentially consistent behavior.

To conclude with the examples we remark that compare-and-swap commands are not a substitute for barriers as can be seen in the following example, where we make the standard assumptions regarding the initial state of memory locations p and q:

$$\begin{bmatrix}(\mathtt{cas}\ p)\,;\\ (\mathtt{cas}\ q)\end{bmatrix} \quad \| \quad \begin{bmatrix}r_0 := (!q)\,;\\ \langle \mathtt{rd}|\mathtt{rd}\rangle\,;\\ r_1 := (!p)\end{bmatrix}$$

there is nothing in this example that forbids $r_0 = tt$ and $r_1 = ff$ as a final result, since the actions generated by compare-and-swap instructions can be reordered as in the first example above. A correct (that is sequentially consistent) implementation of this example requires a barrier between the compare-and-swap expressions.

$$\begin{bmatrix}(\mathtt{cas}\ p)\,;\\ \langle \mathtt{wr}|\mathtt{wr}\rangle\,;\\ (\mathtt{cas}\ q)\end{bmatrix} \quad \| \quad \begin{bmatrix}r_0 := (!q)\,;\\ \langle \mathtt{rd}|\mathtt{rd}\rangle\,;\\ r_1 := (!p)\end{bmatrix}$$

We notice that this kind of behavior is possible in architectures like Sparc [SPARC, 1994].

Let us move on now with the technical development to prove that POSMA is a sufficient condition on programs to guarantee robust computations. Let us prove the following lemma stating that if in a computation we have that a thread performs two consecutive *equal* actions, then these actions can be reordered in the computation:

Lemma 3.43. *Let $\gamma = (C \xrightarrow[t,o]{a} \cdot\ \delta\ \cdot \xrightarrow[t,o']{a} C')$ be a speculative computation such that $\delta|_t = \varepsilon$, $C = (S, (t,e)\|T)$ and $\xrightarrow[o]{a} \cdot \xrightarrow[o']{a} \propto^{\mathcal{D}} \xrightarrow[\bar{o'}]{a} \cdot \xrightarrow[\bar{o}]{a}$ with $o' \equiv \bar{o'}/_e(a,o)$ and $\bar{o} \equiv o/_e(a,\bar{o'})$. Then there exists δ' such that $\delta'|_{t'} = \delta|_{t'}$ for any t', and $\gamma = (C \xrightarrow[t,\bar{o'}]{a} \cdot\ \delta' \cdot \xrightarrow[t,\bar{o}]{a} C')$ is a speculative computation.*

Proof. Let $\gamma = (C_i \xrightarrow[t_i,o_i]{a_i} C_{i+1})_{0 \leqslant i \leqslant n}$, where $n \geqslant 1$, $t_0 = t_n = t$, $a_0 = a_n = a$, $o_0 = o$, $o_n = o'$ and $C_0 = C$, $C_{n+1} = C'$. If $C_i = (S_i, T_i, P_i)$ where $T_i = (t, e_i)\|T_i'$, we have $e_i = e_1$ for $1 \leqslant i \leqslant n$, and there exists e_1' such that $e_0 \xrightarrow[\bar{o'}]{a} e_1' \xrightarrow[\bar{o}]{a} e_{n+1}$. For $1 \leqslant i < n$ we let $C_i' = (S_i, (t, e_1')\|T_i', P_i)$. Clearly

$C'_i \xrightarrow[t_i,o_i]{a_i} C'_{i+1}$ for $1 \leqslant i < n$, and therefore we just have to check that $C_0 \xrightarrow[t,\bar{o'}]{a} C'_1$ and $C'_n \xrightarrow[t,\bar{o}]{a} C_{n+1}$, which is trivial since these transitions are determined by the action a (notice that a cannot be $\nu_{p,v}$, nor $\mathsf{spw}_{t',e}$, since a given pointer or thread name cannot be fresh twice). □

Now we prove that the $\mathcal{D}$-posma property is indeed a correct criterion for sequential consistency for speculative computations. A crucial property of $\mathcal{D}$-posma configurations is the following:

Lemma 3.44. *Let C be a configuration that is $\mathcal{D}$-posma, and let $\gamma : C \xrightarrow{*} C'$ be a $\mathcal{D}$-valid speculative computation such that $\gamma = \gamma_0 \cdot \xrightarrow[t,o_0]{a_0} \cdot \gamma_1 \cdot \xrightarrow[t,o_1]{a_1} \cdot \gamma_2$ with $a_0 \neq a_1$, $\gamma_1|_t = \varepsilon$ and $h : \gamma|_t \propto^{\mathcal{D}} \sigma$ where σ is normal. If the length of $\gamma_0|_t$ is j and $h(j) > h(j+1)$ then $\gamma_1 = \delta_0 \cdot \delta_1$ for some δ_0 and δ_1 such that $\neg(a_0 \# \delta_0)$ and $\neg(a_1 \# \delta_1)$.*

Proof. if $\neg(a_0 \# \gamma_1)$ we are done (with $\delta_0 = \gamma_1$ and $\delta_1 = \varepsilon$), so let us assume that $\gamma_1 = \delta_0 \cdot \xrightarrow[t',o]{a} \cdot \gamma'$ with $a_0 \# a$ and $\neg(a_0 \# \delta_0)$, where $t' \neq t$ since $\gamma_1|_t = \varepsilon$. Now we show that $a_1 \# \xrightarrow[t',o]{a} \cdot \gamma'$ is not possible. Assume the contrary, that is $\xrightarrow[t',o]{a} \cdot \gamma' = \gamma'' \cdot \xrightarrow[t'',o']{a'} \cdot \delta_1$ with $a' \# a_1$ and $\neg(a_1 \# \delta_1)$. By Lemma 3.29 we can commute a_0 with δ_0 and a_1 with δ_1 (if a thread is spawned with some name in δ_1 then this name is distinct from t since t was known before δ_1), obtaining a $\mathcal{D}$-valid computation

$$\delta = \gamma_0 \ \cdot \ \delta'_0 \cdot \xrightarrow[t,o_0]{a_0} \cdot \xrightarrow[t',o]{a} \cdot \gamma'' \cdot \xrightarrow[t'',o']{a'} \cdot \xrightarrow[t,o_1]{a_1} \cdot \delta'_1 \ \cdot \ \gamma_2$$

with $\delta|_t = \gamma|_t$. Then, since C is $\mathcal{D}$-posma, and $\delta'_0|_t = \varepsilon = \gamma''|_t = \delta'_1|_t$, we should have $h(j) > h(j+1)$, a contradiction. □

Let us prove an intermediate result stating that a $\mathcal{D}$-posma speculative computation necessarily has a normal speculative computation that corresponds to it. This will later be used to show that $\mathcal{D}$-posma speculative computations have a corresponding normal coherent computation, and thus are sequentially consistent.

Proposition 3.45. *Let C be a well-formed, closed configuration. If C is $\mathcal{D}$-posma and $\gamma : C \xrightarrow{*} C'$ is a $\mathcal{D}$-valid speculative computation, then there exists a normal computation $\overline{\gamma}$ from C to C'.*

Proof. For any t there exists a normal speculation ς^t such that $\gamma|_t \propto^{\mathcal{D}} \varsigma^t$. We proceed by induction on the total number of transpositions used to transform the speculations $\gamma|_t$ into ς^t. If this number is 0, γ is a normal computation. Otherwise, let

$$\gamma = \gamma_0 \cdot \xrightarrow[t,o]{a} \cdot \gamma_1 \cdot \xrightarrow[t,o']{a'} \cdot \gamma_2$$

where $\gamma_1|_t = \varepsilon$ and $h : \gamma|_t \propto^{\mathcal{D}} \varsigma^t$, with $h(j) > h(j+1)$ where j is the length of $\gamma_0|_t$. Notice that $\neg(a' \mathcal{D} a)$. There are two cases.

- If $a = a'$, we conclude using Lemma 3.43, and the induction hypothesis.

- Otherwise ($a \neq a'$), since C is $\mathcal{D}$-posma, by Lemma 3.44 there exist δ_0 and δ_1 such that $\gamma_1 = \delta_0 \cdot \delta_1$, with $\neg(a \# \delta_0)$ and $\neg(a' \# \delta_1)$. If $a = \mathsf{spw}_{t',e}$ we have $\neg(a' \# \gamma_1)$ since $\neg(a' \mathcal{D} a)$, and we may let $\delta_0 = \varepsilon$ and $\delta_1 = \gamma_1$ in this case. Then, possibly using repeatedly Lemma 3.29, we can build a speculative computation of the form

$$\gamma_0 \cdot \delta_0' \cdot \xrightarrow[t,o]{a} \cdot \xrightarrow[t,o']{a'} \cdot \delta_1' \cdot \gamma_2$$

and by Lemma 3.30 we get a computation of the form

$$\gamma_0 \cdot \delta_0' \cdot \xrightarrow[t,\bar{o'}]{a'} \cdot \xrightarrow[t,\bar{o}]{a} \cdot \delta_1' \cdot \gamma_2$$

We conclude using the induction hypothesis.

□

The following lemmas are auxiliary results that are required to prove that a normal computation starting from a $\mathcal{D}$-posma configuration has a corresponding normal speculative computation.

Lemma 3.46. *Given a speculative computation $\gamma : C \xrightarrow{*} C'$ such that $\gamma = \gamma|_t \cdot \xrightarrow[t',o]{a}$ with $t \neq t'$ and $\neg(\gamma|_t \# a)$ then there is a speculative computation $\gamma' : C \xrightarrow{*} C'$ such that $\gamma' = \xrightarrow[t',o]{a} \cdot \gamma|_t$.*

Proof. This is actually a trivial consequence of Lemma 3.29. □

Let us now extend the definition of conflict to an action and a speculation in the obvious way, that is:

$$\gamma \# a \iff \exists\, \gamma_0, \gamma_1, a', t, o \,.\, \gamma = \gamma_0 \cdot \xrightarrow[t,o]{a'} \cdot \gamma_1 \;\&\; a \# a'$$

We can now state the following lemma.

Lemma 3.47. *Given a speculative computation $\gamma : C \xrightarrow{*} C'$ and a thread t such that for all γ_0, γ_1, a, o and $t' \neq t$ we have $\gamma = \gamma_0 \cdot \xrightarrow[t,o]{a} \cdot \gamma_1 \Rightarrow \neg(\gamma_0|_{t'} \# a)$. Then there exists a speculative computation $\gamma' : C \xrightarrow{*} C'$ such that $\gamma' = \gamma'|_t \cdot \gamma''$ for some γ'' such that $\gamma''|_t = \varepsilon$ and $\gamma''|_{t''} = \gamma_{t''}$ for all thread $t'' \neq t$.*

Proof. The proof a simple induction on the length of $\gamma|_t$ where we reorder the first action of thread t using Lemma 3.46. □

Lemma 3.48. *Given a speculative computation $\gamma : C \xrightarrow{*} C'$ such that $\gamma = \gamma_0 \cdot \xrightarrow[t,o]{a} \cdot \gamma_1 \cdot \xrightarrow[t',o']{a'} \cdot \gamma_2$ with $a \# a'$ and $t \neq t'$ then there exists a computation $\gamma' : C \xrightarrow{*} C'$ such that $\gamma' = \gamma_0 \cdot \xrightarrow[t,o]{a} \cdot \gamma_0' \cdot \xrightarrow[t,o_0]{a_0} \cdot \xrightarrow[t_1,o_1]{a_1} \cdot \gamma_1' \cdot \xrightarrow[t',o']{a'} \cdot \gamma_2$ with $a_0 \# a_1$ and $t \neq t_1$, where possibly $\gamma_0' = \varepsilon$, $a_0 = a$ and $o_0 = o$; and also possibly $\gamma_1' = \varepsilon$ with $a_1 = a'$ and $o' = o_1$. Moreover for all thread $t \in \mathcal{T}id$ we have $\gamma|_t = \gamma'|_t$.*

Proof. The proof if by induction on the length of γ_1. The base case is trivial with $\gamma_0' = \varepsilon$ and $\gamma_1' = \varepsilon$, $o_0 = o$, $o_1 = o'$ and obviously $a_0 = a$ and $a_1 = a'$. For the induction case let us assume that $\gamma_1 = \xrightarrow[t_2,o_2]{a_2} \cdot \gamma_1'$ and consider the following cases:

- if $t \neq t_2$ and $a_0 \# a_2$ we have the conclusion. Otherwise ($\neg a_0 \# a_2$), we apply the Lemma 3.29 and conclude by the induction hypothesis.
- if $t = t_2$ we consider the following cases:
  - if $\gamma_1|_t = \gamma_1$ we can apply the Lemma 3.49 to conclude.
  - if $\gamma_1|_t \neq \gamma_1$ then let $\gamma_1 = \gamma_2' \cdot \xrightarrow[t_2,o_2]{a_2} \cdot \gamma_2''$ with $\gamma_2'|_t = \gamma_2'$ (i.e. γ_2' only contains actions of thread t) and $t \neq t_2$. Then if ($\xrightarrow[t,o]{a} \cdot \gamma_2' \# a_2$) we can conclude applying Lemma 3.49. Else we apply Lemma 3.46 to get $\xrightarrow[t_2,o_2]{a_2} \cdot \xrightarrow[t,o]{a} \cdot \gamma_2' \cdot \gamma_2''$ and conclude by the induction hypothesis.

□

Lemma 3.49. *Given a speculative computation $\gamma : C \xrightarrow{*} C'$ such that $\gamma = \gamma|_t \cdot \xrightarrow[t',o]{a}$ with $t \neq t'$ and $\gamma|_t \# a$ then there is a speculative computation $\gamma' : C \xrightarrow{*} C'$ and there exists subcomputations γ_0, γ_1, an action a' and an occurrence o' such that $\gamma' = \gamma_0 \cdot \xrightarrow[t,o']{a'} \cdot \xrightarrow[t',o]{a} \cdot \gamma_1$ with $a \# a'$ and $\gamma|_t = \gamma'|_t$.*

Proof. The proof is easy by induction on the length of $\gamma|_t$ and the application of the Lemma 3.29. □

The following proposition is the core of our robustness result.

Proposition 3.50. *Let C be a well-formed, closed configuration. If C is $\mathcal{D}$-posma and $\gamma : C \xrightarrow{*} C'$ is a $\mathcal{D}$-valid speculative computation, then there exists a normal coherent computation $\overline{\gamma}$ from C to C'.*

Proof. We have that since C is $\mathcal{D}$-posma, by Proposition 3.45 there is a $\mathcal{D}$-valid *normal* speculative computation $\gamma' : C \xrightarrow{*} C'$. Let us assume that γ' is not coherent, and let us find a coherent speculative computation departing from γ'. If γ' is not coherent we have $\gamma' = \gamma_0' \cdot ((S_0, L_0, T_0) \xrightarrow[t,o]{a} C_0') \cdot \gamma_0''$ such that $a = \mathsf{rd}^{\mathsf{o}}_{p,v} \Rightarrow S_0(p) \neq v$ and $a = \mathsf{cas}^{\mathsf{o}}_{p,tt} \Rightarrow S_0(p) \neq ff$ and finally $a = \mathsf{cas}^{\mathsf{o}}_{p,ff} \Rightarrow S_0(p) \neq tt$. Let us focus on the case of $a = \mathsf{rd}^{\mathsf{o}}_{p,v}$ the other cases being similar. Then $\gamma_0' = \delta_0 \cdot \xrightarrow[t,\bar{o}]{b} \cdot \delta_1 \cdot \xrightarrow[t',o']{b'} \cdot \delta_2$ with $b, b' \in \mathcal{MW}r_p$ where there are no further writes to p in δ_2 (meaning that this is the last action in $\mathcal{MW}r_p$), $t \neq t'$ and $\mathsf{match}[\gamma_0'/_t, (a, o)] = [\delta_0/_t, (b, \bar{o})]$. We have that since $b \# b'$ and these actions from different threads happen in γ_0' we can apply the Lemma 3.48 to obtain an equivalent $\overline{\gamma_0}$ such that there are two consecutive conflicting actions of t in $\overline{\gamma_0}$. Now, from the definition of $\mathcal{D}$-posma-coherent there are no conflicting actions relating thread t and any other thread in δ_2, else we would get a violation of the property $\mathcal{D}$-posma-coherent. Therefore we can apply the Lemma 3.47 to conclude that there is $\gamma'' : C \xrightarrow{*} C'$ and $\gamma'' = \delta_0 \cdot \xrightarrow[t,\bar{o}]{b} \cdot \delta_1 \cdot \delta_2/_t \cdot \xrightarrow[t,o]{a} \cdot \xrightarrow[t',o']{b'} \cdot \delta_2' \cdot \gamma_0''$, where δ_2' is δ_2 with the actions of t removed. In the case in which there are still writes of p by other threads in

δ_1 we repeat this argument inductively (on the number of such writes between the speculative read and its matching write) until there is none left.

□

We can finally conclude with our main result which is an obvious consequence of the above Proposition 3.50.

Theorem 3.51 (Robustness)**.** *Let e be a closed expression such that the configuration $(\emptyset, (t, e))$ is $\mathcal{D}$-posma-coherent. Then e is $\mathcal{D}$-robust.*

3.4 Conclusion

In this chapter we have considered an operational formalization of speculative computation. Moreover, we have considered the semantical effects of speculations for two different synchronization models: a high-level language (λ-lock) which provides locks for synchronizing threads, and a low-level language (λ-barrier) that relies only on barriers and compare-and-swap constructs to impose ordering and atomicity constraints on computations.

The semantic models we of this chapter are very permissive, in the sense that most of the common "litmus tests" present in the relaxed memory models literature can be captured by this formalization. Of particular interest is that we can model the relaxation of a read with respect to a previous read ($\mathbf{R} \to \mathbf{R}$) as shown in example 2.40, and the relaxation of a write with respect to a previous read ($\mathbf{R} \to \mathbf{W}$) as in example 1.4. These relaxations were not possible with the semantics of write buffers of the previous chapter.

Unlike the formalization of the previous chapter, the DRF guarantee does not hold for the sublanguages of this chapter. We have identified robustness properties that allow the programmer to recover an interleaving semantics for his/her programs. For the language with locks this property is a rather direct extension of the DRF guarantee to speculative computations. On the other hand, for the language λ-barrier we claim that in general entirely avoiding data races is, first, very hard, and second, not necessarily desirable. Hence we propose a property on computations (which resembles to the one proposed in [Shasha and Snir, 1988]) for the language λ-lock.

The discussions of this chapter and Chapter 2 have remained at a rather theoretical level, where two frameworks for describing operational semantics of relaxed memory models are given. In the next chapter we will use both of these frameworks to describe the semantics of the Sparc [SPARC, 1994] memory architecture. We hope that these formalizations provide support to our claim that operational models are feasible for relaxed memory models.

Chapter 4

Operational Specifications for the Sparc Family of Memory Models

In the previous chapters we have presented two general frameworks to formalize the semantics of relaxed memory models. However, we have been rather vague on how to specialize these frameworks to concrete models, that is memory models pertaining to commercially available architectures. To assess the potential of these operational theories in describing the semantics of relaxed memory models we formalize in this chapter the memory models of the Sparc architecture [SPARC, 1994]. Indeed, these memory models are not among the most complex ones, but the fact that they are well documented (see the formal description in the Appendix D of [SPARC, 1994]), and that they are incremental, with a uniform underlying basis, makes them attractive candidates for the kind of formalization we consider here. These models are also attractive from an "applicability" point of view, since in recent work Owens et al. [2009] advocate that the x86 architecture provides a memory model that belongs to the family of models we discuss here.

Additionally, the formalization of these memory models allows us to compare the techniques presented in the Chapters 2 and 3. As we shall see, the RMO memory model goes beyond the formalization of write buffers of Chapter 2, in fact the lack of reordering of read actions in the formalization of Chapter 2 makes it unsuited for formalizing RMO. On the other hand, speculative computations do allow this kind of reordering which makes speculations an appropriate technique to model RMO. Also, by means of these formalizations we can find and prove how these models differ from each other. We will show many examples in this chapter that provide support for a better understanding of the models in consideration.

Here we provide formalizations by means of both, the framework of write buffers, and the framework of speculations, for the TSO and PSO variants of Sparc. These formalizations, being of the same memory models and with the same language constructs, must be similar in some sense. We prove that our formalizations of PSO by means of write-buffers and by means of speculations are equivalent. This theorem supports the claim that the technique of speculations

is, to some extent, more general than the one of write buffering.

4.1 Preliminaries

The Sparc [SPARC, 1994] specification provides three different memory models TSO, PSO and RMO in increasing order of allowed relaxations (w.r.t. sequential consistency). One important characteristic of all the flavors of the Sparc memory model specification is that they allow for write-buffering. This relaxation will be present as a constant for all the models considered in this chapter. Let us now discuss the memory models of Sparc in order, from the less relaxed to the most relaxed:

TSO: The TSO acronym stands for *T*otal *S*tore *O*rdering. The only relaxation allowed by this memory model w.r.t. sequentially consistency is that reads are allowed to bypass preceding writes on different references. Indeed, this relaxation is very simple but is essential to capture the effects of write buffering. As we will see, this is the only flavor of Sparc models that imposes an order between writes, whether they are on the same reference or not. This constraint could explain the TSO terminology. A typical example for the relaxation allowed by TSO was considered in Example 1.2 and is reproduced in Figure 4.1a. One can see in this example, that if in the thread on the left we allow the write to p to be performed after the read of q, the depicted behavior becomes possible. An alternative explanation, by means of write buffers (cf. Chapter 2), is to say that the write to p has been issued but it is still standing in the buffers when the read of q is performed.

As in the example, allowing for this relaxation means that in general programs do not necessarily provide sequential consistent semantics. To circumvent the effects of this relaxation, the Instruction Set Architecture (ISA) of the Sparc architecture includes a barrier instruction (which we denote by $\langle$`wr`|`rd`$\rangle$ for simplicity, as we did in Chapter 3) that disallows reads appearing after the barrier in the program order, to be executed before the effects of any write that precedes the barrier are globally performed; or in terms of the semantics with buffers, before the write is committed into the main memory.

PSO Here the acronym stands for *P*artial *S*tore *O*rdering. In addition to the relaxations allowed by TSO, PSO allows two writes on different references to be performed in an order other than that of the program, thus the "partial" terminology in the name. An example of the behaviors allowed by PSO, but not by TSO can be seen in Figure 4.1b, which corresponds to Example 1.3 in the introduction of this thesis. Here we can simply perform, provided the above relaxation, the write of q before the one of p and thus the behavior shown is possible. To put it in terms of buffers, we can consider that writes on different references pending in a buffer can reach the memory in a different order than the one in which they were issued (or, equivalently, that for each processor there is one write buffer per memory location). This is indeed the case of the formalization we considered in Chapter 2. If we were to explain this example by means of the semantics with buffers, we simply update the buffer of q before the

$$\begin{bmatrix} p := 1\,; \\ r_0 := (!q) \end{bmatrix} \quad \| \quad \begin{bmatrix} q := 1\,; \\ r_1 := (!p) \end{bmatrix}$$

(a) TSO: $r_0 = r_1 = 0$

$$\begin{bmatrix} p := 1\,; \\ q := 1 \end{bmatrix} \quad \| \quad \begin{bmatrix} r_0 := (!q)\,; \\ r_1 := (!p) \end{bmatrix}$$

(b) PSO: $r_0 = 1$ & $r_1 = 0$

$$\begin{bmatrix} p := 1\,; \\ \langle \texttt{wr}|\texttt{wr} \rangle\,; \\ q := 1 \end{bmatrix} \quad \| \quad \begin{bmatrix} r_0 := (!q)\,; \\ r_1 := (!p) \end{bmatrix}$$

(c) RMO: $r_0 = 1$ & $r_1 = 0$

$$\begin{bmatrix} r_0 := (!q)\,; \\ p := 1 \end{bmatrix} \quad \| \quad \begin{bmatrix} r_1 := (!p)\,; \\ q := 1 \end{bmatrix}$$

(d) RMO: $r_0 = 1$ & $r_1 = 1$

Figure 4.1: Litmus tests of Sparc memory models

one of p. It should be clear that the example of Figure 4.1a is also possible in PSO since reads can still be reordered w.r.t. preceding writes.

As in the case of TSO, the ISA of Sparc provides an instruction to impose ordering between writes capable of being reordered. This barrier instruction, which we will denote here by $\langle \texttt{wr}|\texttt{wr} \rangle$ disables writes following the barrier (in the order of the program text) from being committed previously to any write that comes before the barrier.

RMO Finally we find RMO which is the most relaxed of all the Sparc memory models, and it stands for *R*elaxed *M*emory *O*rdering model. RMO adds to the relaxations of PSO the possibility to reorder reads w.r.t. preceding reads as well as writes w.r.t. preceding reads. Thus, in the example of Figure 4.1c (cf. Example 2.40), which is identical to the one of Figure 4.1b (cf. Example 1.4) with an explicit $\langle \texttt{wr}|\texttt{wr} \rangle$ barrier added on the thread on the left to disallow the reordering for these instructions, one can reorder the reads of the thread in the left to obtain the result in the figure. In the example on Figure 4.1d we depict a behavior that is only possible if we allow the reordering of reads w.r.t. subsequent writes on a different reference.

As one can expect, there are barriers present in the ISA of Sparc to prevent these reorderings. We will use the syntax $\langle \texttt{rd}|\texttt{wr} \rangle$ and $\langle \texttt{rd}|\texttt{rd} \rangle$ to denote the barriers that disallow reordering a write w.r.t. a previous read, and a read w.r.t. a previous read, respectively.

We can see the in Figure 4.2 a graphical representation of the hierarchy of Sparc memory models. This picture is a slight modification of the Figure 41 of the Sparc manual [SPARC, 1994]. It should be clear that more relaxations imply that more behaviors are possible. Hence, this graphic representation could be seen as both representing the set of behaviors allowed by the different memory models, or the set of relaxation w.r.t. sequential consistency allowed by them. More details about these memory models can be found in [Adve and Gharachorloo, 1996].

We could also consider the IBM 370 memory model, which is similar to TSO but where one cannot read values from writes still pending in the write buffers. This memory model can be easily modeled by means of the semantical models of Chapter 2, where a flush is required before reading, and also by the

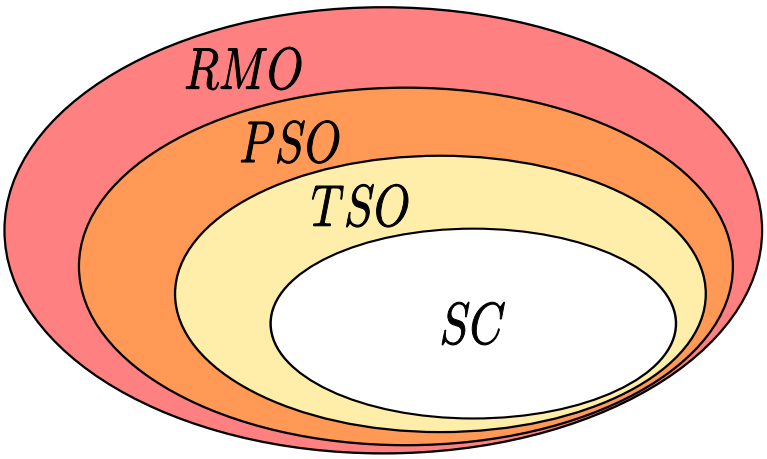


Figure 4.2: Relations of Sparc models

formalization of Chapter 3, where the $\mathsf{rd}^{\circ}_{p,v}$ and $\mathsf{cas}^{\circ}_{p,v}$ actions are discarded. However, we will not discuss this model in detail, since all our arguments on TSO apply in a rather straightforward way to the IBM 370 architecture.

4.2 The Language

Instead of introducing the full ISA of the Sparc specification we will consider a sublanguage of the λ-barrier language of previous chapters. Indeed, from the viewpoint of the specification of a relaxed memory model, the only significant actions are queries and updates of memory locations, and actions related to synchronization. Here we will consider only barriers and will discuss a simple *compare-and-swap* ($\mathsf{cas}\ v$) construct that we presented in the previous chapter, disregarding other instructions present in the Sparc architecture, as ldstub and casxa for example. These latter instructions can be dealt with in a way similar to our compare-and-swap construct. Unlike the λ-lock language of Chapter 3 we will dispose of the ($\mathsf{thread}\, e$) construct, since the specification of a machine architecture does not consider the creation of new "threads" (a simple analogy with processors should explain why). For the investigation of this chapter we will always consider systems with a fixed number of threads – or processors.

Recall from the previous chapter that we take a language in ANF, which can be seen as an intermediate language for the language of Chapter 2. Indeed it is not hard to see that the results from that chapter still hold if we consider the language in ANF, since the sequential semantics of both versions do not change as justified by Lemma 3.5.

Another difference with respect to the the real architecture, that we will consider at the language level is reference creation, induced in our language by the expression ($\mathsf{ref}\, v$). Of course no such instruction exist in the ISA of Sparc, but we consider that it is a reasonable assumption to have one such construct to describe the semantics of a programming language. In any case, this assumption has no important implications regarding the results we will develop in this chapter, alternatively we could have simply taken a fixed finite

number of references.

Without further ado, the language we will consider is:

$$
\begin{array}{rcll}
e & ::= & v \mid (ve_1) \mid (\mathsf{if}\ v\ \mathsf{then}\ e_0\ \mathsf{else}\ e_1) & \textit{expressions} \\
& \mid & (\mathsf{ref}\, v) \mid (!\, v) \mid (v_0 := v_1) & \\
& \mid & \langle \mathtt{wr}|\mathtt{rd} \rangle \mid \langle \mathtt{wr}|\mathtt{wr} \rangle \mid \langle \mathtt{rd}|\mathtt{wr} \rangle \mid \langle \mathtt{rd}|\mathtt{rd} \rangle & \textit{barriers} \\
& \mid & (\mathsf{cas}\ v) & \\
v & ::= & x \mid \lambda xe \mid \mathit{tt} \mid \mathit{ff} \mid () & \textit{values}
\end{array}
$$

Let us, for similarity with the previous chapter, call this language λ-barrier.

Indeed some considerations regarding atomicity are due. The *compare-and-swap* construct we present here – which is the same as in the previous chapter – resembles (which is not necessarily a coincidence) to that of the Sparc architecture in several respects. Some implementations of compare-and-swap in other architectures [Sewell et al., 2010] impose barrier semantics to this construct, and this fact is often taken for granted. However in the Sparc specification [SPARC, 1994], as in the semantics of the previous chapter, this is not the case. We will assume here that the $(\mathsf{cas}\ v)$ construct does not imply ordering constraints per se, but is subject to the ones imposed by barriers and the particular memory model in consideration. This allows us to separate the concerns of atomicity with that of ordering, that in general need not be bound. In particular, and consistently with the previous chapter, the $(\mathsf{cas}\ v)$ construct will be considered to have both read and write semantics, therefore it will be affected by barriers imposing ordering on both reads and writes. Nevertheless, in the absence of barriers, the memory model requirements for writes and reads will apply to $(\mathsf{cas}\ v)$, meaning for example that in RMO, which allows the reordering of reads w.r.t. subsequent writes, an event produced by a $(\mathsf{cas}\ v)$ instruction can be reordered w.r.t. to a subsequent write as well. Notice that if we were to consider a refined $(\mathsf{cas}\ v)$ instruction implying barrier semantics, as with the Intel x86 read-modify-write instructions [Intel Corporation, 2007; Owens et al., 2009], it would be sufficient to add the proper barriers surrounding each compare-and-swap action in the formalization of this chapter. In that sense we adopt a more general approach here, which fortunately coincides with the Sparc specification. It must be acknowledged that compare-and-swap is not the only instruction Sparc provides for atomicity, some other instructions are casxa, ldstub and variants of these. However, the memory model specificities of these instructions are very similar in essence to that of the $(\mathsf{cas}\ v)$ construct we provide here and adding all these instructions would only clutter our definitions providing very little (if any) additional insights on aspects related to the memory model. Finally, in this work we do not consider the atomicity aspects of accesses to a single memory location. In fact, accessing double words in some architectures is not atomic, and instructions are provided to guarantee atomic accesses to this kind of locations. It would not be hard to consider an instruction that atomically reads or modifies two memory locations, but we prefer to obviate this issue throughout this work, since it is not of great importance as regards the results we target.

We shall not discuss our language in more detail, since we have already done that in Chapter 3. The runtime language, the redexes and the evaluation

contexts are given by:

$$
\begin{array}{lcll}
e & ::= & \ldots \mid (\lambda v^{?} e_0\ e_1) & \textit{expressions} \\
v & ::= & \ldots \mid p \mid (\lambda v^{?} e_0) & \textit{values} \\
r & ::= & (\lambda x e v) \mid (\lambda v^{?} e v) \mid (\mathsf{if}\ v\ \mathsf{then}\ e_0\ \mathsf{else}\ e_1) & \textit{redexes} \\
 & \mid & (\mathsf{ref}\ v) \mid (\mathbf{!}\, p) \mid (p := v) \mid (\mathsf{cas}\ p) & \\
 & \mid & \langle \mathtt{wr}|\mathtt{rd}\rangle \mid \langle \mathtt{wr}|\mathtt{wr}\rangle \mid \langle \mathtt{rd}|\mathtt{wr}\rangle \mid \langle \mathtt{rd}|\mathtt{rd}\rangle & \\
\mathbf{E} & ::= & [] \mid (v\mathbf{E}) & \textit{evaluation contexts}
\end{array}
$$

where references p are sampled from the infinite set $\mathcal{R}ef$ of reference names and the expression $(\lambda v^{?} e_0\ e_1)$ is the as in the previous chapter.

As before, we need to present the actions produced by transitions in the semantics. These are:

$$
\begin{array}{lcl}
a \in \mathcal{A}ct & ::= & \beta \mid \beta_v \mid \swarrow \mid \searrow \mid \nu_{p,v} \mid \mathsf{wr}_{p,v} \\
 & \mid & \mathsf{rd}_{p,v} \mid \mathsf{rd}^{\circ}_{p,v} \mid \mathsf{cas}_{p,v} \mid \mathsf{cas}^{\circ}_{p,v} \mid b \\
b \in \mathcal{B}ar & ::= & \mathtt{wr} \mid \mathtt{ww} \mid \mathtt{rw} \mid \mathtt{rr}
\end{array}
$$

All of these actions have already been presented in Chapter 3. However, it should be noticed that the semantics of the actions $\mathsf{rd}^{\circ}_{p,v}$ and $\mathsf{cas}^{\circ}_{p,v}$ changes slightly here to model the exact behavior of write buffers, which is slightly different from the one considered in the previous chapter as we shall see. Indeed, the intended semantics of the $\mathsf{rd}^{\circ}_{p,v}$ action will be twofold. In the case of the formalization with write-buffers, we will use this action to differentiate reads that are satisfied directly by the buffer from those that obtain their value from the store. Notice that in that sense, the distinction between $\mathsf{rd}_{p,v}$ and $\mathsf{rd}^{\circ}_{p,v}$ is almost an annotation. On the other hand, in the formalization by means of speculations, this action takes its value from the last write by the same thread (where last is w.r.t. the program order), which somehow has the same effect as reading from an uncommitted write in the semantics of write-buffers. As we did in Chapter 3 the validity definition will indeed guarantee that the last write is read. Moreover, to model exactly the behavior of write buffers additional constraints will be imposed by the validity condition of this formalization. Similar considerations apply to the $\mathsf{cas}^{\circ}_{p,v}$ construct. We will come back to these issues when presenting the formalization by means of speculations.

We notice at this point that if a model like the IBM 370 (according to [Adve and Gharachorloo, 1996]) was to be considered, then the $\mathsf{rd}^{\circ}_{p,v}$ and $\mathsf{cas}^{\circ}_{p,v}$ can be discarded. In this case, our definitions and results would be greatly simplified.

One can observe here, that since the cas_v and $\mathsf{cas}^{\circ}_{p,v}$ actions correspond to read and write actions at the same time, most of the definitions and results regarding writes and/or reads also apply in a rather obvious way to these actions. In particular, notice that since we consider a $\mathsf{cas}_{p,v}$ action to have the semantics of a $\mathsf{wr}_{p,v'}$ and a $\mathsf{rd}_{p,w}$ at the same time (with v' and w directly determined by v), and the $\mathsf{cas}^{\circ}_{p,v}$ action to have the semantics of a $\mathsf{wr}_{p,v'}$ and a $\mathsf{rd}^{\circ}_{p,w}$ at the same time, we do not need to mention the action $\mathsf{cas}_{p,v}$ explicitly in every definition, since the considerations for $\mathsf{rd}_{p,v}$,$\mathsf{rd}^{\circ}_{p,v}$ and $\mathsf{wr}_{p,v}$ uniquely determine similar considerations for $\mathsf{cas}_{p,v}$ and $\mathsf{cas}^{\circ}_{p,v}$ actions. This simplification will greatly improve the presentation of our results without making them less formal. To reassure the reader we will provide intuitions on the implication of our definitions on these actions whenever they are concerned.

Finally we will adopt the following simplification as regards the languages for the different memory models (TSO, PSO and RMO). We will consider three different sublanguages with the appropriate barriers in each. Let us call λ-RMO the language for RMO which is λ-barrier (without dynamic thread creation). We will call λ-PSO the sublanguage of λ-barrier that does not include the $\langle \texttt{rd}|\texttt{wr} \rangle$ and the $\langle \texttt{rd}|\texttt{rd} \rangle$ constructs. And finally we will consider λ-TSO to be the sublanguage of λ-barrier that does not include $\langle \texttt{wr}|\texttt{wr} \rangle$, $\langle \texttt{rd}|\texttt{wr} \rangle$ and $\langle \texttt{rd}|\texttt{rd} \rangle$. We could have alternatively observed that the additional barriers do not have any effect when they are not concerned by the memory model in consideration, for instance, adding a $\langle \texttt{wr}|\texttt{wr} \rangle$ in a program for TSO does not harm the program, and the executions are identical. However this approach only adds burden to the proofs, and this is the reason why we prefer to disregard barriers entirely whenever they do not add any expressive power to the memory model in consideration. We obviously have the following set inclusion relations between these languages:

$$\lambda - \text{TSO} \subset \lambda - \text{PSO} \subset \lambda - \text{RMO} = \lambda - \text{barrier}$$

4.3 PSO and TSO: a Formalization with Write Buffers

The operational semantics of write buffers that we presented in Chapter 2 does not perform any kind of speculation. Actually, the reordering effects that we can simulate by means of this semantics are due to the structure of buffers, and the buffer update rules only. Observe as well that the operational semantics of write-buffers of Chapter 2 does not support the relaxations $\mathbf{R} \rightarrow \mathbf{R}$ and $\mathbf{R} \rightarrow \mathbf{W}$, that is the reordering of reads w.r.t. subsequent actions, that we have discussed in the previous chapter. This limitation motivated in part our investigations on speculative computation of Chapter 3. The only memory model of the Sparc family to allow these kind of relaxations is RMO, which implies that modeling RMO by means of the framework of write-buffers would be somewhat unnatural. As a consequence we concentrate on TSO and PSO in this section. In the next section we will see how we can model RMO (as well as TSO and PSO) by means of the framework of speculative computations.

As we anticipated in Chapter 2 we will use almost the same syntactic constructions for buffers as defined there. However, barriers have nontrivial interactions with buffers, and importantly, barriers whose intention is to constraint the reordering of actions of a certain kind should not impose constraints on memory accesses of a different kind. For instance in PSO, upon encountering a $\langle \texttt{wr}|\texttt{wr} \rangle$ barrier it is not realistic to require that the buffers be emptied before proceeding – the approach we followed for the unlock action in Chapter 2 – since that would impose restrictions upon subsequent read actions as well, which is not the intended semantics of this barrier. In fact, flushing the buffer would be sufficient and sound for TSO, since the only available relaxation is of reads w.r.t. preceding writes, and the only barrier is $\langle \texttt{wr}|\texttt{rd} \rangle$ which requires writes to be performed before the subsequent read. However we will consider a more general formalization for these constraints by means of buffers which, additionally, is a requirement for the modeling of PSO.

The key modification to the syntax of buffers is that we now incorporate

barrier symbols in the buffers – we will actually use the same symbols as for the barrier actions. Consequently, the new definition of buffers becomes:

$$B ::= \ \epsilon \ \mid \ B \triangleleft [p \mapsto v] \ \mid \ B \triangleleft [b]$$

where $b \in \mathcal{B}ar$. In the semantics – that we shall present shortly – barrier instructions have as their only effect adding the corresponding symbol into the buffer. The actual constraints imposed by barriers will be reflected by the way in which pending writes in the buffers are updated into the memory and by the way in which one can read into the buffers. Barriers are included at the right of the buffers (or bottom cf. Figure 2.1) and are removed only when reaching the left end (or the top respectively). In this way, a read action that finds a `wr` symbol in its corresponding buffer cannot proceed, since there are possibly pending writes in the buffer that were issued before the barrier. That means that a barrier symbol `wr` in the buffer constraints the use of $\mathsf{rd}_{p,v}$ actions. Similarly, a pending write (present in a buffer) cannot be updated if there is a previously issued `ww` barrier symbol in the same buffer. Notice that since here we only consider λ-PSO and λ-TSO (i.e. we do not consider λ-RMO) the only meaningful barriers at this point are `ww` and `wr`.

We will denote by $[p \mapsto v] \triangleright B$ (or $[b] \triangleright B$) the buffer that has as first element the update $[p \mapsto v]$ (or the barrier b respectively).

The way in which the elements of the buffer can be accessed and updated is fundamental in describing the semantics. To formalize the semantics of reads and buffer updates we shall use the following function which retrieves from a given buffer B and a reference p the sequence of barriers and values on that reference that can be found in the buffer. The returned sequence preserves the order of the pending updates in the buffer:

$$B(p) \ \triangleq \ \begin{cases} \varepsilon & \text{if } B = \epsilon \\ B'(p) \cdot b & \text{if } B = B' \triangleleft [b] \\ B'(p) \cdot v & \text{if } B = B' \triangleleft [p \mapsto v] \\ B'(p) & \text{if } B = B' \triangleleft [q \mapsto v] \ \& \ q \neq p \end{cases}$$

To be more concrete about how the barrier symbols impose constraints on the actions concerning the barrier, we require that a read action $\mathsf{rd}^{\circ}_{p,v}$ – we anticipate here that reads satisfied from the buffer will generate a $\mathsf{rd}^{\circ}_{p,v}$ action and reads satisfied from the store will generate $\mathsf{rd}_{p,v}$ actions – reading within a buffer B can succeed only if the corresponding $B(p)$ sequence has the following shape: $B(p) = s \cdot v \cdot \mathtt{ww}^n$ for a sequence s such that `wr` does not occur in s, and where we denote by $\mathtt{ww}^n$ the sequence of n consecutive `ww` symbols. These constraints mean that there must be no `wr` barrier symbols in the buffer, and v has to be the last value of a pending write for p in B. A simple example to see why we require these constraints is the following, where configurations have the form $C = (S, T)$, with S the store and the thread pool T contains threads of the shape (B, t, e) with B being the buffer of the processor, $t \in \mathcal{T}id$ is a unique

thread identifier and e is the program expression:

$$(\{p \mapsto 0, q \mapsto 0\}, (\epsilon, t, p := 1\,;\langle \mathtt{wr}|\mathtt{rd}\rangle\,; q := 1\,; r_0 := (!q))) \xrightarrow[t,[]]{\mathsf{wr}_{p,1}}$$
$$(\{p \mapsto 0, q \mapsto 0\}, (\epsilon \lhd [p \mapsto 1], t, \langle \mathtt{wr}|\mathtt{rd}\rangle\,; q := 1\,; r_0 := (!q))) \xrightarrow[t,[]]{\mathtt{wr}}$$
$$(\{p \mapsto 0, q \mapsto 0\}, (\epsilon \lhd [p \mapsto 1] \lhd [\mathtt{wr}], t, q := 1\,; r_0 := (!q))) \xrightarrow[t,[]]{\mathsf{wr}_{q,1}}$$
$$(\{p \mapsto 0, q \mapsto 0\}, (\epsilon \lhd [p \mapsto 1] \lhd [\mathtt{wr}] \lhd [q \mapsto 1], t, r_0 := (!q)))$$

If we denote by B the buffer $\epsilon \lhd [p \mapsto 1] \lhd [\mathtt{wr}] \lhd [q \mapsto 1]$ we see that $B(q) = \mathtt{wr} \cdot 1$. Indeed, if we were to perform a $\mathsf{rd}^{\circ}_{q,1}$ action in the last configuration we could read the value 1 which is the last value for reference q standing in the buffer B. However, doing so would imply that the previous write of p has not been made globally visible when the read returns a value; contradicting the purpose of the $\langle \mathtt{wr}|\mathtt{rd}\rangle$ instruction in between them. That is why we require that the sequence s does not contain $\mathtt{wr}$ symbols when performing a $\mathsf{rd}^{\circ}_{p,v}$ action. Similarly, a read from the store, i.e. $\mathsf{rd}_{p,v}$, requires the buffer B to contain no pending write on p as well as no pending barrier $\mathtt{wr}$, thus the condition for read actions on the store will be $B(p) = \mathtt{ww}^n$. Let us consider a computation of the following simple example of a thread that has a write followed by a read with a $\langle \mathtt{wr}|\mathtt{rd}\rangle$ barrier in between them (in particular this is how we can make the example of Figure 4.1a sequentially consistent). We assume here that configurations are similar to those of the Chapter 3.

$$(\{p \mapsto 0, q \mapsto 0\}, (\epsilon, t, p := 1\,;\langle \mathtt{wr}|\mathtt{rd}\rangle\,; r_0 := (!q))) \xrightarrow[t,[]]{\mathsf{wr}_{p,1}}$$
$$(\{p \mapsto 0, q \mapsto 0\}, (\epsilon \lhd [p \mapsto 1], t, \langle \mathtt{wr}|\mathtt{rd}\rangle\,; r_0 := (!q))) \xrightarrow[t,[]]{\mathtt{wr}}$$
$$(\{p \mapsto 0, q \mapsto 0\}, (\epsilon \lhd [p \mapsto 1] \lhd [\mathtt{wr}], t, r_0 := (!q)))$$

We can see that in the final configuration $(\epsilon \lhd [p \mapsto 1] \lhd [\mathtt{wr}])(q) = \mathtt{wr}$. This means, as we will formalize soon, that we cannot proceed with a $\mathsf{rd}_{q,0}$ nor a $\mathsf{rd}^{\circ}_{q,0}$ action. In essence this condition requires that the $\mathtt{wr}$ symbol be eliminated from the buffer before proceeding with the subsequent read of q. Moreover, the conditions for updating the buffers will require that there be no previous uncommitted writes before removing the $\mathtt{wr}$ symbol from the buffer; that is to say that the update of p will have to be performed before the read of q actually takes place. In this way barrier symbols impose constraints on actions. The condition to update a pending write in PSO will require $B(p)$ to have the following shape: $B(p) = \mathtt{wr}^n \cdot v \cdot s$, meaning that there is a value v that can be updated and there are no previously issued $\mathtt{ww}$ actions pending for update. If there were any such $\mathtt{ww}$ updating the write of p could bypass a previous update on a different reference, which would be against the intended behavior of the barrier.

The following auxiliary function returns for a given buffer B and a reference p, the buffer with the first-in pending write on reference p removed from B. This function will be used for describing the buffer resulting from updating a

$$
\begin{array}{rcl}
\mathbf{E}[(\lambda x e v)] & \xrightarrow{\beta_v} & \mathbf{E}[\lambda v^?\{x \mapsto v\}ev] \\
\mathbf{E}[(\lambda v^? e v)] & \xrightarrow{\beta} & \mathbf{E}[e] \\
\mathbf{E}[(\mathsf{if}\ tt\ \mathsf{then}\ e_0\ \mathsf{else}\ e_1)] & \xrightarrow{\swarrow} & \mathbf{E}[e_0] \\
\mathbf{E}[(\mathsf{if}\ ff\ \mathsf{then}\ e_0\ \mathsf{else}\ e_1)] & \xrightarrow{\searrow} & \mathbf{E}[e_1] \\
\mathbf{E}[(\mathsf{ref}\ v)] & \xrightarrow{\nu_{p,v}} & \mathbf{E}[p] \\
\mathbf{E}[(p := v)] & \xrightarrow{\mathsf{wr}_{p,v}} & \mathbf{E}[()] \\
\mathbf{E}[(!\, p)] & \xrightarrow{\mathsf{rd}_{p,v}} & \mathbf{E}[v] \\
\mathbf{E}[(!\, p)] & \xrightarrow{\mathsf{rd}^{\circ}_{p,v}} & \mathbf{E}[v] \\
\mathbf{E}[(\mathsf{cas}\ p)] & \xrightarrow{\mathsf{cas}_{p,v}} & \mathbf{E}[v] \\
\mathbf{E}[(\mathsf{cas}\ p)] & \xrightarrow{\mathsf{cas}^{\circ}_{p,v}} & \mathbf{E}[v] \\
\mathbf{E}[\langle \mathtt{wr} | \mathtt{rd} \rangle] & \xrightarrow{\mathtt{wr}} & \mathbf{E}[()] \\
\mathbf{E}[\langle \mathtt{wr} | \mathtt{wr} \rangle] & \xrightarrow{\mathtt{ww}} & \mathbf{E}[()]
\end{array}
$$

Figure 4.3: λ-PSO Semantics of Single Expressions

write to reference p from B into the store.

$$
B{\downarrow}p \triangleq \begin{cases} B & \text{if } B(p) \text{ contains no } v \in \mathcal{V}al \\ (B'{\downarrow}p) \triangleleft u & \text{if } B = B' \triangleleft u \ \&\ B'(p) \text{ contains a } v \in \mathcal{V}al \\ B' & \text{if } B = B' \triangleleft [p \mapsto u] \ \&\ B'(p) \text{ contains no } v \in \mathcal{V}al \end{cases}
$$

4.3.1 Write buffers semantics

Once more, we present the semantics of single expressions first in Figure 4.3 which has very little variations from the rules of Figure 3.2 of the previous chapter. Let us recall that the actions $\mathsf{rd}^{\circ}_{p,v}$ and $\mathsf{cas}^{\circ}_{p,v}$ will be used in the semantics with write-buffers to distinguish reads from the buffers from reads from the store. In some sense we can consider that $\mathsf{rd}^{\circ}_{p,v}$ and $\mathsf{cas}^{\circ}_{p,v}$ are speculative since the values that they return are not yet globally visible, or "performed".

Provided with the semantics of single expressions we proceed to define the configurations required to describe the semantics of parallel programs. This configurations should not surprise the reader, since they are very similar to the ones we have considered before.

$$C = (S, (B, t, e) \| T)$$

An unimportant difference w.r.t. to the configurations we used in Chapter 2 is that, in the absence of locks, no lock pool is needed. A more interesting difference regards the thread components of the thread system. Notice that threads are now composed of a thread identifier t sampled from the infinite set $\mathcal{T}id$, a buffer B and finally the thread expression e. One can see that there is

$$\frac{B(p) = \mathtt{wr}^n \cdot v \cdot s \qquad S' = S\{p \leftarrow v\}}{(S,(B,t,e)\|T) \xrightarrow[t,\varepsilon]{\mathsf{bu}_{p,v}} (S',(B{\downarrow}p,t,e)\|T)} \qquad \frac{B = b \triangleright B' \qquad b \in \mathcal{B}ar}{(S,(B,t,e)\|T) \xrightarrow[t,\varepsilon]{\overline{b}} (S,(B',t,e)\|T)}$$

$$\frac{e \xrightarrow{a} e'}{(S,(B,t,e)\|T) \xrightarrow[t,\varepsilon]{a} (S',(B',t,e')\|T)}\ (*)$$

Figure 4.4: PSO with Write Buffers

a single buffer per thread here. Actually, since we are considering static thread systems (i.e. the number of threads does not change in the execution of the program) we need not consider the hierarchical structure of buffers as presented in Chapter 2. The reason for considering static threads systems is that in the architectural specification one deals with processors rather than threads, therefore creation of threads (or processors!) is not part of the specification, this is also the reason why we have a single buffer per thread, which is here the code running in some processor. Having a single buffer for each thread greatly simplifies the rules for updating buffers. Of course, we will adopt the standard well-formedness constraint for configurations.

To compare the formalization by means of write-buffers with that of speculations we will be interested in identifying the actions of updating the contents of the buffers into the memory. This actions did not have a label in the semantics of write-buffers of Chapter 2, and were called "silent" actions there. On the other hand, here we will need to identify these actions since they will indicate how the computations of both formalizations relate. We will see these transitions play an important role in the definition of the conflict relation, since changing the order in which two buffer updates (of different threads) are performed could result in different final configurations. For that purpose we include in the syntax of (global) actions the buffer update actions which we shall denote by $\mathsf{bu}_{p,v}$ and $\overline{b}$ for the action of removing a barrier b from the buffers. We will refer generically to these actions as commit actions. A buffer update can be seen as the commit of a write, since it makes it globally visible, and the removal of a barrier symbol from the buffer means that all the constraints this barrier imposed have already been satisfied. The exact rules governing the use of these actions depend on the memory model in consideration. The ones for PSO are presented below.

PSO: Let us now concentrate on the semantics of λ-PSO, which is presented in Figure 4.4 where the constraint $(*)$ is unfolded to:

$$(*)\begin{cases} a \in \{\beta, \swarrow, \searrow\} & \Rightarrow & S' = S \;\&\; B' = B \\ a = \beta_v & \Rightarrow & \mathsf{FRef}(v) \subseteq \mathsf{dom}(S) \\ a = \nu_{p,v} & \Rightarrow & p \notin \mathsf{dom}(S) \;\&\; S' = S \cup \{p \mapsto v\} \\ a = \mathsf{wr}_{p,v} & \Rightarrow & B' = B \triangleleft [p \mapsto v] \\ a = \mathsf{rd}_{p,v} & \Rightarrow & S(p) = v \;\&\; B(p) = \mathtt{ww}^n \\ a = \mathsf{rd}^{\circ}_{p,v} & \Rightarrow & B(p) = s \cdot v \cdot \mathtt{ww}^n \;\&\; \mathtt{wr} \text{ does not occur in } s \\ a = \mathsf{cas}_{p,ff} & \Rightarrow & S(p) = tt \;\&\; B(p) = \mathtt{ww}^n \\ a = \mathsf{cas}_{p,tt} & \Rightarrow & S(p) = ff \;\&\; B(p) = \mathtt{ww}^n \;\&\; B' = B \triangleleft [p \mapsto tt] \\ a = \mathsf{cas}^{\circ}_{p,ff} & \Rightarrow & B(p) = s \cdot tt \cdot \mathtt{ww}^n \;\&\; \mathtt{wr} \text{ does not occur in } s \\ a = \mathsf{cas}^{\circ}_{p,tt} & \Rightarrow & B(p) = s \cdot ff \cdot \mathtt{ww}^n \;\&\; \mathtt{wr} \text{ does not occur in } s \\ & & \quad \&\; B' = B \triangleleft [p \mapsto tt] \\ a \in \{\mathtt{ww}, \mathtt{wr}\} & \Rightarrow & B' = B \triangleleft [a] \end{cases}$$

There are important changes from the constraints of the write buffering semantics of Chapter 2 that deserve being observed. Most of these differences are due to the syntax, but a significant difference can be seen in the treatment we gave to synchronization in Chapter 2 with respect to the semantics of barriers here. While in Chapter 2 releasing a lock required flushing the buffers, here barrier actions just add their corresponding symbol in the buffers. Then, the presence of barriers restricts the execution of subsequent actions that depend on them. Notice that $\mathsf{rd}_{p,v}$ directly reads into the memory provided that the buffer associated with the thread performing the read action contains no pending writes on p or `wr` barriers. The action $\mathsf{rd}^{\circ}_{p,v}$ requires that the last value for p in the buffer be v, and moreover, it verifies that there are no pending `wr` actions in the buffers. There are several rules regarding the compare-and-swap construct, based on the action ($\mathsf{cas}_{p,v}$ or $\mathsf{cas}^{\circ}_{p,v}$) and on the possible result. The first two, $\mathsf{cas}_{p,ff}$ and $\mathsf{cas}_{p,tt}$ directly obtain their value from the memory, and thus check that $B(p)$ contains no pending write on p or a `wr` barrier. Finally the actions $\mathsf{cas}^{\circ}_{p,tt}$ and $\mathsf{cas}^{\circ}_{p,ff}$ read from the buffers, as the action $\mathsf{rd}^{\circ}_{p,v}$ does. As with $\mathsf{rd}^{\circ}_{p,v}$ we have to verify that no pending `wr` barriers are pending in the buffer.

We have two rules that regard buffer updates. The rule on the right of Figure 4.4 states that a barrier in a buffer can be eliminated from it upon reaching the top of the buffer, that is, once all previously issued write actions have been performed. Notice that this action produces an action symbol $\overline{b}$. The buffer update rule simply takes the first value in the buffer of a reference p, nondeterministically chosen, and updates the memory provided that there are no previously issued `ww` actions in the buffer. Notice that all the buffer update rules, either the removal of a barrier, or the removal of a pending update are nondeterministically executed both as regards the thread and the reference that is updated.

Two simple examples that illustrate the formalization of λ-PSO are 1.2 and 1.3. Let us concentrate on the latter one, since the former one also applies to λ-TSO, and we will treat it after presenting the formalization for λ-TSO. We

recall the example 1.3 for clarity:

$$\begin{bmatrix} p := 1\,; \\ q := 1 \end{bmatrix} \quad \| \quad \begin{bmatrix} r_0 := (!q)\,; \\ r_1 := (!p) \end{bmatrix}$$

The behavior in question here is whether $r_0 = 1$ and $r_1 = 0$ is allowed. As the reader knows by now, the behavior is possible for our semantics of λ-PSO, and the justifying computation has already been described in the page 43 (Chapter 2). However, since in Chapter 2 the buffer structure is more complex than here we will present the justifying computation once more, omitting the store and the registers r_0 and r_1 for clarity, and assuming that the initial store has $p = q = 0$.

$$\begin{array}{rcll}
\langle\epsilon\rangle(p := 1\,;\ q := 1) & \| & \langle\epsilon\rangle((!q)\,;\,(!p)) & \xrightarrow[t_0]{\mathsf{wr}_{p,1}} \\
\langle\epsilon \triangleleft [p \mapsto 1]\rangle(q := 1) & \| & \langle\epsilon\rangle((!q)\,;\,(!p)) & \xrightarrow[t_0]{\mathsf{wr}_{q,1}} \\
\langle\epsilon \triangleleft [p \mapsto 1] \triangleleft [q \mapsto 1]\rangle() & \| & \langle\epsilon\rangle((!q)\,;\,(!p)) & \xrightarrow[t_0]{\mathsf{bu}_{q,1}} \\
\langle\epsilon \triangleleft [p \mapsto 1]\rangle() & \| & \langle\epsilon\rangle((!q)\,;\,(!p)) & \xrightarrow[t_1]{\mathsf{rd}_{q,1}} \\
\langle\epsilon \triangleleft [p \mapsto 1]\rangle() & \| & \langle\epsilon\rangle(!p) & \xrightarrow[t_1]{\mathsf{rd}_{p,0}} \\
\langle\epsilon \triangleleft [p \mapsto 1]\rangle() & \| & \langle\epsilon\rangle(0) & \xrightarrow[t_0]{\mathsf{bu}_{p,1}} \\
\langle\epsilon\rangle() & \| & \langle\epsilon\rangle(0) &
\end{array}$$

Let us now consider a slight modification of this program where we add barriers to make the program sequentially consistent. As we can see from the previous computation, the reason why the behavior can happen, is because the writes can update the memory in an order other than the one in which they where issued. To avoid such reordering we add a barrier between the two writes, which is example 2.40.

$$\begin{bmatrix} p := 1\,; \\ \langle \mathtt{wr} | \mathtt{wr} \rangle\,; \\ q := 1 \end{bmatrix} \quad \| \quad \begin{bmatrix} r_0 := (!q)\,; \\ r_1 := (!p) \end{bmatrix}$$

Let us try to reproduce the computation above, where we omit the store for clarity:

$$\begin{array}{rcll}
\langle\epsilon\rangle(p := 1\,;\ \langle \mathtt{wr} | \mathtt{wr} \rangle\,;\ q := 1) & \| & \langle\epsilon\rangle((!q)\,;\,(!p)) & \xrightarrow[t_0]{\mathsf{wr}_{p,1}} \\
\langle\epsilon \triangleleft [p \mapsto 1]\rangle(\langle \mathtt{wr} | \mathtt{wr} \rangle\,;\ q := 1) & \| & \langle\epsilon\rangle((!q)\,;\,(!p)) & \xrightarrow[t_0]{\mathtt{ww}} \\
\langle\epsilon \triangleleft [p \mapsto 1] \triangleleft [\mathtt{ww}]\rangle(q := 1) & \| & \langle\epsilon\rangle((!q)\,;\,(!p)) & \xrightarrow[t_0]{\mathsf{wr}_{q,1}} \\
\langle\epsilon \triangleleft [p \mapsto 1] \triangleleft [\mathtt{ww}] \triangleleft [q \mapsto 1]\rangle() & \| & \langle\epsilon\rangle((!q)\,;\,(!p)) &
\end{array}$$

If we want to follow the computation we considered before, the write of q has to be updated before the one of p. However, in this configuration we cannot do so, since we have $(\epsilon \triangleleft [p \mapsto 1] \triangleleft [\mathtt{ww}] \triangleleft [q \mapsto 1])(q) = \mathtt{ww} \cdot 1$, and the condition for updating the buffer in Figure 4.4 requires that there be no pending $\mathtt{ww}$ barriers previous to the value to be updated. Therefore, in this example there is no

choice but to update the write of p first, or read q in the second thread, and both options render sequentially consistent computations. Just to finish the example, let us choose the former option:

$$\begin{array}{rcll}
\langle \epsilon \triangleleft [p \mapsto 1] \triangleleft [\mathtt{ww}] \triangleleft [q \mapsto 1] \rangle () & \| & \langle \epsilon \rangle ((!q)\,;\,(!p)) & \xrightarrow[t_0]{\mathsf{bu}_{p,1}} \\
\langle \epsilon \triangleleft [\mathtt{ww}] \triangleleft [q \mapsto 1] \rangle () & \| & \langle \epsilon \rangle ((!q)\,;\,(!p)) & \xrightarrow[t_0]{\overline{\mathtt{ww}}} \\
\langle \epsilon \triangleleft [q \mapsto 1] \rangle () & \| & \langle \epsilon \rangle ((!q)\,;\,(!p)) & \xrightarrow[t_0]{\mathsf{bu}_{q,1}} \\
\langle \epsilon \rangle () & \| & \langle \epsilon \rangle ((!q)\,;\,(!p)) & \xrightarrow[t_1]{\mathsf{rd}_{q,1}} \cdot \xrightarrow{*} \cdot \xrightarrow[t_1]{\mathsf{rd}_{p,1}} \\
\langle \epsilon \rangle () & \| & \langle \epsilon \rangle (1) &
\end{array}$$

TSO: In fact, λ-TSO is easy to describe by just simplifying the rules given for λ-PSO since its semantics is more restrictive than the one of λ-PSO. Since the language λ-TSO does not have a $\langle \mathtt{wr}|\mathtt{wr} \rangle$ construct (writes cannot be reordered in TSO) the conditions where the term $\mathtt{ww}^n$ appear have that part of the constraint trivially satisfied with $n = 0$. The only rule that changes is the one for updating the memory by flushing the contents of the buffers. This rule is now as follows,

$$\frac{B = [p \mapsto v] \triangleright B' \qquad S' = S\{p \leftarrow v\}}{(S, (B, t, e) \| T) \xrightarrow[t,\varepsilon]{\mathsf{bu}_{p,v}} (S', (B', t, e) \| T)}$$

where we can see that pending writes have to be updated in the order in which they were added in the buffer regardless of the location. In other words, pending writes cannot bypass other pending writes in the buffers despite the fact that they can be on different references.

Let us reconsider the Example 1.2 which regards TSO and that we discussed in the introduction of this chapter.

$$\begin{bmatrix} p := 1\,; \\ r_0 := (!q) \end{bmatrix} \quad \| \quad \begin{bmatrix} q := 1\,; \\ r_1 := (!p) \end{bmatrix}$$

A possible computation that justifies the result $r_0 = r_1 = 0$ is the following, where once more, we omit the store and the registers, and we assume that the store initially holds $p = q = 0$:

$$\begin{array}{rcll}
\langle \epsilon \rangle (p := 1\,;\,(!q)) & \| & \langle \epsilon \rangle (q := 1\,;\,(!p)) & \xrightarrow[t_0]{\mathsf{wr}_{p,1}} \\
\langle \epsilon \triangleleft [p \mapsto 1] \rangle (!q) & \| & \langle \epsilon \rangle (q := 1\,;\,(!p)) & \xrightarrow[t_1]{\mathsf{wr}_{q,1}} \\
\langle \epsilon \triangleleft [p \mapsto 1] \rangle (!q) & \| & \langle \epsilon \triangleleft [q \mapsto 1] \rangle (!p) & \xrightarrow[t_0]{\mathsf{rd}_{q,0}} \\
\langle \epsilon \triangleleft [p \mapsto 1] \rangle (0) & \| & \langle \epsilon \triangleleft [q \mapsto 1] \rangle (!p) & \xrightarrow[t_1]{\mathsf{rd}_{p,0}} \\
\langle \epsilon \triangleleft [p \mapsto 1] \rangle (0) & \| & \langle \epsilon \triangleleft [q \mapsto 1] \rangle (0) & \xrightarrow[t_0]{\mathsf{bu}_{p,1}} \\
\langle \epsilon \rangle (0) & \| & \langle \epsilon \triangleleft [q \mapsto 1] \rangle (0) & \xrightarrow[t_q]{\mathsf{bu}_{q,1}} \\
\langle \epsilon \rangle (0) & \| & \langle \epsilon \rangle (0) &
\end{array}$$

It is not hard to see that if we add the appropriate barriers the example above becomes sequentially consistent. Its modified version is then:

Example 4.1.

$$\begin{bmatrix} p := 1\,; \\ \langle \mathtt{wr}|\mathtt{rd}\rangle\,; \\ r_0 := (!q) \end{bmatrix} \quad \| \quad \begin{bmatrix} q := 1\,; \\ \langle \mathtt{wr}|\mathtt{rd}\rangle\,; \\ r_1 := (!p) \end{bmatrix}$$

And omitting all the details, one can see that if we consider a computation like the one above we will reach a configuration of the form:

$$\langle \epsilon \lhd [p \mapsto 1] \lhd [\mathtt{wr}]\rangle(!q) \quad \| \quad \langle \epsilon \lhd [q \mapsto 1] \lhd [\mathtt{wr}]\rangle(!p)$$

from which the only possibility is to update one of the pending buffers, since the rule for reading demands that there be no `wr` barriers in $(\epsilon \lhd [p \mapsto 1] \lhd [\mathtt{wr}])(q)$ and $(\epsilon \lhd [q \mapsto 1] \lhd [\mathtt{wr}])(p)$, a condition that does not hold.

4.4 Speculative Semantics for TSO, PSO and RMO

Let us now see how can we formalize the TSO and PSO memory models by means of the speculative framework of Chapter 3. Of course, we need to extend the contexts to include the speculative ones, as well as increment the redexes (cf. Chapter 3):

$$\begin{array}{lcll} r & ::= & \ldots \mid (\lambda x e_0 e_1) & \textit{redexes} \\ \mathbf{\Sigma} & ::= & [] \mid (v\mathbf{\Sigma}) \mid (\lambda v \mathbf{\Sigma} e) \mid (\lambda v^{?} \mathbf{\Sigma} e) & \textit{speculation contexts} \end{array}$$

As in the semantics of Chapter 3 we will label the transitions of expressions not only with the action but also with their occurrence in the expression. We will use indeed the same set of occurrences:

$$\begin{array}{rcl} \mathcal{O}cc & = & \{(v[])\} \\ \mathcal{SO}cc & = & \mathcal{O}cc \cup \{(\lambda\_[]\_)\} \end{array}$$

where the occurrence corresponds to the place of the hole ([]) in the speculation context as given by:

$$\begin{array}{rcl} @[] & = & \epsilon \\ @(v\mathbf{\Sigma}) & = & (\_[]) \cdot @\mathbf{\Sigma} \\ @(\lambda x \mathbf{\Sigma} e) & = & (\lambda\_[]\_) \cdot @\mathbf{\Sigma} \\ @(\lambda v^{?} \mathbf{\Sigma} e) & = & (\lambda\_[]\_) \cdot @\mathbf{\Sigma} \end{array}$$

We shall not repeat the semantics of expressions, since it is almost the same as the one of Figure 4.3 with evaluation contexts replaced by speculation contexts, and every transition labeled with its corresponding occurrence. Thus, for instance the rule for reading from the memory is:

$$\mathbf{\Sigma}[(!\,p)] \xrightarrow[@\mathbf{\Sigma}]{\mathsf{rd}_{p,v}} \mathbf{\Sigma}[v]$$

with the others following the same pattern.

To define the global semantics of speculations we remove the write buffer components from the configurations. Then the configurations are simply:

$$C = (S, (t, e) \| T)$$

We can now define the small step semantics for configurations as we did previously. Notice that we annotate the transitions, not only with the occurrence of the redex, but also the thread identifier cf. Chapter 3:

$$\frac{e \xrightarrow[o]{a} e'}{(S, (t, e) \| T) \xrightarrow[t,o]{a} (S', (t, e') \| T)} \ (*)$$

where S' is determined by the action a being produced as follows:

$$(*) \begin{cases} a \in \left\{ \begin{array}{c} \beta, \swarrow, \searrow, \mathsf{cas}^{\circ}_{p,ff} \\ \mathtt{wr}, \mathtt{ww} \end{array} \right\} & \Rightarrow \quad S' = S \\ a \in \{\beta_v, \mathsf{rd}^{\circ}_{p,v}\} & \Rightarrow \quad \mathsf{FRef}(v) \subseteq \mathsf{dom}(S) \\ a = \nu_{p,v} & \Rightarrow \quad p \notin \mathsf{dom}(S) \ \& \ S' = S \cup \{p \mapsto v\} \\ a = \mathsf{rd}_{p,v} & \Rightarrow \quad v = S(p) \\ a = \mathsf{wr}_{p,v} & \Rightarrow \quad p \in \mathsf{dom}(S) \ \& \ S' = S\{p \leftarrow v\} \\ a = \mathsf{cas}_{p,tt} & \Rightarrow \quad S(p) = ff \ \& \ S' = S\{p \leftarrow tt\} \\ a = \mathsf{cas}_{p,ff} & \Rightarrow \quad S(p) = tt \\ a = \mathsf{cas}^{\circ}_{p,tt} & \Rightarrow \quad S' = S\{p \leftarrow tt\} \end{cases}$$

Notice at this point that the $\mathsf{rd}^{\circ}_{p,v}$ action returns a value (v) which is possibly different from the one in the store for the reference p. This is also the case for the $\mathsf{cas}^{\circ}_{p,v}$ action that predicts the value of the reference p. These actions will be used to model the effects of write buffers as the ones of Chapter 2, but the speculative semantics of these actions has to be refined, as regards the validity criterion, in order to correspond exactly to the write buffers of that chapter. In fact, pending writes on the buffers are updated only once and when they happen to be updated their value is globally visible through the store. This means for example that if a thread observes (by a read on the store) a write performed by other thread, then we are certain that the write must be already in the memory and the writing thread cannot perform further reads on the buffer (i.e. $\mathsf{rd}^{\circ}_{p,v}$ actions) with that value. This kind of reasoning (which is of a global nature) was not considered for the read-own actions of the previous chapter. We will soon see that the *validity* requirement will impose that the value returned be the last one written by the same thread, among other constraints which guarantee the exact correspondence with the behaviors of buffers.

4.4.1 Validity

A crucial aspect of the framework of speculation is the *validity* condition. In the case of the memory models of Sparc, that allow for write buffering, this condition is not trivial. We anticipated that the actions $\mathsf{rd}^{\circ}_{p,v}$ and $\mathsf{cas}^{\circ}_{p,v}$ were introduced to model, by means of speculations, the effects of write buffers. To that end, we will require in the validity condition for our calculus, that these

read and compare-and-swap actions do actually see the last written value to that reference (in this case p) by the thread performing the read. This is similar to the treatment of the $\mathsf{rd}^{\circ}_{p,v}$ and $\mathsf{cas}^{\circ}_{p,v}$ in Chapter 3, however here the validity condition will include global constraints, something we did not consider thus far. This is required because we want to faithfully model the memory models of Sparc, which allow for write-buffering. Moreover, we want to formally prove the correspondence of the formalizations by means of write-buffers and speculations of PSO and TSO. Therefore, we must be consistent with respect to the semantics of buffers, and for instance the fact that a thread reads a value produced by a second thread means that no subsequent read of the same reference by the second thread can be fulfilled by the buffers, and this fact can only be detected when considering the global computation. Let us see a simple example to illustrate this point (where we omit occurrences):

$$
\begin{array}{l}
(\{p \mapsto \mathit{ff}\}, (t_0, p := \mathit{tt}\,;(!p)\,;(!p)) \| (t_1, (!p))) \xrightarrow[t_0]{\mathsf{wr}_{p,\mathit{tt}}} \\
\qquad (\{p \mapsto \mathit{tt}\}, (t_0, (!p)\,;(!p)) \| (t_1, (!p))) \xrightarrow[t_0]{\mathsf{rd}^{\circ}_{p,\mathit{tt}}} \xrightarrow[t_0]{*} \\
\qquad\qquad (\{p \mapsto \mathit{tt}\}, (t_0, (!p)) \| (t_1, (!p))) \xrightarrow[t_1]{\mathsf{rd}_{p,\mathit{tt}}} \\
\qquad\qquad\qquad (\{p \mapsto \mathit{tt}\}, (t_0, (!p)) \| (t_1, \mathit{tt}))
\end{array}
$$

Here we see that the final read of thread t_0 (the only remaining redex) cannot obtain its value speculatively (or read its own write) since we know by the read of thread t_1 that the write has already been made globally visible. Otherwise thread t_1 could not have seen a value tt for its read on p. Actually, in the semantics with write-buffers we know that between the $\mathsf{rd}^{\circ}_{p,\mathit{tt}}$ action of thread t_0 and the $\mathsf{rd}_{p,\mathit{tt}}$ action of thread t_1 there must have been a buffer update. The corresponding computation in the semantics with write-buffers is:

$$
\begin{array}{l}
(\{p \mapsto \mathit{ff}\}, (\epsilon, t_0, p := \mathit{tt}\,;(!p)\,;(!p)) \| (\epsilon, t_1, (!p))) \xrightarrow[t_0]{\mathsf{wr}_{p,\mathit{tt}}} \\
\qquad (\{p \mapsto \mathit{ff}\}, (\epsilon \triangleleft [p \mapsto \mathit{tt}], t_0, (!p)\,;(!p)) \| (t_1, (!p))) \xrightarrow[t_0]{\mathsf{rd}^{\circ}_{p,\mathit{tt}}} \xrightarrow[t_0]{*} \\
\qquad\qquad (\{p \mapsto \mathit{ff}\}, (\epsilon \triangleleft [p \mapsto \mathit{tt}], t_0, (!p)) \| (t_1, (!p))) \xrightarrow[t_0]{\mathsf{bu}_{p,\mathit{tt}}} \\
\qquad\qquad\qquad (\{p \mapsto \mathit{tt}\}, (\epsilon, t_0, (!p)) \| (t_1, (!p))) \xrightarrow[t_1]{\mathsf{rd}_{p,\mathit{tt}}} \\
\qquad\qquad\qquad\qquad (\{p \mapsto \mathit{tt}\}, (\epsilon, t_0, (!p)) \| (\epsilon, t_1, \mathit{tt}))
\end{array}
$$

and here we can clearly see that according to the semantics with write-buffers we cannot perform a $\mathsf{rd}^{\circ}_{p,\mathit{tt}}$ action. We will shortly define when a write action is considered as "committed" in the speculative semantics, which corresponds to this kind of behavior.

Let us move on by defining some technical tools that will be used to define the validity condition. We need not repeat the definitions of residuals here, the reader is invited to consult Definition 3.14 to refresh this notion. Indeed since the definition of the reordering relation 3.18 is parametric on the dependency relation $\mathcal{D}$ we will just instantiate it with the corresponding dependency relations for TSO, PSO and RMO to get the corresponding reordering relations for each of these models.

Let us recall, from Chapter 3 the sets $\mathcal{MR}d_p$ and $\mathcal{MW}r_p$ of reads and writes *on the memory* for reference p:

$$\mathcal{MR}d_p \triangleq \{\mathsf{rd}_{p,v}, \mathsf{cas}_{p,v} \mid v \in \mathcal{V}al\}$$
$$\mathcal{MW}r_p \triangleq \{\mathsf{wr}_{p,v} \mid v \in \mathcal{V}al\} \cup \{\mathsf{cas}_{p,tt}, \mathsf{cas}^{\circ}_{p,tt}\}$$

that we use to define the conflict relation, that we recall here as well:

Definition 3.16 (Conflicting Actions)**.** *We define the conflict relation, denoted by #, to be the following binary relation on actions:*

$$\# \triangleq \bigcup_{p \in \mathcal{R}ef} (\mathcal{MW}r_p \times \mathcal{MW}r_p) \cup (\mathcal{MW}r_p \times \mathcal{MR}d_p) \cup (\mathcal{MR}d_p \times \mathcal{MW}r_p)$$

Notice that we explicitly have that speculative read actions are not conflicting with write actions on the same thread. Formally:

$$(\mathsf{wr}_{p,v}, \mathsf{rd}^{\circ}_{p,w}) \notin \#$$

A similar observation cannot be made for the $\mathsf{cas}^{\circ}_{p,ff}$ action, that is a compare-and-swap that does not succeed in modifying the location p, since we consider that this action always has write semantics (regardless of its result), and thus we would have a conflict with any other write actions on the same reference.

A fundamental ingredient of the formalization of the Sparc memory models by means of speculations is the dependency relation. But before defining the dependencies of Sparc let us recall, once more from Chapter 3, the notations $\mathcal{R}d_p$ and $\mathcal{W}r_p$ of read and write actions on location p (not necessarily accessing the memory as opposed to $\mathcal{MR}d_p$ and $\mathcal{MW}r_p$):

$$\mathcal{R}d_p \triangleq \mathcal{MR}d_p \cup \{\mathsf{rd}^{\circ}_{p,v}, \mathsf{cas}^{\circ}_{p,v} \mid v \in \mathcal{V}al\}$$
$$\mathcal{W}r_p \triangleq \mathcal{MW}r_p \cup \{\mathsf{cas}_{p,ff}, \mathsf{cas}^{\circ}_{p,ff}\}$$

and their obvious generalizations:

$$\mathcal{R}d \triangleq \bigcup_{p \in \mathcal{R}ef} \mathcal{R}d_p \qquad \text{and} \qquad \mathcal{W}r \triangleq \bigcup_{p \in \mathcal{R}ef} \mathcal{W}r_p$$

Let us concentrate now on the dependencies induced by barriers. We define by $\ltimes^{TSO}$ the set of dependencies that barriers impose on λ-TSO, and by $\ltimes^{PSO}$ and $\ltimes^{RMO}$ the ones for λ-PSO and λ-RMO respectively.

Definition 4.2 (Barrier Dependencies: TSO, PSO and RMO)**.** *We define the barrier dependencies relations for λ-TSO, λ-PSO and λ-RMO as:*

$$\begin{aligned}
\ltimes^{TSO} &\triangleq (\mathcal{W}r \times \{\mathtt{wr}\}) \cup (\{\mathtt{wr}\} \times \mathcal{R}d) \\
\ltimes^{PSO} &\triangleq \ltimes^{TSO} \cup (\mathcal{W}r \times \{\mathtt{ww}\}) \cup (\{\mathtt{ww}\} \times \mathcal{W}r) \\
\ltimes^{RMO} &\triangleq \ltimes^{PSO} \cup (\mathcal{R}d \times \{\mathtt{rw}, \mathtt{rr}\}) \cup (\{\mathtt{rw}\} \times \mathcal{W}r) \cup (\{\mathtt{rr}\} \times \mathcal{R}d)
\end{aligned}$$

The $\ltimes^{RMO}$ relation will also be named $\ltimes^{\mathcal{B}ar}$, since it includes all possible barriers of the language λ-barrier

As the reader might expect, to characterize the different memory models of Sparc we have to simply specify the appropriate dependency relations. We will denote by $\mathcal{D}^{TSO}$ the *actual* dependency relation of λ-TSO, and similarly for $\mathcal{D}^{PSO}$ and $\mathcal{D}^{RMO}$, that is:

Definition 4.3 (Sparc Dependencies: TSO, PSO and RMO)**.** *We characterize the RMO, PSO and TSO memory models by the following dependency relations:*

$$\begin{array}{lcl} \mathcal{D}^{RMO} & \triangleq & \# \cup \ltimes^{RMO} \\ \mathcal{D}^{PSO} & \triangleq & \# \cup \ltimes^{PSO} \cup (\mathcal{R}d \times \mathcal{R}d) \cup (\mathcal{R}d \times \mathcal{W}r) \\ \mathcal{D}^{TSO} & \triangleq & \# \cup \ltimes^{TSO} \cup (\mathcal{R}d \times \mathcal{R}d) \cup (\mathcal{R}d \times \mathcal{W}r) \cup (\mathcal{W}r \times \mathcal{W}r) \end{array}$$

Now we can provide the final, and perhaps the most important, notion we needed to establish a concrete formalization of the semantics of the Sparc memory models: that is the *validity* condition. Indeed, as in the previous chapter, it is only in this definition that the speculative semantics of $\mathsf{rd}^{\circ}_{p,v}$ and $\mathsf{cas}^{\circ}_{p,v}$ actions becomes clear. We consider a speculation as *valid* if there is an equivalent *normal* computation, where speculated reads have to see the last write on the same reference.

Definition 4.4 (Speculation validity)**.** *We say a speculation σ is valid with respect to the dependency relation (or the memory model) $\mathcal{D}$ if there is a normal speculation σ' such that $\sigma \propto^{\mathcal{D}} \sigma'$ and if $\sigma' = \sigma'_0 \cdot \xrightarrow[o]{\mathsf{rd}^{\circ}_{p,v}} \cdot\, \sigma'_1$ then there exists σ''_0, σ'''_0 and o' such that $\sigma'_0 = \sigma''_0 \cdot \xrightarrow[o']{\mathsf{wr}_{p,v}} \cdot\, \sigma'''_0$ where σ'''_0 contains no $\mathsf{wr}_{p,w}$ action for any $w \in \mathcal{V}al$. We will call the event $[\sigma''_0, (\mathsf{wr}_{p,v}, o')]$ the matching write of the event $[\sigma'_0, (\mathsf{rd}^{\circ}_{p,v}, o)]$, and we shall denote it by $\mathsf{match}\big([\sigma'_0, (\mathsf{rd}^{\circ}_{p,v}, o)]\big)$.*

Notice that, in the definition, the constraints imposed by barriers (which must be included in the dependency relation $\mathcal{D}$) are incorporated by the condition that requires the existence of a normal reordered computation. We will actually consider the definition of $\mathsf{match}(\alpha)$ up to step equivalence ($\sim$). Thus we will write $\mathsf{match}(\alpha_0) \approx \alpha_1$ to represent that given α_0 and α_1 two events in $\mathsf{Step}(\sigma)$, with σ a valid speculation justified by the normal speculation σ' (i.e. $\sigma \propto \sigma'$), and $\alpha'_0, \alpha'_1 \in \mathsf{Step}(\sigma')$ with $\alpha_0 \sim \alpha'_0$ and $\alpha_1 \sim \alpha'_1$ then $\mathsf{match}(\alpha'_0) = \alpha'_1$. Notice as well that the validity we consider here is simpler than the one presented in 3.20 (Chapter 3), since it does not impose constraints regarding barriers (or more generally dependencies) for $\mathsf{rd}^{\circ}_{p,v}$ actions. These constraints will be required later in the validity condition for speculative computations.

To faithfully characterize the behavior of write buffers with speculations we have to consider when a write is actually made visible to all the threads involved in the system. As we mentioned before this is easy to see in the semantics with buffers, since it corresponds exactly with the buffer update event ($\mathsf{bu}_{p,v}$). However, with speculations this concept is more elusive. The following definition captures the minimal conditions that imply that a write has been made visible to the entire thread system in the speculative semantics. Obviously, this definition has to be provided for speculative computations rather than speculations alone, since it is of a global nature.

Definition 4.5 (Committed write)**.** *Given a speculative computation $\gamma = \gamma_0 \cdot \xrightarrow[t,o]{\mathsf{wr}_{p,v}} \cdot\, \gamma_1$ we say that $[\gamma_0|_t, (\mathsf{wr}_{p,v}, o)]$ is* committed *in γ if there*

are γ_1', γ_1'', t', o' *and* w *such that* $\gamma_1 = \gamma_1' \cdot \xrightarrow[t',o']{\mathsf{rd}_{p,w}} \cdot \gamma_1''$ *, or such that* $\gamma_1 = \gamma_1' \cdot \xrightarrow[t,o']{\mathsf{wr}_{q,w}} \cdot \gamma_1''$ *with* $[\gamma_0|_t, (\mathsf{wr}_{p,v}, o)] \prec_{\gamma|_t} [\gamma_0 \cdot \xrightarrow[t,o]{\mathsf{wr}_{p,v}} \cdot \gamma_1'|_t, (\mathsf{wr}_{q,w}, o')]$ *and* $[\gamma_0 \cdot \xrightarrow[t,o]{\mathsf{wr}_{p,v}} \cdot \gamma_1', (\mathsf{wr}_{p,q}, o')]$ *is committed in* γ.

The consequence, regarding writes on different references, is that if a write on q is dependent on a previous write of p (typically because we have an intermediate $\langle \mathtt{wr}|\mathtt{wr}\rangle$ instruction), and the write of q is committed in γ, then so it is the one of p. To see why we require this condition consider the following thread system where we depict only threads:

$$\begin{bmatrix} p := tt\,; \\ \langle \mathtt{wr}|\mathtt{wr}\rangle\,; \\ q := tt\,; \\ (!p) \end{bmatrix} \;\|\; \Big[(!q)\Big] \xrightarrow[t_0]{\mathsf{wr}_{p,tt}} \begin{bmatrix} \langle \mathtt{wr}|\mathtt{wr}\rangle\,; \\ q := tt\,; \\ (!p) \end{bmatrix} \;\|\; \Big[(!q)\Big] \xrightarrow[t_0]{\mathtt{ww}}$$

$$\begin{bmatrix} q := tt\,; \\ (!p) \end{bmatrix} \;\|\; \Big[(!q)\Big] \xrightarrow[t_0]{\mathsf{wr}_{q,tt}} \Big[(!p)\Big] \;\|\; \Big[(!q)\Big]$$

$$\xrightarrow[t_1]{\mathsf{rd}_{q,tt}} \Big[(!p)\Big] \;\|\; \begin{bmatrix} tt \end{bmatrix}$$

It is clear that the final read of p cannot be a $\mathsf{rd}^{\circ}_{p,v}$ (that is a read of an uncommitted write), since the write of q has already been made globally visible, and there is a $\langle \mathtt{wr}|\mathtt{wr}\rangle$ between the write of p and the one of q. Therefore the write of p must have also been made globally visible. In the case of the semantic with write-buffers it is clear that the last read of p must obtain its value from the memory.

Once we have the definition of committed write we can finally define the validity condition of speculative computations.

Definition 4.6 (Valid speculative computation)**.** *A speculative computation* γ *is* valid *iff for every thread* t *we have that* $\gamma|_t$ *is a valid speculation; and additionally, if* $\gamma = \gamma' \cdot \xrightarrow[t,o']{\mathsf{rd}^{\circ}_{p,v}} \cdot \gamma_2$ *where* $\gamma' = \gamma_0 \cdot \xrightarrow[t,o]{\mathsf{wr}_{p,v}} \cdot \gamma_1$*, and* $\mathsf{match}([\gamma'|_t, (\mathsf{rd}^{\circ}_{p,v}, o')]) \approx [\gamma_0|_t, (\mathsf{wr}_{p,v}, o)]$ *then* $[\gamma_0|_t, (\mathsf{wr}_{p,v}, o)]$ *is not committed in* γ'.

In this definition we have intentionally disregarded compare-and-swap actions. If we were to include them a condition similar to the one for $\mathsf{rd}^{\circ}_{p,v}$ should be required for $\mathsf{cas}^{\circ}_{p,v}$, and the matching write could in general be a $\mathsf{cas}_{p,tt}$ or $\mathsf{cas}^{\circ}_{p,tt}$ action.

To illustrate the definition of valid speculative computations let us reconsider some of the examples presented in Figure 4.1. Let us focus mainly on the ones considering RMO, since the others are similar, and we have already considered them in the formalization with write buffers. The example that shows the capability of RMO to reorder reads w.r.t. subsequent writes is the following (cf. Example 1.4).

$$\begin{bmatrix} r_0 := (!q)\,; \\ p := 1 \end{bmatrix} \;\|\; \begin{bmatrix} r_1 := (!p)\,; \\ q := 1 \end{bmatrix}$$

One wonders here whether $r_0 = r_1 = 1$ is a possible final result assuming that initially $p = q = 0$. It is not hard to see that by means of speculations we can obtain the following computation, where we omit the occurrences, and chose the name t_0 for the thread on the left and t_1 for the one on the right.

$$\begin{array}{rcl} r_0 := (!q)\,;\ p := 1 & \| & r_1 := (!p)\,;\ q := 1 \quad \xrightarrow[t_0]{\mathsf{wr}_{q,1}} \\ r_0 := (!q)\,;\ () & \| & r_1 := (!p)\,;\ q := 1 \quad \xrightarrow[t_1]{\mathsf{wr}_{p,1}} \\ r_0 := (!q)\,;\ () & \| & r_1 := (!p)\,;\ () \quad \xrightarrow[t_0]{\mathsf{rd}_{q,1}} \\ r_0 := 1\,;\ () & \| & r_1 := (!p)\,;\ () \quad \xrightarrow[t_1]{\mathsf{rd}_{q,1}} \\ r_0 := 1\,;\ () & \| & r_1 := 1\,;\ () \end{array}$$

From here the result is obvious. However, to be certain that this computation represents an RMO one we have to verify the validity condition. Since there are no $\mathsf{rd}^{\circ}_{p,v}$ or $\mathsf{cas}^{\circ}_{p,v}$ actions, the only thing to do is to reorder the speculations of t_0 and t_1 to reach a normal speculations. But it is trivial to see that according to the dependency relation $\mathcal{D}^{RMO}$ we have for t_0 that:

$$\xrightarrow{\mathsf{wr}_{q,1}} \cdot \xrightarrow{\mathsf{rd}_{p,1}} \cdot \ \ldots \ \propto^{\mathcal{D}^{RMO}} \ \xrightarrow{\mathsf{rd}_{p,1}} \cdot \ \ldots \ \cdot \xrightarrow{\mathsf{wr}_{q,1}}$$

and similarly for t_1.

If, on the other hand we add the required barriers to make this program sequentially consistent as follows:

$$\begin{bmatrix} r_0 := (!q)\,; \\ \langle \mathtt{rd}|\mathtt{wr}\rangle\,; \\ p := 1 \end{bmatrix} \quad \| \quad \begin{bmatrix} r_1 := (!p)\,; \\ \langle \mathtt{rd}|\mathtt{wr}\rangle\,; \\ q := 1 \end{bmatrix}$$

we see that the normal computation of thread t_0 should have the following shape:

$$\xrightarrow{\mathsf{rd}_{p,1}} \cdot \ \ldots \ \cdot \xrightarrow{\mathtt{rw}} \cdot \xrightarrow{\mathsf{wr}_{q,1}}$$

and since $(\mathsf{rd}_{p,1}\ \mathcal{D}^{RMO}\ \mathtt{rw})$ and $(\mathtt{rw}\ \mathcal{D}^{RMO}\ \mathsf{wr}_{q,1})$ there is no possible reordering that could render the result we had in the previous program. Therefore, the barriers render the program sequentially consistent.

The other example that is allowed by RMO is the one involving the reordering of reads (Example 2.40),

$$\left[\begin{array}{l} p := 1\,; \\ \langle \mathtt{wr}|\mathtt{wr}\rangle\,; \\ q := 1 \end{array}\right] \quad \| \quad \left[\begin{array}{l} r_0 := (!q)\,; \\ r_1 := (!p) \end{array}\right]$$

where we explicitly add a barrier in the first thread, to disallow the reordering of writes, which could be another source to obtain $r_0 = 1$ and $r_1 = 0$ as the final result. We shall leave the checking of this example, which is very similar to the previous one, as an exercise to the reader.

Perhaps a more challenging example involves checking that behaviors allowed by write buffers are also allowed by the speculative formalization of these memory models. To that end we consider the Example 2.3 in the λ-PSO case,

where once more, we assume that initially $p = q = 0$. We recall here that λ-PSO does not allow the reordering of reads with subsequent actions.

$$\left[\begin{array}{l} p := 1\,; \\ r_0 := (!p)\,; \\ r_1 := (!q) \end{array}\right] \quad \| \quad \left[\begin{array}{l} q := 1\,; \\ r_2 := (!q)\,; \\ r_3 := (!p) \end{array}\right]$$

Let us discuss first this program from the point of view of the reordering of instructions alone. In the left thread since at the beginning we have a write on p followed by a read on p these events cannot be reordered; and since then we have two reads (on p and q), which are not reordered in PSO, these events cannot be reordered either. One might think then that this program is sequentially consistent. However, if we consider this program from the point of view of write buffers, we immediately see that we could start by writing in both threads (that is buffering the writes), then reading the "own" pending writes in both threads (which does not require updating the writes), and finally reading the reference that has not yet been updated to the memory, obtaining the initial value 0.

As we said before this is exactly the purpose of $\mathsf{rd}^{\circ}_{p,v}$ actions in the semantics of speculations. If instead of performing reads in the memory we speculate them, we can have the following computation:

$$\begin{array}{rcl} p := 1\,;\ r_0 := (!p)\,;\ r_1 := (!q) & \| & q := 1\,;\ r_2 := (!q)\,;\ r_3 := (!p) \xrightarrow[t_0]{\mathsf{rd}^{\circ}_{p,1}} \\ p := 1\,;\ r_0 := 1\,;\ r_1 := (!q) & \| & q := 1\,;\ r_2 := (!q)\,;\ r_3 := (!p) \xrightarrow[t_1]{\mathsf{rd}^{\circ}_{q,1}} \\ p := 1\,;\ r_0 := 1\,;\ r_1 := (!q) & \| & q := 1\,;\ r_2 := 1\,;\ r_3 := (!p) \xrightarrow[t_0]{\mathsf{rd}_{q,0}} \\ p := 1\,;\ r_0 := 1\,;\ r_1 := 0 & \| & q := 1\,;\ r_2 := 1\,;\ r_3 := (!p) \xrightarrow[t_1]{\mathsf{rd}_{p,0}} \\ p := 1\,;\ r_0 := 1\,;\ r_1 := 0 & \| & q := 1\,;\ r_2 := 1\,;\ r_3 := 0 \xrightarrow[t_0]{\mathsf{wr}_{p,1}} \\ \langle\rangle\,;\ r_0 := 1\,;\ r_1 := 0 & \| & q := 1\,;\ r_2 := 1\,;\ r_3 := 0 \xrightarrow[t_1]{\mathsf{wr}_{q,1}} \\ \langle\rangle\,;\ r_0 := 1\,;\ r_1 := 0 & \| & \langle\rangle\,;\ r_2 := 1\,;\ r_3 := 0 \end{array}$$

And from this point the result $r_0 = r_2 = 1$ and $r_1 = r_3 = 0$ is obvious. We still have to consider the validity of the example. Indeed, since we explicitly have that $\neg(\mathsf{wr}_{p,1}\ \mathcal{D}^{PSO}\ \mathsf{rd}^{\circ}_{p,1})$ and $\neg(\mathsf{wr}_{p,1}\ \mathcal{D}^{PSO}\ \mathsf{rd}_{q,0})$, we have that for t_0:

$$\xrightarrow{\mathsf{rd}^{\circ}_{p,1}} \cdot \xrightarrow{\mathsf{rd}_{q,0}} \cdot \xrightarrow{\mathsf{wr}_{p,1}} \propto^{\mathcal{D}^{PSO}} \xrightarrow{\mathsf{wr}_{p,1}} \cdot \xrightarrow{\mathsf{rd}^{\circ}_{p,1}} \cdot \xrightarrow{\mathsf{rd}_{q,0}}$$

where we can see that the value obtained by the $\mathsf{rd}^{\circ}_{p,v}$ action corresponds to the last write on that reference in the speculation, and there are no intermediate `wr` actions in the normal computation. To disallow this behavior it suffices to add a $\langle \mathtt{wr}|\mathtt{rd}\rangle$ construct between the write and the read of the same reference (cf. Example 3.19), as follows:

$$\left[\begin{array}{l} p := 1\,; \\ \langle \mathtt{wr}|\mathtt{rd}\rangle\,; \\ r_0 := (!p)\,; \\ r_1 := (!q) \end{array}\right] \quad \| \quad \left[\begin{array}{l} q := 1\,; \\ \langle \mathtt{wr}|\mathtt{rd}\rangle\,; \\ r_2 := (!q)\,; \\ r_3 := (!p) \end{array}\right]$$

The last example that we considered illustrates the need of a complex validity definition to allow the behaviors induced by write buffers. Unfortunately

describing by means of full computations the need of a global validity definition is rather uncomfortable to present on paper. However, we can provide an example program for PSO.

Example 4.7.

$$\left[\begin{array}{l} p := 1\,; \\ r_0 := (!q)\,; \\ r_1 := (!p) \end{array}\right] \quad \| \quad \left[\begin{array}{l} \mathsf{let}\ x = (!p)\ \mathsf{in}\ (\\ \qquad q := x\ ) \end{array}\right]$$

In this example, if $r_0 = 1$ then the read of p is not a case of "read own", that is, it is not a $\mathsf{rd}^{\circ}_{p,1}$ action. This is due to the fact that since the read of q in the left thread sees the write of q on the right thread, it must be the case that the write of p in the left thread was gobally performed (or *committed*) at the time of that write, and since the read of q in the thread of the left cannot be reordered with the following read of p (as per $\mathcal{D}^{PSO}$) it must be the case that the write of p is globally performed, and thus the read cannot produce a $\mathsf{rd}^{\circ}_{p,1}$ action, but has to produce a normal $\mathsf{rd}_{p,1}$ actions instead. In this particular example, that we include because is not too complex, the final result would not be modified if the read could be satisfied with a $\mathsf{rd}^{\circ}_{p,1}$ action. However, there are examples where this choice can lead to different results, but they are complex, and therefore we do not include them here.

4.5 A Correspondence Result

In this section we prove that the two formalizations of the semantics of PSO and TSO actually describe the same semantics. We will prove that PSO executions in the semantics of write buffers correspond to executions in the semantics with speculations and vice versa. In particular, the proof also applies in a trivial way to the formalizations of TSO, and therefore we will mainly focus on PSO, which is more general. Since we do not have an "architectural" formalization of the operational semantics of RMO (similar to the write-buffer semantics), we can not draw a similar result for RMO.

To prove the correspondence of both formalizations of PSO we need means to compare their executions. We have to show how a computation in each of these calculus corresponds to a computation in the other. To that end, we introduce a third calculus, which we call the *merge-calculus*, that includes both write buffering and speculations. This calculus could be of interest in itself but here we will only use it as a technical tool to prove our correspondence result.

The semantics of single expressions of the merge-calculus is the speculative one, that is, identical to that of the speculative formalization of Figure 4.3 with $\mathbf{E}$ replaced by $\mathbf{\Sigma}$ and transitions annotated with the occurrences. On the other hand, its global semantics is almost that of the write-buffer semantics given in Figure 4.4, except that transitions are allowed to happen at speculative occurrences, and we need to annotate the occurrence of actions. In addition, the rule for reading in a buffer ($\mathsf{rd}^{\circ}_{p,v}$) will be replaced by its speculative version, where one does not verify the contents of the buffers. Actually the only rule

that changes from Figure 4.4 becomes:

$$\frac{e \xrightarrow[o]{a} e'}{(S,(B,t,e)\|T) \xrightarrow[t,o]{a} (S',(B',t,e')\|T)}\ (*)$$

where

$$(*) = \begin{cases} \dots \\ a = \mathsf{rd}^{\circ}_{p,v} \quad \Rightarrow \quad S' = S \ \&\ B' = B \\ \dots \end{cases}$$

with ... standing for the conditions $(*)$ of the rules of Figure 4.4.

The basis of our proof will be the reordering relation, which refines the equivalence by permutations so many times discussed in this thesis. We will actually prove that provided with a computation in the source semantics (either with buffers, or speculations) there is a trivially corresponding merge-calculus computations. With this merge-calculus computation we will show that we can reorder the events to reach a merge computation that corresponds exactly to one of the target formalism. Of course, we will prove that the resulting speculations are related by reordering, and in particular coincide in their initial and final configuration.

To define the reordering relation for the merge-calculus semantics we have to redefine the conflict and dependencies relation. Notice that in the merge-calculus different threads do not conflict upon writing actions ($\mathsf{wr}_{p,v}$) because the effects of writing are local to the buffer of the thread performing the write. However, updating the buffers has a global effect. Thus we modify the global conflict relation ($\#$) to match this observation (recall that we are not considering compare-and-swap actions here):

Definition 4.8. *(Merge conflict)*

$$\#^{MG} = \bigcup_{p\in\mathcal{R}ef,\ v,w\in\mathcal{V}al} \{(\mathsf{bu}_{p,v},\mathsf{bu}_{p,w}),(\mathsf{rd}_{p,v},\mathsf{bu}_{p,w}),(\mathsf{bu}_{p,v},\mathsf{rd}_{p,w})\}$$

On the other hand we adopt the dependency relation we defined for PSO in the speculative case with the obvious restrictions regarding compare-and-swap actions:

$$\mathcal{D}^{MG} \triangleq \mathcal{D}^{PSO}$$

Notice that in the definition of $\mathcal{D}^{PSO}$ we considered the standard definition of conflict $\#$ (with writes instead of buffer update actions). This is still the case here, and moreover $\mathsf{bu}_{p,v}$ actions do not appear in the single expression semantics.

Replacing the dependency relation above for the one of λ-TSO (that is $\mathcal{D}^{TSO}$) renders the correspondence result for λ-TSO.

Once more, we base our results on the reordering relation, which is a refinement of the permutations equivalence [Berry and Lévy, 1979]. Then, we need to state the *asynchrony* lemma for the merge-calculus as we did in 3.15.

Lemma 4.9. *(Merge Asynchrony) If* $e_0 \xrightarrow[o_0]{a_0} e \xrightarrow[o_1']{a_1} e_1$ *with* $o_1' \equiv o_1/_{e_0}(a_0,o_0)$ *and* $o_0' \equiv o_0/_{e_0}(a_1,o_1)$ *then there exists* e' *such that* $e_0 \xrightarrow[o_1]{a_1} e' \xrightarrow[o_0']{a_0} e_1$.

Proof. The proof is almost identical to that of 3.15. □

The reordering relation we will consider here is the instantiation of the definition 3.18 with the dependency relation of the merge-calculus, in notation $\propto^{\ltimes^{MG}}$. In fact, since we will only consider this reordering relation in the rest of this chapter the parameter will be omitted, so we will denote this reordering relation simply by $\propto^{MG}$.

4.5.1 Preliminaries: Relevant Moves

Before focusing on the proof of the correspondence, we introduce some definitions that will simplify the following developments. In particular, one can observe that from the memory model point of view, actions of the calculus other than reads, writes, and barriers are irrelevant. Indeed, they are not involved in the laws of the model and in fact their semantics is independent of the memory model restrictions. In essence, the only way in which these actions matter, is in sequencing the execution by means of redex creations.

We will consider executions where irrelevant actions have been separated from the memory related (more precisely memory model related) actions. We can do that because we are considering an ANF calculus, where redex creation can happen at any time, by means of the β_v reduction. One can observe then, that for any computation of the merge calculus, or the speculative one for that matter, redex creations by means of β_v reductions, reference creations ($\nu_{p,v}$) and conditional branchings ($\swarrow$ and $\searrow$) can be pushed to the beginning of the computation. In some sense, all redexes are created first, and all β reductions, whose only purpose is to match the predicted argument with the actual one, can be pushed to the end of the computation. In between we find the relevant events. Thus all validations of argument predictions can be performed at the end of the execution. What is interesting about this transformation, is that we can then consider only memory model related actions without having to analyze all the possible combinations of cases. Let us make these intuitions more formal.

We start by defining the condition that states that a speculation has its memory model related actions isolated. In order to do so, let us adopt the following notation that represents the set of actions that happen in a speculation:

$$\mathsf{act}(\sigma) = \begin{cases} \emptyset & \text{if } \sigma = \varepsilon \\ \mathsf{act}(\sigma') \cup \{a\} & \text{if } \sigma = \xrightarrow[o]{a} \cdot \sigma' \end{cases}$$

We can now define the *Relevant Moves Normal Form* (RMNF) for merge-calculus speculations.

Definition 4.10 (Relevant Moves Normal Form)**.** *We say a speculation σ is in Relevant Moves Normal Form (RMNF) if there are σ_0, σ_1 and σ_2 such that $\sigma = \sigma_0 \cdot \sigma_1 \cdot \sigma_2$ with $\mathsf{act}(\sigma_0) \subseteq \{\beta_v, \nu_{p,v}, \swarrow, \searrow \mid v \in \mathcal{V}al, p \in \mathcal{R}ef\}$, also $\mathsf{act}(\sigma_1) \subseteq \mathcal{R}d \cup \mathcal{W}r \cup \mathcal{B}ar$ and finally $\sigma_2 \in \beta^*$.*

Notice in the above definition that the subspeculation σ_2 contains all, and only, the memory model related actions. This is exactly the purpose of the RMNF.

Now we show that any speculation can be transformed into RMNF.

Lemma 4.11. *Given* $e_0 \xrightarrow[o_0]{a_0} e \xrightarrow[o_1']{a_1} e_1$ *such that* $a_0 \notin \{\beta_v, \nu_{p,v}, \swarrow, \searrow\}$ *and* $a_1 \neq \beta$ *there exist* o_1, o_0' *and* e' *such that* $e_0 \xrightarrow[o_1]{a_1} e' \xrightarrow[o_0']{a_0} e_1$ *with* $o_0' \equiv o_0/_{e_0}(a_1, o_1)$ *and* $o_1' \equiv o_1/_{e_0}(a_0, o_0)$.

Proof. Notice that by the Asynchrony Lemma 4.9 we only need to prove that there exist o_0' and o_1 satisfying the required conditions. Let us proceed by case analysis on the relation between o_0 and o_1':

- Suppose first that $(o_0 \leq o_1')$. Then $o_1' = o_0 \cdot o''$ for some o'' and that exists $\mathbf{\Sigma}_0$ such that $e_0 = \mathbf{\Sigma}_0[r_0] \xrightarrow[o_0]{a_0} \mathbf{\Sigma}_0[\bar{e}]$ with $o_0 = @\mathbf{\Sigma}_0$. Moreover $\bar{e}$ contains a redex at o''. The only cases for a_0 such that e_0 produces a subexpression capable of containing a redex to be reduced in the next step are $a_0 \in \{\beta_v, \swarrow, \searrow, \beta\}$ by a simple analysis on the semantic rules. Notice that by the hypothesis the only case that remains to be analyzed is $a_0 = \beta$ in which case we have $r_0 = (\lambda v^? \bar{e} v)$ for some $v \in \mathcal{V}al$ and thus $o_1 = o_0 \cdot (\lambda\_[]\_) \cdot o''$ and $o_0' = o_0$ satisfy the conditions required by the lemma.
- if $(o_0 > o_1')$ then $o_0 = o_1' \cdot o''$. If $e_0@o_1' = r_1$ (recall that this notation has been introduced in page 72) is a redex then $r_1 = (\lambda v^? \bar{e} v)$ for some $\bar{e}$ and v (there is no other expression containing a redex that can be reduced in the previous step), in which case $a_1 = \beta$ contradicting the hypothesis. If $e_0@o_1'$ is not a redex then then $o_0 = o_1' \cdot (\_[])$ with again $a_1 = \beta$, a contradiction.
- if $(o_0 \not\leq o_1')$ and $(o_0 \not> o_1')$ we conclude simply with $o_0' = o_0$ and $o_1 = o_1'$.

□

Then it is not hard to see that any speculation can be reordered to obtain a RMNF equivalent speculation. In particular none of the actions in $\{\beta, \beta_v, \swarrow, \searrow\}$ is involved in the dependency relation, which justifies the following corollary.

Corollary 4.12 (Relevant Moves Normal Form)**.** *For every speculation* γ *there is a speculation* γ' *such that* $\gamma' \propto^{MG} \gamma$ *and* γ' *is in RMNF.*

Proof. The proof is trivial by reordering the actions from the left by means of the lemma 4.11. □

4.5.2 Global Relevant Moves Normal Form

In the subsequent results we will disregard the cases of actions that are "irrelevant". This will simplify the following proofs. Let us now consider the obvious extension of the dependency relation to speculations, denoted by $\sigma\ \mathcal{D}^{MG}\ a$, and meaning that there exists $a' \in \mathsf{act}(\sigma)$ such that $a'\ \mathcal{D}^{MG}\ a$. Then we can prove that steps can be reordered w.r.t. *independent* subspeculations.

Lemma 4.13 (Independent moves)**.** *Let* $\gamma = \sigma \cdot \sigma' \cdot \sigma''$ *be a RMNF computation such that* σ' *contains no* $\{\beta_v, \beta, \swarrow, \searrow\}$ *action and let* $\sigma' = \sigma_0 \cdot \xrightarrow[o]{a}$ *with* $\neg(\sigma_0 \mathcal{D}^{MG} a)$. *Then there exist* σ_0' *and* o' *such that* $\xrightarrow[o']{a} \cdot\ \sigma_0' \propto^{MG} \sigma'$ *and thus* $\sigma \cdot \xrightarrow[o']{a} \cdot\ \sigma_0' \ \cdot\ \sigma'' \propto^{MG} \gamma$.

Proof. Induction in the length of σ_0. We use the lemma 4.11 for the inductive case and conclude by the induction hypothesis. □

As we did in Chapter 2 we will consider buffers up to reordering of updates of different references. The equivalence of buffers is recalled in the following definition:

Definition 2.23 (Buffer equivalence)**.** *The buffers equivalence relation is the least equivalence $\equiv$ between buffers satisfying:*

$$\frac{p \neq q}{B_0 \lhd [p \leftarrow v] \lhd [q \leftarrow w] \lhd B_1 \equiv B_0 \lhd [q \leftarrow w] \lhd [p \leftarrow v] \lhd B_1}$$

An important consequence of adding buffers to the speculative semantics is that now there are creations of transitions induced by the write-buffers; for instance, a buffer update ($\mathsf{bu}_{p,v}$) action cannot happen if the buffer is empty, and a normal read $\mathsf{rd}_{p,v}$ cannot happen if there is a pending update on reference p in the buffer, in this last case it is only after the pending writes are committed into the memory that a normal read can proceed. Simply said, there are actions that are only enabled for buffers of a certain shape. The buffer-dependency relation is a relation of two consecutive semantic steps, that clearly depends on the originating configuration. The following definition captures that intuition:

Definition 4.14 (Buffer Dependency)**.** *Whenever we have $C \xrightarrow[t,o]{a} C' \xrightarrow[t,o']{a'} C''$ we say that a creates a' from C, which we denote by $(a,t) \triangleright_C (a',t)$, if the following conditions hold: $C = (S, (B,t,e) \parallel T)$ and*

$$\begin{cases} B \neq [b] \triangleright B' & \text{and } a' = \overline{b} \quad \text{or} \\ B(p) \notin (\mathtt{ww})^* & \text{and } a' = \mathsf{rd}_{p,v} \quad \text{or} \\ B(p) \neq (\mathtt{wr})^* \cdot v \cdot s' & \text{and } a' = \mathsf{bu}_{p,v} \end{cases}$$

We can now prove that whenever we have two consecutive events of the same thread in a speculative computation, such that the actions they produce are not dependent, and the events are not dependent through the buffers, then these events can happen in the reverse order in the computation, resulting in the same final configuration.

Lemma 4.15 (Global Reordering: Intrathread)**.** *Given a configuration C_0 with $C_0 = (S_0, (B_0, t, e_0) \| T_0)$ such that:*

i) $C_0 \xrightarrow[t,o_0]{a_0} C \xrightarrow[t,o_1]{a_1} C_1$, *and*
ii) $a_1 \neq \beta$ *or* $a_0 \notin \{\beta_v, \swarrow, \searrow \mid v \in \mathcal{V}al\}$, *and*
iii) $\neg(a_0 \triangleright_{C_0} a_1)$, *and*
iv) if both $a_0, a_1 \notin \{\mathsf{bu}_{p,v}, \overline{\mathtt{wr}}, \overline{\mathtt{ww}} \mid p \in \mathcal{R}ef, v \in \mathcal{V}al\}$ *then* $\neg a_0 \mathcal{D}^{MG} a_1$,

then there is C' such that:

$$C_0 \xrightarrow[t,o_1']{a_1} C' \xrightarrow[t,o_0']{a_0} C_1$$

Proof. Let us consider the possible cases for a_0 and a_1:

- If $a_0 = \beta$ we need to consider the following cases for a_1. If $a_1 \in \{\mathsf{bu}_{p,v}, \overline{\mathtt{wr}}, \overline{\mathtt{ww}} \mid p \in \mathcal{R}ef, v \in \mathcal{V}al\}$ we have the conclusion directly with $B' = B_1$ and $e' = e$. In the cases where $a_1 \in \mathcal{A}ct$ we have by hypothesis ii) and Lemma 4.11 that there is e' with $e_0 \xrightarrow[o_1']{a_1} e' \xrightarrow[o_0']{a_0} e_1 \propto^{MG} e_0 \xrightarrow[o_0]{a_0} e \xrightarrow[o_1]{a_1} e_1$ and clearly $S' = S_1$ and $B' = B_1$ since a_0 does not modify neither the store nor the buffers.
  Almost the same reasoning applies to all the cases for a_0 with $a_1 \in \{\beta_v, \swarrow, \searrow \mid v \in \mathcal{V}al\}$ which we will not develop in the sequel.
- If both $a_0, a_1 \notin \{\beta, \beta_v, \swarrow, \searrow, \mathsf{bu}_{p,v}, \overline{\mathtt{wr}}, \overline{\mathtt{ww}} \mid p \in \mathcal{R}ef, v \in \mathcal{V}al\}$ we simply apply the hypothesis iv) and the Local Asynchrony Lemma 4.9 to obtain the appropriate e'. Obviously the store does not change in any of these steps, and it is easy to verify that the resulting buffer is the same up to the buffer equivalence $\simeq$.
- If $a_0 = \mathsf{rd}_{p,v}$ we have, from the semantics, that $B(p) = (\mathtt{ww})^*$ and $S(p) = v$. Clearly in this case by hypothesis iii) we have $a_1 \notin \{\overline{\mathtt{wr}}, \mathsf{bu}_{p,w} \mid w \in \mathcal{V}al\}$ and thus the conclusion is very easy. Notice that the case where $a_1 = \mathsf{wr}_{p,w}$ has already been discarded in the previous case.
- If $a_0 = \mathsf{rd}^{\circ}_{p,v}$ the conclusion is immediate since this action does not depend nor modify in any way the buffers or the store.
- If $a_0 = \mathsf{wr}_{p,v}$ the conclusion is simple as well observing that if $a_1 = \mathsf{bu}_{p,w}$ then $B(p) = (\mathtt{wr})^* \cdot [p \mapsto w] \cdot s$ for some s; otherwise we would violate the hypothesis ii). Also it is clear that $a_1 \neq \overline{\mathtt{ww}}$. We obtain the conclusion easily.
- If $a_0 \in \{\mathtt{ww}, \mathtt{wr}\}$ we have the that, given the condition iii), the requirement for performing a_0 is such that the reordering is guaranteed. For instance if $a_0 = \mathtt{ww}$ and $a_1 = \overline{\mathtt{ww}}$ then $B = \mathtt{ww} \cdot s$ for some s. Similarly for $\mathtt{wr}$.
- If $a_0 = \mathsf{bu}_{p,v}$ then $a_1 \notin \{\overline{\mathtt{ww}}, \mathsf{bu}_{p,w}, \mathsf{rd}_{p,w} \mid w \in \mathcal{V}al\}$; and if $a_1 = \overline{\mathtt{wr}}$ by iii) we know that $B(p) = \mathtt{wr} \cdot s$ for some s, the conclusion is immediate in the remaining cases.
- If $a_0 = \overline{\mathtt{ww}}$ then $a_1 \notin \{\mathsf{bu}_{q,w}, \overline{\mathtt{wr}} \mid r \in \mathcal{R}ef,\ w \in \mathcal{V}al\}$. Again in this case the conclusion is direct. The same reasoning applies to $a_0 = \overline{\mathtt{wr}}$ where we know by iii) that $a_1 \neq \mathsf{rd}_{q,w}$ for all q and w, nor $a_1 = \overline{\mathtt{ww}}$.

□

And a similar, but simpler result can be derived for the case in which the events occur in different threads.

Lemma 4.16 (Global Reordering)**.** *Let* $C_0 = (S, (B_0, t_0, e_0) \| (B_1, t_1, e_1) \| T)$ *and* $C_0 \xrightarrow[t_0,o_0]{a_0} C \xrightarrow[t_1,o_1]{a_1} C_1$ *with* $t_0 \neq t_1$ *and* $\neg(a_0 \ \#^{MG} \ a_1)$*. Then there exists* C' *such that*

$$C_0 \xrightarrow[t_1,o_1]{a_1} C' \xrightarrow[t_0,o_0]{a_0} C_1$$

Proof. We proceed by case analysis on a_0 and a_1. We consider only the cases of actions that access the memory, the other being trivial (Notice that $\mathsf{wr}_{p,v}$ actions do not directly access the memory and thus are trivial too):

- $a_0 = \mathsf{rd}_{p,v}$. If:
  - $a_1 = \mathsf{rd}_{q,w}$ the conclusion is trivial, even if $p = q$ (with $w = v$).
  - $a_1 = \mathsf{bu}_{q,w}$. If $p = q$ then $a_0 \# a_1$ contradicting the hypothesis. In case $p \neq q$ the conclusion is immediate.
  - Notice that $a_1 = \mathsf{wr}_{q,w}$ does not modify the memory and thus is trivial.

- $a_0 = \mathsf{bu}_{p,v}$. If:
  - $a_1 = \mathsf{rd}_{q,w}$. If $p = q$ we have a contradiction to the hypothesis and if $p \neq q$ the conclusion is immediate.
  - $a_1 = \mathsf{bu}_{q,w}$. Then $p = q \Rightarrow (a_0 \# a_1)$ and $p \neq q$ the conclusion is trivial.

□

By means of this lemma we can relate the RMNF of speculations with global computations.

Proposition 4.17 (Global RMNF)**.** *Given a global computation γ with $\gamma = (C_i \xrightarrow[t_i]{a_i,o_i} C_{i+1})_{0\leq i\leq n}$ there exists an execution γ' starting from C_0 and ending in C_{n+1} such that $\gamma'|_t \propto^{MG} \gamma|_t$ and $\gamma'|_t$ is in RMNF for all $t \in \mathcal{T}id$.*

Proof. The proof is trivial by repeatedly applying lemmas 4.16 and 4.11. □

4.5.3 Step Ordering Analysis

For the results that follow we will need to identify events that are necessarily ordered by the dependencies induced by the memory model. The reader can observe that *relevant* (i.e. "nonirrelevant") events that cannot be reordered by the reordering relation are somehow related in a dependency chain. For that purpose we establish the following ordering definition between steps in a computation.

Definition 4.18 (Step ordering)**.** *Given a speculation σ, such that $\sigma = \sigma_0 \cdot \xrightarrow[o_0]{a_0} \cdot \sigma_1 \cdot \xrightarrow[o_1]{a_1} \cdot \sigma_2$, we say that the step $[\sigma_0, (a_0, o_0)]$ is ordered before the event $[\sigma_0 \cdot \xrightarrow[o_0]{a_0} \cdot \sigma_1, (a_1, o_1)]$, which we shall denote $[\sigma_0, (a_0, o_0)] \prec_\sigma [\sigma_0 \cdot \xrightarrow[o_0]{a_0} \cdot \sigma_1, (a_1, o_1)]$, iff for all σ' with $\sigma' \propto^{MG} \sigma$ then $\sigma' = \sigma'_0 \cdot \xrightarrow[o'_0]{a_0} \cdot \sigma'_1 \cdot \xrightarrow[o'_1]{a_1} \cdot \sigma'_2$ with $[\sigma_0, (a_0, o_0)] \sim [\sigma'_0, (a_0, o'_0)]$ and $[\sigma_0 \cdot \xrightarrow[o_0]{a_0} \cdot \sigma_1, (a_1, o_1)] \sim [\sigma'_0 \cdot \xrightarrow[o'_0]{a_0} \cdot \sigma'_1, (a_1, o'_1)]$.*

Notice that here we are using the step equivalence $\sim$ that was defined in 3.24.

We can immediately observe that if in a computation we have two dependent actions, in every reordering of that computation the events are in the same order. In particular steps with conflicting actions are step ordering related.

Remark 4.19 (Dependency implies Ordering)**.** *Given a speculation γ, such that $\gamma = (e_i \xrightarrow[o_i]{a_i} e_{i+1})_{0\leq i\leq n}$ where $a_j \mathcal{D}^{MG} a_h$ with $1 \leq j < h \leq n$ we have $[\sigma_{j-1}, (a_j, o_j)] \prec_\gamma [\sigma_{h-1}, (a_h, o_h)]$.*

Proof. The proof is trivial by induction on n and the definition of the reordering relation $\propto^{MG}$. □

Conversely, an event following a write, such that they are ordered, indicates that the second event is conflicting with the write, unless they are related by redex creation.

Remark 4.20 (Ordering implies Dependency)**.** *Given a speculation σ such that $\sigma = \sigma_0 \cdot \xrightarrow[o]{\mathsf{wr}_{p,v}} \cdot \xrightarrow[o_1]{a_1} \cdot \sigma_1$, if $[\sigma_0, (\mathsf{wr}_{p,v}, o)] \prec_\sigma [\sigma_0 \cdot \xrightarrow[o]{\mathsf{wr}_{p,v}}, (a_1, o_1)]$ with $a_1 \neq \beta$ then $\mathsf{wr}_{p,v} \mathcal{D}^{MG} a_1$.*

Proof. The proof is immediate by contradiction. □

We can now prove that if two events in a RMNF speculation are not related, there must be an equivalent speculation where these events are adjacent and in the opposite order.

Lemma 4.21. *Given a merge-calculus speculation σ such that $\sigma = \sigma_0 \cdot \xrightarrow[o_0]{a_0} \cdot \sigma_1 \cdot \xrightarrow[o_1]{a_1} \cdot \sigma_2$ and such that σ is in RMNF and $a_0, a_1 \notin \{\beta, \beta_v, \nu_{p,v}, \swarrow, \searrow\}$ and also $\neg([\sigma_0, (a_0, o_0)] \prec_\sigma [\sigma_0 \cdot \xrightarrow[o_0]{a_0} \cdot \sigma_1, (a_1, o_1)])$, then there exist σ_1' and σ_1'' such that:*

$$\sigma_0 \cdot \sigma_1' \cdot \xrightarrow[o_1']{a_1} \cdot \xrightarrow[o_0']{a_0} \cdot \sigma_1'' \cdot \sigma_2 \propto^{MG} \sigma$$

Proof. The proof proceeds by induction on the length of σ_1:

- In the base case $\sigma_1 = \epsilon$ and thus $\sigma = \sigma_0 \cdot \xrightarrow[o_0]{a_0} \cdot \xrightarrow[o_1]{a_1} \cdot \sigma_2$ and by lemma 4.19 we know $\neg(a_0 \mathcal{D}^{MG} a_1)$, and thus we can apply the lemma 2.15 to conclude.
- In the case where $\sigma_1 = \xrightarrow[o_2]{a_2} \cdot \sigma_1'''$ we proceed by cases:

  • If $\neg(a_0 \mathcal{D}^{MG} a_2)$ we can simply apply the asynchrony lemma (2.15) to obtain $\xrightarrow[o_2']{a_2} \cdot \xrightarrow[o_0']{a_0} \cdot \sigma_1''' \propto^{MG} \xrightarrow[o_0]{a_0} \cdot \sigma_1$. with σ_1''' having a shorter length than σ_1 we conclude by the induction hypothesis.

  • If $a_0 \mathcal{D}^{MG} a_2$ then $\neg([\sigma_0 \cdot \xrightarrow[o_0]{a_0}, (a_2, o_2)] \prec_\sigma [\sigma_0 \cdot \xrightarrow[o_0]{a_0} \cdot \sigma_1, (a_1, o_1)])$, otherwise we would have a contradiction with the hypothesis about the ordering of a_0 and a_1. Thus we can apply the induction hypothesis on the step $[\sigma_0 \cdot \xrightarrow[o_0]{a_0}, (a_2, o_2)]$ to obtain $\widehat{\sigma_1'}$ and $\widehat{\sigma_1''}$ with $\sigma_0 \cdot \xrightarrow[o_0]{a_0} \cdot \widehat{\sigma_1'} \cdot \xrightarrow[o_1'']{a_1} \cdot \xrightarrow[o_2']{a_2} \cdot \widehat{\sigma_1''} \cdot \sigma_2 \propto^{MG} \sigma$, where clearly $\widehat{\sigma_1'}$ has a shorter length than σ_1 which allows us to use the induction hypothesis to conclude.

□

Also, if two events are ordered by the step ordering relation, either their actions are dependent or there is an intermediate event that is conflicting with the first one and ordered with the second one.

Lemma 4.22. *Given $\gamma = \sigma_0 \cdot \xrightarrow[o_0]{a_0} \cdot \sigma_1 \cdot \xrightarrow[o_1]{a_1} \cdot \sigma_2$ a RMNF execution with $a_0, a_1 \notin \{\beta, \beta_v, \swarrow, \searrow\}$ and $[\sigma_0, (a_0, o_0)] \prec_\gamma [\sigma_0 \cdot \xrightarrow[o_0]{a_0} \cdot \sigma_1, (a_1, o_1)]$ we have one of the following:*

i) $a_0 \mathcal{D}^{MG} a_1$, or

ii) there are σ_1', σ_1'', a_2 and o_2 such that $\sigma_1 = \sigma_1' \cdot \xrightarrow[o_2]{a_2} \cdot \sigma_1''$ and $a_0 \mathcal{D}^{MG} a_2$ and $[\sigma_0 \cdot \xrightarrow[o_0]{a_0} \cdot \sigma_1', (a_2, o_2)] \prec_\gamma [\sigma_0 \cdot \xrightarrow[o_0]{a_0} \cdot \sigma_1, (a_1, o_1)]$

Proof. The proof is by induction on the length of σ_1. In the base case the conclusion is obvious satisfying the condition *i)*. In the induction case $\sigma_1 = \xrightarrow[o_3]{a_3} \cdot \widehat{\sigma_1}$. Clearly if $\neg(a_0 \mathcal{D}^{MG} a_3)$ we use lemma 4.13 to reorder them and conclude by means of the induction hypothesis. If $(a_0 \mathcal{D}^{MG} a_3)$ and $[\sigma_0 \cdot \xrightarrow[o_0]{a_0}, (a_3, o_3)] \prec_\gamma [\sigma_0 \cdot \sigma_1, (a_1, o_1)]$ we have the conclusion directly. Let us suppose then that $\neg[\sigma_0 \cdot \xrightarrow[o_0]{a_0}, (a_3, o_3)] \prec_\gamma [\sigma_0 \cdot \sigma_1, (a_1, o_1)]$. In this case we can apply lemma 4.21 to obtain $\widehat{\sigma_1'}$, $\widehat{\sigma_1''}$, o_1' and o_3' such that $\gamma \propto^{MG} \sigma_0 \cdot \xrightarrow[o_0]{a_0} \cdot \widehat{\sigma_1'} \cdot \xrightarrow[o_1']{a_1} \cdot \xrightarrow[o_3']{a_3} \cdot \widehat{\sigma_1''} \cdot \sigma_2$, with $\widehat{\sigma_1'}$ a shorter speculation than $\xrightarrow[o_3]{a_3} \cdot \sigma_1$, and hence we conclude by the induction hypothesis. □

The following lemmas state that some particular cases of events related by the step reordering relation, whose actions are not dependent as per the memory model reordering relation, have intermediate events that transitively relate them. These lemmas will be of importance for the following developments.

Lemma 4.23. *Given a RMNF speculation σ such that $\sigma = \sigma_0 \cdot \xrightarrow[o_0]{\mathsf{wr}_{p,v}} \cdot \sigma_1 \cdot \xrightarrow[o_1]{a_1} \cdot \sigma_2$ and $[\sigma_0, (\mathsf{wr}_{p,v}, o_0)] \prec_\sigma [\sigma_0 \cdot \xrightarrow[o_0]{\mathsf{wr}_{p,v}} \cdot \sigma_1, (a_1, o_1)]$ then either:*

i) $\mathsf{wr}_{p,v} \mathcal{D}^{MG} a_1$, *or*

ii) $\sigma_1 = \sigma_1' \cdot \xrightarrow[o']{b} \cdot \sigma_1'' \cdot \xrightarrow[o_2]{a_2} \cdot \sigma_1'''$ *with $b \in \mathcal{B}ar$ (where we consider possible that $\xrightarrow[o_2]{a_2} \cdot \sigma_1''' = \varepsilon$, in which case a_1 stands for a_2), and $b\ \mathcal{D}^{PSO}\ a_2$.*

Proof. The proof is by induction on the length of σ_1. Clearly if $\sigma_1 = \varepsilon$ then $\mathsf{wr}_{p,v} \mathcal{D}^{MG} a_1$ by lemma 4.20. Let us consider the induction case now. Let us assume that $\neg(\mathsf{wr}_{p,v} \mathcal{D}^{MG} a_1)$, otherwise we have the conclusion. We can then apply the lemma 4.22 to conclude that $\sigma_1 = \overline{\sigma_1} \cdot \xrightarrow[o_2]{a_2} \cdot \overline{\sigma_1'}$ with $\mathsf{wr}_{p,v} \mathcal{D}^{MG} a_2$ and $[\sigma_0 \cdot \xrightarrow[o_0]{\mathsf{wr}_{p,v}} \cdot \overline{\sigma_1}, (a_2, o_2)] \prec_\sigma [\sigma_0 \cdot \xrightarrow[o_0]{\mathsf{wr}_{p,v}} \cdot \sigma_1, (a_1, o_1)]$. Let us now consider the cases for a_2:

- if $a_2 = \mathsf{wr}_{q,w}$ for some $q \in \mathcal{R}ef$ and $w \in \mathcal{V}al$ we can apply the induction hypothesis considering $[\sigma_0 \cdot \xrightarrow[o_0]{\mathsf{wr}_{p,v}} \cdot \overline{\sigma_1}, (\mathsf{wr}_{q,w}, o_2)]$ in the place of $[\sigma_0, (\mathsf{wr}_{p,v}, o)]$, which renders the conclusion.
- if $a_2 = \mathtt{ww}$ we consider the following cases for a_1:
  - with $a_1 = \mathsf{wr}_{q,w}$ for some q and w we obtain the conclusion.
  - with $a_1 \in \{\mathsf{rd}_{q,w}, \mathsf{rd}^{\mathsf{o}}_{q,w} \mid q \in \mathcal{R}ef, w \in \mathcal{V}al\}$ we can consider using lemma 4.22 again to obtain that $\overline{\sigma_1'} = \widehat{\sigma_1} \cdot \xrightarrow[o_3]{a_3} \cdot \widehat{\sigma_1'}$ and $a_2 \mathcal{D}^{MG} a_3$ which implies that $a_3 = \mathsf{wr}_{q,w}$ for some q and w. Moreover

$$[\sigma_0 \cdot \xrightarrow[o]{\mathsf{wr}_{p,v}} \cdot \overline{\sigma_1} \cdot \xrightarrow[o_2]{a_2} \cdot \widehat{\sigma_1}, (a_3, o_3)] \prec_\sigma [\sigma_0 \cdot \xrightarrow[o]{\mathsf{wr}_{p,v}} \cdot \sigma_1, (a_1, o_1)]$$

  This concludes the case.
- if $a_2 = \mathtt{wr}$, then we consider the following cases for a_1:
  - with $a_1 \in \{\mathsf{rd}_{q,w}, \mathsf{rd}^{\mathsf{o}}_{q,w} \mid q \in \mathcal{R}ef, w \in \mathcal{V}al\}$ we have the conclusion of the lemma.

- with $a_1 = \mathsf{wr}_{q,w}$ for some q and w we can apply the lemma 4.22 to obtain that $\overline{\sigma_1'} = \widehat{\sigma_1} \cdot \xrightarrow[o_3]{a_3} \cdot \widehat{\sigma_1'}$ and $\mathtt{wr}\mathcal{D}^{MG}a_3$ which implies that $a_3 \in \{\mathsf{rd}_{r,v'}, \mathsf{rd}^{\circ}_{r,v'} \mid r \in \mathcal{R}ef, v' \in \mathcal{V}al\}$ which renders the conclusion. Moreover $[\sigma_0 \cdot \xrightarrow[o]{\mathsf{wr}_{p,v}} \cdot \overline{\sigma_1} \cdot \xrightarrow[o_2]{a_2} \cdot \widehat{\sigma_1}, (a_3, o_3)] \prec_\sigma [\sigma_0 \cdot \xrightarrow[o]{\mathsf{wr}_{p,v}} \cdot \sigma_1, (a_1, o_1)]$.

□

Lemma 4.24. *Given a speculation* $\sigma = \sigma_0 \cdot \xrightarrow[o]{\mathsf{wr}_{p,v}} \cdot \sigma_1 \cdot \xrightarrow[o']{a} \cdot \sigma_2$ *with* $a \in \{\mathsf{rd}_{q,w}, \mathsf{rd}^{\circ}_{q,w} \mid p \neq q\}$, *or* $a = \mathsf{rd}^{\circ}_{p,w}$, *and where there are no* `wr` *actions in* σ_1 *and* $[\sigma_0, (\mathsf{wr}_{p,v}, o)] \prec_\sigma [\sigma_0 \cdot \xrightarrow[o]{\mathsf{wr}_{p,v}} \cdot \sigma_1, (a, o')]$. *Then there exist* σ_1', σ_1'' *and* σ_1''' *such that* $\sigma_1 = \sigma_1' \cdot \xrightarrow[o_0]{\mathsf{wr}_{r,w}} \cdot \sigma_1'' \cdot \xrightarrow[o_1]{\mathsf{rd}_{r,w}} \cdot \sigma_1'''$.

Proof. The proof is by induction on the length of σ_1, with the base case being vacuously true since we assume that $[\sigma_0, (\mathsf{wr}_{p,v}, o)] \prec_\sigma [\sigma_0 \cdot \xrightarrow[o]{\mathsf{wr}_{p,v}} \cdot \sigma_1, (a, o')]$. Let us consider the inductive case. We have $\neg(\mathsf{wr}_{p,v} \ltimes^{MG} a)$, and thus by lemma 4.22 there must be the case that $\sigma_1 = \delta \cdot \xrightarrow[o_2]{a_2} \cdot \delta'$ with $\mathsf{wr}_{p,v}\mathcal{D}^{MG}a_2$ and $[\sigma_0 \cdot \xrightarrow[o]{\mathsf{wr}_{p,v}} \cdot \delta, (a_2, o_2)] \prec_\sigma [\sigma_0 \cdot \xrightarrow[o]{\mathsf{wr}_{p,v}} \cdot \sigma_1, (a, o')]$. Let us consider the cases for a_2 such that $\mathsf{wr}_{p,v}\mathcal{D}^{MG}a_2$ is satisfied:

- if $a_2 = \mathsf{rd}_{p,v'}$ we have the conclusion with $r = p$, $\sigma_1' = \varepsilon$ and taking $[\sigma_0, (\mathsf{wr}_{p,v}, o)]$ for the write event.
- if $a_2 = \mathsf{wr}_{r,v'}$ we have from the lemma 4.22 that $[\sigma_0 \cdot \xrightarrow[o]{\mathsf{wr}_{p,v}} \cdot \delta, (\mathsf{wr}_{r,v'}, o_2)] \prec_\sigma [\sigma_0 \cdot \xrightarrow[o]{\mathsf{wr}_{p,v}} \cdot \sigma_1, (a, o')]$, and thus we can apply the induction hypothesis to conclude.
- if $a_2 = \mathtt{ww}$ then can apply again the lemma 4.22 to obtain that $\delta' = \delta_0 \cdot \xrightarrow[o_3]{a_3} \cdot \delta_1$ with $\mathtt{ww}\mathcal{D}^{MG}a_3$ which implies that $a_3 = \mathsf{wr}_{r,v'}$ for some r and v'. Once more $[\sigma_0 \cdot \xrightarrow[o]{\mathsf{wr}_{p,v}} \cdot \delta \cdot \xrightarrow[o_2]{a_2} \cdot \delta_0, (a_3, o_3)] \prec_\sigma [\sigma_0 \cdot \xrightarrow[o]{\mathsf{wr}_{p,v}} \cdot \sigma_1, (a, o')]$ so we can apply the induction hypothesis to conclude.

□

Corollary 4.25. *Given a speculation* $\sigma = \sigma_0 \cdot \xrightarrow[o]{\mathtt{ww}} \cdot \sigma_1 \cdot \xrightarrow[o']{a} \cdot \sigma_2$ *with* $a \in \{\mathsf{rd}_{q,v}, \mathsf{rd}^{\circ}_{q,w} \mid p \neq q\}$ *or* $a = \mathsf{rd}^{\circ}_{p,w}$, *and where there are no* `wr` *actions in* σ_1 *and* $[\sigma_0, (\mathtt{ww}, o)] \prec_\sigma [\sigma_0 \cdot \xrightarrow[o]{\mathtt{ww}} \cdot \sigma_1, (a, o')]$. *Then there exist* σ_1', σ_1'' *and* σ_1''' *such that* $\sigma_1 = \sigma_1' \cdot \xrightarrow[o_0]{\mathsf{wr}_{r,w}} \cdot \sigma_1'' \cdot \xrightarrow[o_1]{\mathsf{rd}_{r,v'}} \cdot \sigma_1'''$.

Proof. The proof is trivial applying lemmas 4.22 and 4.24. □

Lemma 4.26. *Let* $\sigma = \sigma_0 \cdot \xrightarrow[o]{\mathsf{wr}_{p,v}} \cdot \sigma_1 \cdot \xrightarrow[o']{\mathsf{wr}_{q,w}} \cdot \sigma_2$ *where* σ_1 *contains no* `ww` *action, with* $p \neq q$ *and* $[\sigma_0, (\mathsf{wr}_{p,v}, o)] \prec_\sigma [\sigma_0 \cdot \xrightarrow[o]{\mathsf{wr}_{p,v}} \cdot \sigma_1, (\mathsf{wr}_{q,w}, o')]$. *Then* $\sigma_1 = \sigma_1' \cdot \xrightarrow[o']{\mathsf{rd}_{p,v'}} \cdot \sigma_1''$.

Proof. The proof is by induction on the length of σ_1, with the base case being vacuous since $\sigma_1 = \epsilon$ contradicts $[\sigma_0, (\mathsf{wr}_{p,v}, o)] \prec_\sigma [\sigma_0 \cdot \xrightarrow[o]{\mathsf{wr}_{p,v}} \cdot \sigma_1, (\mathsf{wr}_{q,w}, o')]$. Let us consider the inductive case. We have $\neg(\mathsf{wr}_{p,v} \ltimes^{MG} \mathsf{wr}_{q,w})$ if $p \neq q$, and thus by lemma 4.22 there must be the case that $\sigma_1 = \delta \cdot \xrightarrow[o_2]{a_2} \cdot \delta'$ with $(\mathsf{wr}_{p,v} \mathcal{D}^{MG} a_2)$ and $[\sigma_0 \cdot \xrightarrow[o]{\mathsf{wr}_{p,v}} \cdot \delta, (a_2, o_2)] \prec_\sigma [\sigma_0 \cdot \xrightarrow[o]{\mathsf{wr}_{p,v}} \cdot \sigma_1, (\mathsf{wr}_{q,w}, o')]$. Let us consider the cases for a_2 such that $\mathsf{wr}_{p,v} \mathcal{D}^{MG} a_2$ is satisfied: if $a_2 = \mathsf{rd}_{p,v'}$ we have the conclusion, and if $a_2 = \mathsf{wr}_{p,v'}$ we apply the induction hypothesis. We have from the hypothesis that `ww` does not occur in σ_1 so this concludes the lemma. □

4.5.4 From Write-Buffers to Speculations

It is fairly straightforward to see that for every computation of PSO as given by the semantics of write-buffers the exact same computation is a computation of the merge-calculus.

Remark 4.27. *Any computation $\gamma : C \xrightarrow{*} C'$ of PSO as provided by the semantics of write-buffers (of Figure 4.4) is a legal execution of the merge-calculus as well.*

We will call these computations of the merge-calculus *purely buffered*, since the only relaxation is provided by means of buffers and not speculations.

To prove our correspondence result, we need to construct a speculative computation that corresponds to one in the semantics with buffers. However, in the semantics of the merge-calculus the requirements for $\mathsf{rd}^{\mathsf{o}}_{p,v}$ actions are almost vacuous, whereas in the semantics with write-buffers these actions can only happen under some conditions regarding the buffers. Indeed, the conditions required in the semantics with write-buffers are important to prove the correspondence of the semantics. Our proof proceeds by showing how actions can be reordered to reach a computation of the merge-calculus that corresponds to a speculative computation. In order to prove that we can reorder the actions we use the fact that they were generated by a valid computation of the semantics with buffers. For that purpose we define a property on computations of the merge-calculus that indicates that actions that have not yet been reordered do comply with the semantics of write-buffers.

Definition 4.28 (Buffer Compliance)**.** *We say that the computation γ complies with the semantics of buffers, denoted by $\mathsf{WB}(\gamma)$, if whenever $\gamma = \gamma_0 \cdot \xrightarrow[t,o]{\mathsf{wr}_{p,v}} \cdot \gamma_1 \cdot (C \xrightarrow[t,o']{\mathsf{rd}^{\mathsf{o}}_{p,v}} C') \cdot \gamma_2$ with $C = (S, (B, t, e) \| T)$ and $\mathsf{match}[(\gamma_0 \cdot \xrightarrow[t,o]{\mathsf{wr}_{p,v}} \cdot \gamma_1)|_t, (\mathsf{rd}^{\mathsf{o}}_{p,v}, o')] = [\gamma_0|_t, (\mathsf{wr}_{p,v}, o)]$ then $B(p) = s \cdot v \cdot \mathtt{ww}^n$ and `wr` does not occur in s.*

In other words $\mathsf{rd}^{\mathsf{o}}_{p,v}$ actions respect the semantics of write-buffers. Importantly the condition $\mathsf{WB}(\gamma)$ only requires that the value read be the last value in the buffer for own reads that follow their matching write in γ. It is easy to see that this condition is satisfied for every computation of the merge-calculus that is a computation of the semantics of PSO with write-buffers (Figure 4.4).

Remark 4.29. *Given a computation $\gamma : C \xrightarrow{*} C'$ of PSO as provided by the semantics of write-buffers (as in Figure 4.4) we have* $\mathsf{WB}(\gamma)$.

Evidently every purely buffered computation γ satisfies $\mathsf{WB}(\gamma)$.

Before considering the reordering of actions required to obtain a speculative computation from one in the semantics of write-buffers we need the following definition, stating that in a prefix of the computation every buffer update, or committed barrier is immediately preceded by the write, or a barrier action that justifies it.

Definition 4.30 (Late-commit freedom)**.** *We say a speculative computation γ is late-commit free, if $\gamma = \sigma_0 \cdot (C_0 \xrightarrow[t_0,o_0]{a_0} C \xrightarrow[t_1]{a_1} C_1) \cdot \sigma_1$ implies that:*

$$\begin{cases} a_1 = \mathsf{bu}_{p,v} & \Rightarrow \quad t_0 = t_1 \;\&\; a_0 = \mathsf{wr}_{p,v} \;\& \\ & \qquad\quad C_0 = (S, (B, t, e) \| T) \Rightarrow B(p) = \epsilon \\ a_1 = \overline{\mathtt{ww}} & \Rightarrow \quad t_0 = t_1 \;\&\; a_0 = \mathtt{ww} \;\&\; C_0 = (S, (\epsilon, t, e) \| T) \\ a_1 = \overline{\mathtt{wr}} & \Rightarrow \quad t_0 = t_1 \;\&\; a_0 = \mathtt{wr} \;\&\; C_0 = (S, (\epsilon, t, e) \| T) \end{cases}$$

We can prove a lemma that shows that events that depend on a write event can be permuted after the buffer update that corresponds to the write being considered. This will later enable us to move the write action to the place where its corresponding buffer update is. Once all writes immediately precede their corresponding update we have an execution that is similar to a speculative one.

Lemma 4.31 (Delayed Dependencies)**.** *Let γ be a RMNF computation satisfying* $\mathsf{WB}(\gamma)$. *Suppose that $\gamma = \gamma_0 \cdot \xrightarrow[t,o]{\mathsf{wr}_{p,v}} \cdot \gamma_1 \cdot \xrightarrow[t,\varepsilon]{\mathsf{bu}_{p,v}} \cdot \gamma_2$, where $\gamma_0 \cdot \xrightarrow[t,o]{\mathsf{wr}_{p,v}} \cdot \gamma_1$ is the longest late commit free prefix of γ. Suppose as well that $[\gamma_0|_t, (\mathsf{wr}_{p,v}, o)]$ is the first uncommitted write to p by t in γ. Let γ_1' and γ_1'' be such that $\gamma_1 = \gamma_1' \cdot \xrightarrow[t,o_1]{a_1} \cdot \gamma_1''$ with $[\gamma_0|_t, (\mathsf{wr}_{p,v}, o)] \prec_{\gamma|_t} [\gamma_0 \cdot \xrightarrow[o]{\mathsf{wr}_{p,v}} \cdot \gamma_1', (a_1, o_1)]$ and $\neg(a_1 \# \gamma_1''|_t)$. Then there exists $\widehat{\gamma_1}$ such that*

$$\gamma' = \gamma_0 \cdot \xrightarrow[t,o]{\mathsf{wr}_{p,v}} \cdot \gamma_1' \cdot \widehat{\gamma_1} \cdot \xrightarrow[t,\varepsilon]{\mathsf{bu}_{p,v}} \cdot \xrightarrow[t,o_1']{a_1} \cdot \gamma_2$$

and for all $t' \neq t$ we have $\gamma'|_{t'} = \gamma|_{t'}$ and $\gamma'|_t \propto^{MG} \gamma|_t$. Moreover we have $\mathsf{WB}(\gamma')$.

Proof. By induction on the size of γ_1''.

- In the base case we have $\gamma = \gamma_0 \cdot \xrightarrow[t,o]{\mathsf{wr}_{p,v}} \cdot \gamma_1' \cdot \xrightarrow[t,o_1]{a_1} \cdot \xrightarrow[t,\varepsilon]{\mathsf{bu}_{p,v}} \cdot \gamma_2$. Here we clearly have that $a_1 \neq \mathsf{rd}_{p,v'}$ for the semantics disallows $a_1 = \mathsf{rd}_{p,v'}$, and we notice that lemma 4.15 allows to reorder a_1 and $\mathsf{bu}_{p,v}$ provided that $\neg(a_1 \triangleright_C \mathsf{bu}_{p,v})$ which is guaranteed since the buffer of t has a pending write on p not generated by a_1. Also notice that if $a_1 = \mathsf{rd}^{\mathsf{o}}_{q,w}$ (where possibly $p = q$) then by lemma 4.23 that there must be a preceding barrier b, which by the hypothesis $\mathsf{WB}(\gamma)$ cannot be a `wr`. From the construction of the proof of the lemma 4.23 we know that the barrier b (= `ww`) is ordered before $\mathsf{rd}^{\mathsf{o}}_{q,w}$, so we can apply the lemma 4.25 which implies that there is a $\mathsf{wr}_{q,w}$ and a following $\mathsf{rd}_{q,v'}$ action in γ_1'; a contradiction to the semantics of buffers, provided by hypothesis $\mathsf{WB}(\gamma)$, since the write of p is pending and there is an intermediate `ww` barrier. Hence $a_1 \neq \mathsf{rd}^{\mathsf{o}}_{q,w}$. This guarantees that the permutation of a_1 after the $\mathsf{bu}_{p,v}$ action preserves $\mathsf{WB}(\gamma')$.

- Suppose now that $\gamma_1'' = \xrightarrow[t',o_2]{a_2} \cdot \bar{\gamma_1}$. Consider the following cases:
  - if $t = t'$ we can verify once more that $a_1 \neq \mathsf{rd}^{\circ}_{q,w}$ since $[\gamma_0|_t, (\mathsf{wr}_{p,v}, o)] \prec_{\gamma|_t} [\gamma_0 \cdot \xrightarrow[o]{\mathsf{wr}_{p,v}} \cdot \gamma_1', (a_1, o_1)]$ in conjunction with lemmas 4.23 and 4.25 would violate the semantics of buffers granted by $\mathsf{WB}(\gamma)$.
    By cases on a_1. Suppose that $a_1 = \mathsf{rd}_{q,w}$ then we have from the semantics, that $q \neq p$ (no read can happen with pending writes). By lemma 4.23 there must be a preceding barrier b and a following action that is dependent on b. Of course b cannot be a `wr` since it would violate the semantics of buffers. So it must be a `ww` and there must be an intermediate write on reference q that conflicts with the read $\mathsf{rd}_{q,w}$, as per lemma 4.25. Thus, the preceding write on p should be committed before the one for q and there could not be a $\mathsf{rd}_{q,w}$ action. Hence $a_1 \neq \mathsf{rd}_{q,w}$. Otherwise suppose that $a_1 = \mathsf{wr}_{q,w}$. If $q = p$ we have from the semantics of buffers that a_1 is not committed in σ_1'' and thus it is trivial to see that can be reordered after the buffer update (using lemmas 4.15,4.16 and 4.13). So, in particular can be reordered with a_2. If $p \neq q$ then there must be an intermediate `ww`, else we would have no ordering (again as per lemmas 4.23 and 4.26); thus we know that the write is not committed in σ_1' (otherwise we would have a violation to the semantics of buffers) and we can permute a_1 after the buffer update, thus in particular after a_2. We observe then that the resulting computation γ' satisfies $\mathsf{WB}(\gamma')$ and we can use the induction hypothesis to conclude.
  - if $t \neq t'$ we can trivially reorder a_1 (recall from the previous case that $a_1 \notin \{\mathsf{rd}_{q,w}, \mathsf{rd}^{\circ}_{q,w}\}$) with a_2 by lemma 4.16 and conclude by the induction hypothesis. Notice that the permutation trivially preserves $\mathsf{WB}(\gamma')$.

□

It should be easy to see that similar (but simpler) results can be derived in the case where the first late-commit is a $\overline{\texttt{ww}}$ or $\overline{\texttt{wr}}$ action. What is important to observe here is that from the proof we know that only $\mathsf{wr}_{p,w}$, `ww` or `wr` actions need to be permuted. Read actions need never be reordered. We will use this observation in the sequel.

The following corollary states that we can always find a speculation where writes are immediately followed by their corresponding buffer updates. Clearly the same holds for barrier actions as shown in the subsequent corollaries. This property is the core of the proof establishing that the semantics of speculations can simulate the one of write-buffers.

Corollary 4.32 (Matching Write Update)**.** *Let $\gamma : C \xrightarrow{*} C'$ be a RMNF computation such that $\mathsf{WB}(\gamma)$ and $\gamma = \sigma_0 \cdot \xrightarrow[t,o]{\mathsf{wr}_{p,v}} \cdot \sigma_1 \cdot \xrightarrow[t,\varepsilon]{\mathsf{bu}_{p,v}} \cdot \sigma_2$ with $\sigma_0 \cdot \xrightarrow[t,o]{\mathsf{wr}_{p,v}} \cdot \sigma_1$ the longest late-commit free prefix of γ, and with $[\sigma_0|_t, (\mathsf{wr}_{p,v}, o)]$ the first uncommitted write on reference p of thread t in γ. Then there exists o', σ_1' and σ_1'' and $\gamma' : C \xrightarrow{*} C'$ such that*

$$\gamma' = \sigma_0 \cdot \sigma_1' \cdot \xrightarrow[t,o']{\mathsf{wr}_{p,v}} \cdot \xrightarrow[t,\varepsilon]{\mathsf{bu}_{p,v}} \cdot \sigma_1'' \cdot \sigma_2$$

and for all $t' \neq t$ we have $\gamma|'_t = \gamma'|'_t$ and $\gamma'|_t \propto^{MG} \gamma|_t$. Moreover $\mathsf{WB}(\gamma')$ holds.

Proof. Simple induction on σ_1 repeatedly applying lemmas 4.31, 4.15 and 4.16. □

Corollary 4.33 (Matching Barrier Update)**.** *Let $\gamma : C \xrightarrow{*} C'$ be a RMNF computation such that $\mathsf{WB}(\gamma)$ and $\gamma = \sigma_0 \cdot \xrightarrow[t,o]{b} \cdot \sigma_1 \cdot \xrightarrow[t,\varepsilon]{\bar{b}} \cdot \sigma_2$ with $b \in \mathcal{S}ync$ and $\sigma_0 \cdot \xrightarrow[t,o]{b} \cdot \sigma_1$ the longest late-commit free prefix of γ, and with $[\sigma_0|_t, (b, o)]$ the first uncommitted b barrier of thread t in γ. Then there exists o', σ'_1 and σ''_1 and $\gamma' : C \xrightarrow{*} C'$ such that*

$$\gamma' = \sigma_0 \cdot \sigma'_1 \cdot \xrightarrow[t,o']{b} \cdot \xrightarrow[t,\varepsilon]{\bar{b}} \cdot \sigma''_1 \cdot \sigma_2$$

and for all $t' \neq t$ we have $\gamma|'_t = \gamma'|'_t$ and $\gamma'|_t \propto^{MG} \gamma|_t$. Moreover $\mathsf{WB}(\gamma')$ holds.

Proof. Same as for the previous lemma. □

To establish a relation between computations in the formalization with write-buffers and the formalization with speculations we need to identify which are the computations of the merge-calculus that correspond to computations of the speculative semantics. For that purpose we will define *quasi-speculative* computations.

Definition 4.34 (Quasi-speculative Computation)**.** *A computation $\gamma : C \xrightarrow{*} C'$ of the merge-calculus is called* quasi-speculative *if every write is immediately followed by its corresponding buffer update, and every barrier action is followed immediately by its corresponding barrier commit.*

Lemma 4.35. *Given $\gamma : C \xrightarrow{*} C'$ a computation of the merge-calculus satisfying $\mathsf{WB}(\gamma)$ there exists a quasi-speculative computation γ' of the merge-calculus such that for every thread t we have $\gamma'|_t \propto^{MG} \gamma|_t$.*

Proof. The proof proceeds by induction on the number of buffer update (including barrier commit) actions present in γ and orders writes, or barriers to match their respective buffer-update action as stated by corollaries 4.32 and 4.33. □

In a quasi-speculative computation of the merge-calculus we still have the buffer updates and the commits of barrier symbols in the buffers. To obtain a truly-speculative computation we need to erase these actions from the computation. Let us denote by $\lfloor\gamma\rfloor$ the computation that results from erasing all commit actions from the merge-calculus computation γ.

Theorem 4.36 (Write Buffering is Speculative)**.** *Given a computation $\gamma : C \xrightarrow{*} C'$ of the formalization of PSO with write-buffers (as defined in Figure 4.4) there exists a quasi-speculative computation $\gamma' : C \xrightarrow{*} C'$ such that for every thread t it holds $\gamma'|_t \propto^{MG} \gamma|_t$. Then $\lfloor\gamma'\rfloor$ is a $\mathcal{D}^{PSO}$-valid speculative computation.*

Proof. Given the computation γ we have from remarks 4.27 and 4.29 that γ is a computation of the merge-calculus and in particular we have $\mathsf{WB}(\gamma)$. We can therefore apply the Lemma 4.35 to obtain $\gamma' : C \xrightarrow{*} C'$ a quasi-speculative computation. It is not hard to see from the construction of γ' that since the execution γ satisfies $\mathsf{WB}(\gamma)$ the final computation $\lfloor\gamma'\rfloor$ is a $\mathcal{D}^{PSO}$-valid speculative computation. □

4.5.5 From Speculations to Write-Buffers

Let us see now how a computation of the speculative formalization of PSO can be turned into one of the formalization with write buffers by reordering speculatively performed actions to the position where they become "normal". In essence, since we consider only $\mathcal{D}^{PSO}$-valid speculations, we know that for every thread projection $\gamma|_t$, for an hypothetical speculation γ, there exists a *normal* speculation that is a $\mathcal{D}^{PSO}$-reordering of $\gamma|_t$, and let us denote such normal computation $\gamma_{[t]}$. What we prove here, is that we can always find a computation γ' of the merge-calculus such that coincides with γ, the initial and final states are the same, and for every thread $\gamma'|_t = \gamma_{[t]}$. Let us now proceed with the proof.

Given a valid speculative computation γ we can trivially obtain a quasi-speculative computation $\lceil\gamma\rceil$ of the merge-calculus where all writes and barrier instructions are immediately committed.

Remark 4.37. *Given a valid computation $\gamma : C \xrightarrow{*} C'$ of the speculative calculus there is a quasi-speculative computation $\lceil\gamma\rceil : C \xrightarrow{*} C'$ of the merge-calculus such that for all t we have $\gamma|_t = \gamma'|_t$.*

We now prove that buffer commit actions can be reordered w.r.t. every action other than a read or a buffer commit action of the same thread to reach the same final configuration.

Lemma 4.38. *Suppose that we have $C \xrightarrow[t,\varepsilon]{a_0} C_0 \xrightarrow[t,o_1]{a_1} C'$ and $a_0 \in \{\mathsf{bu}_{p,v}, \overline{\mathtt{ww}}, \overline{\mathtt{wr}} \mid p \in \mathcal{R}ef, v \in \mathcal{V}al\}$ and $a_1 \notin \{\mathsf{rd}_{p,v}, \mathsf{bu}_{p,v}, \overline{\mathtt{ww}}, \overline{\mathtt{wr}} \mid p \in \mathcal{R}ef, v \in \mathcal{V}al\}$, then there exists C_1 such that $C \xrightarrow[t,o_1]{a_1} C_1 \xrightarrow[t,\varepsilon]{a_0} C'$.*

Proof. The proof is trivial by case analysis. Notice that no a_1 action other than a write modifies or depends on the buffers or memory. In the case of a write action, it simply puts its contents at the end of the buffer which is independent of any previous buffer update. □

If an action does not conflict with preceding actions of different threads, then this action can be moved backwards in the computation obtaining an equivalent computation (in the sense that individual speculations are preserved and the initial and final configurations are the same).

Lemma 4.39. *Let $\gamma = C \xrightarrow{*} C'$ be a computation of the merge-calculus and let $\gamma = \gamma_0 \cdot \xrightarrow[t,o_0]{a_0} \cdot \gamma_1 \cdot \xrightarrow[t,o_1]{a_1} \cdot \gamma_2$ with $a_1 \notin \{\mathsf{rd}_{p,v}, \mathsf{bu}_{p,v}, \overline{\mathtt{wr}}, \overline{\mathtt{ww}} \mid p \in \mathcal{R}ef, v \in \mathcal{V}al\}$ and $\gamma_1|_t = \epsilon$. Then there exists $\gamma' = C \xrightarrow{*} C'$ and γ'_1 such that for all t we have $\gamma|_t = \gamma'|_t$ and $\gamma' = \gamma_0 \cdot \xrightarrow[t,o_0]{a_0} \cdot \xrightarrow[t,o_1]{a_1} \cdot \gamma'_1 \cdot \gamma_2$.*

Proof. We proceed by induction on the length of γ_1. Clearly if $\gamma_1 = \varepsilon$ we have the conclusion. If $\gamma_1 = \overline{\gamma_1} \cdot \xrightarrow[t_2,o_2]{a_2}$ we have the following cases:

- $t = t_2$ and therefore we know that $a_2 \in \{\mathsf{bu}_p, \overline{\mathtt{wr}}, \overline{\mathtt{ww}} \mid p \in \mathcal{R}ef\}$ by $\gamma_1|_t = \varepsilon$. We can simply apply the previous lemma (4.38) and conclude by the induction hypothesis.
- $t \neq t_2$ and since $a_1 \notin \{\mathsf{rd}_{p,v}, \mathsf{bu}_{p,v}, \overline{\mathtt{ww}}, \overline{\mathtt{wr}} \mid p \in \mathcal{R}ef, v \in \mathcal{V}al\}$ we can directly apply Lemma 4.16 and the induction hypothesis to conclude.

□

And similarly to the previous result, if an action does not conflict with subsequent actions of different threads in the computation, we can push it forward to obtain an equivalent computation.

Lemma 4.40. *Let $\gamma : C \xrightarrow{*} C'$ be a merge-calculus computation such that $\gamma = \gamma_0 \cdot \xrightarrow[t,o]{a} \cdot \gamma_1 \cdot \gamma_2$ with $a \notin \{\mathsf{wr}_{p,v}, \mathsf{rd}_{p,v}, \mathsf{bu}_{pv}, \overline{\mathtt{ww}}, \overline{\mathtt{wr}} \mid p \in \mathcal{R}ef, v \in \mathcal{V}al\}$ and $\gamma_1|_t = \epsilon$. Then there are γ' and γ'_1 such that $\gamma' = \gamma_0 \cdot \gamma'_1 \cdot \xrightarrow[t,o]{a} \cdot \gamma_2$ with $\gamma' : C \xrightarrow{*} C'$ and for all $t \in \mathcal{T}id$ we have $\gamma|_t = \gamma'|_t$.*

Proof. The proof is obvious since a does not modify or inspect the buffers or the memory. □

Lemma 4.41. *Let $\gamma : C \xrightarrow{*} C'$ be a merge-calculus computation such that every thread projection is* valid*, i.e. for all $t \in \mathcal{T}id$ then $\gamma|_t \propto^{MG} \gamma_{[t]}$ with $\gamma_{[t]}$ a normal speculation. Suppose as well that $\gamma|_t \propto^{MG} \gamma'_t \propto^{MG} \gamma_{[t]}$ with $\gamma|_t$ reaching γ'_t by a single reordering. Namely $\gamma = \gamma_0 \cdot \xrightarrow[t,o_0]{a_0} \cdot \gamma_1 \cdot \xrightarrow[t,o_1]{a_1} \cdot \gamma_2$ with $\gamma_1|_t = \epsilon$ and $\gamma'_t = \gamma_0|_t \cdot \xrightarrow[o'_1]{a_1} \cdot \xrightarrow[o'_0]{a_0} \cdot \gamma_2|_t$. Then there exists γ'_1, γ''_1 and $\gamma' : C \xrightarrow{*} C'$ such that $\gamma' = \gamma_0 \cdot \gamma'_1 \cdot \xrightarrow[t,o'_1]{a_1} \cdot \xrightarrow[t,o'_0]{a_0} \cdot \gamma''_1 \cdot \gamma_2$. Moreover for all $t' \in \mathcal{T}id$ with $t' \neq t$ we have $\gamma'|_t = \gamma|_t$.*

Proof. Let us proceed by cases on a_1:

- If $a_1 \in \{\mathsf{rd}_{p,v}, \mathsf{rd}^{\circ}_{p,v} \mid p \in \mathcal{R}ef, v \in \mathcal{V}al\}$ we have from the fact that $\gamma|_t \propto^{MG} \gamma'_t$ that $\neg(a_1 \mathcal{D}^{PSO} a_0)$ which implies that $a_0 \notin \{\mathsf{rd}_{q,w}, \mathsf{rd}^{\circ}_{q,w}, \mathsf{wr}_{q,w} \mid q \in \mathcal{R}ef, w \in \mathcal{V}al\}$ and thus we can apply the lemmas 4.40 and 4.15 to conclude.
- If $a_1 = \mathsf{wr}_{p,v}$ we can apply the lemmas 4.39 and 4.15 to get the desired reordering.
- If If $a_1 = \{\mathtt{ww}, \mathtt{wr}\}$ we proceed as in the previous case.
- All the remaining cases are trivial.

□

One remark from the proof of this lemma is that it requires to reorder only memory-model-related actions (that is reads, writes or memory barriers) forward in the computation (in particular in the speculation). One can observe that the actions that need to be reordered are write actions and barrier actions, where read actions can be regarded as remaining at their place. Indeed, we see in the proof that if a_1 is a read action the a_0 action is moved to a later stage in the computation (but in this case a_0 is not a memory model related action), and in the cases where a_1 is a write or a barrier, it is this action that is moved to the front of the computation.

The following lemma states that merge-calculus computations corresponding to $\mathcal{D}^{PSO}$-valid speculative computations can be reordered to reach a computation of the semantics with write-buffers.

Lemma 4.42. *Let $\lceil\gamma\rceil : C \xrightarrow{*} C'$ be a quasi-speculative computation of the merge-calculus corresponding to a $\mathcal{D}^{PSO}$-valid speculative computation γ of the*

formalization of PSO by means of speculations, and let $\gamma' : C \xrightarrow{*} C'$ *be a merge-calculus computation such that for all* $t \in \mathcal{T}id$, $\gamma|_t \propto^{MG} \gamma'|_t$. *Then there is* $\gamma'' : C \xrightarrow{*} C'$ *a purely buffered computation of the merge-calculus such that for all* $t \in \mathcal{T}id$, $\gamma'|_t \propto^{MG} \gamma''|_t$.

Proof. The proof is by induction on the summation of reorderings needed to reach $\gamma_{[t]}$ from $\gamma'|_t$ for every t. In the base case all projections are normal, and thus the computation corresponds to a computation of the calculus with write buffers. We simply apply the previous lemma (4.41) in the induction case and conclude by the induction hypothesis. Notice as well that the validity condition guarantees that $\mathsf{rd}^{\circ}_{p,v}$ actions are actually allowed and moreover obtain the correct value from the buffers. □

And we can finally prove that $\mathcal{D}^{PSO}$-valid computations in the speculative semantics of PSO correspond to computations of the semantics of PSO with write-buffers.

Theorem 4.43 (PSO Speculations are Write Buffering)**.** *Given* $\gamma : C \xrightarrow{*} C'$ *a* $\mathcal{D}^{PSO}$*-valid speculative computation of the formalization of PSO with speculations. There exists a purely buffered computation of the merge-calculus* $\gamma'' : C \xrightarrow{*} C$ *such that for all* $t \in \mathcal{T}id$ *then* $\gamma|_t \propto^{MG} \gamma''|_t = \gamma_{[t]}$. *And in particular* γ'' *is a computation of the calculus with buffers.*

Proof. The proof is simple consequence of Remark 4.37 and Lemma 4.42. □

Interestingly, the formalization by means of speculations resembles significantly to the formal description of the memory models provided in the Appendix D of Sparc's specification [SPARC, 1994]. However, there are some major differences, that are mainly induced by the fact that their formalization is axiomatic, and thus concepts like data and control dependencies are hard to formulate. The axiomatic specification provided by Sparc requires the existence of a total order among the events of the computation that bears some resemblance with the computations we obtain *operationally* by our semantics of speculations.

Then, an interesting consequence of our theorem is that it provides a formalization to the informal claim in the Sparc specification [SPARC, 1994] that states:

> The distinction between local and remote stores permits use of store-buffers, which are explicitly supported in all SPARC-V9 memory models.

To the best of our knowledge there is no previous proof that justifies that the axiomatic formalization in the Sparc document guarantees that behaviors appearing in write buffering architectures are allowed. Our speculative formalization, which as we said above, is intuitively easy to relate to the axiomatic formalization of Sparc, can be proved to allow the behaviors induced by write buffering. We consider that providing such proofs is an important advantage enabled by the use of operational techniques.

4.6 Conclusion

We conclude this chapter by recalling the principal contributions. The main contributions are the formalizations of the relaxed memory models of the Sparc

architecture [SPARC, 1994] in an operational way. In particular, these formalizations serve both, as a faithful semantics for working with those memory models, and as an exercise to prove the flexibility of the frameworks presented in Chapters 2 and 3 in formalizing existing relaxed memory models.

Two of the relaxed memory models of Sparc, namely TSO and PSO, are formalized by means of the framework of write buffers of Chapter 2. Actually, the only modifications, other than the syntax, needed to render the framework of Chapter 2 suitable for TSO and PSO are: to disallow thread creation, adopting a static thread system, with a single write buffer for each processor (which greatly simplifies the buffer update mechanism); and for the particular case of TSO, disabling the possibility of updating buffers in an order different from that in which they are filled, this is simply done by considering the buffer as a single queue of writes (for all references).

In fact, if we consider the syntax of Chapter 2 with the modifications required to model TSO, that is, static thread systems and no reordering of writes in the buffers, we get a model that is almost identical to that presented in [Owens et al., 2009; Sewell et al., 2010] (except for the syntax, of course). However, since for Sparc we have to consider a lower level syntax, we need to add new rules for the handling of barriers and the compare-and-swap constructs of that architecture. The rules we adopt for barriers are minimal, in the sense that a certain type of barrier enforces ordering constraints only on events of the kind required, unlike the lock construct of Chapter 2.

By means of the speculative framework of Chapter 3 we are able to formalize the three memory models of Sparc. The formalization of these models requires a careful instantiation of the validity condition, that has to capture the exact behaviors allowed by write-buffering architectures. In addition to being able to model the RMO memory model, which is not suitable for the framework of write buffers of Chapter 2, the operational formalization we provide for these memory models, by means of speculations, resembles the axiomatic formalization of the original specification of Sparc. This enables comparing these two formalizations, which we plan to do in future research.

Finally, the main result of this chapter is the proof of the coincidence of the formalizations of PSO and TSO with the write buffering framework on one side, and the speculative framework on the other. This result provides support to the claim that the Sparc memory models allow write buffering. Moreover, it provides a proof that, to some extent, the framework of speculative computation is more general than the one of write buffering.

Chapter 5

Conclusion

The goal of this thesis was, as the title indicates, the development of operational semantics for relaxed memory models, and more generally, relaxed execution models. To that end we provided two general frameworks that can be instantiated to obtain operational characterizations of realistic memory models. These frameworks capture a vast set of execution relaxations that are common to existing relaxed memory models.

Our first framework, in Chapter 2, is based on the notion of *write buffers*. This optimizing technique is of paramount importance for the performance of programs, and therefore has been widely adopted in many multiprocessor architectures like [Intel Corporation, 2007; AMD, 2010; SPARC, 1994] just to mention a few. From a semantic point of view the inclusion of write buffers implies many relaxations with respect to the interleaving semantics. If we consider the relaxations allowed by our formalization of write buffers, according to the terminology of [Adve and Gharachorloo, 1996], we see that it enables the reordering of a write followed (in the program text) by a read on a different memory location, denoted ($\mathbf{W} \rightarrow \mathbf{R}$), the reordering of a write followed by another write on a different memory location, ($\mathbf{W} \rightarrow \mathbf{W}$), and it permits a processor to see it "own writes early".

From the point of view of the formalization technique, the fact that our semantics is operational, and that computations can be directly generated considering only the syntax of the program, turns to be invaluable. The standard concepts of *data dependencies* and *control dependencies*, are directly extracted, or observed, from the generated computations, rather than being posed a priori as it is the case with axiomatic formalization of memory models. This is a major advantage of our approach. Moreover, we can use standard "true concurrency" techniques, like the equivalence by permutations of [Berry and Lévy, 1979] that is at the core of our proof that the *fundamental property* of relaxed memory models [Saraswat et al., 2007] holds for the semantics we consider.

However, the framework of Chapter 2 does not allow all possible execution relaxations. In particular, the reordering of reads with respect to a subsequent memory access is not considered, which motivates the framework of Chapter 3.

In Chapter 3 we propose an operational framework for the characterization of speculative computation. In this setting we capture the behaviors induced by several speculative techniques such as instruction level parallelism [Hennessy and Patterson, 1996; Fisher, 1981] and branch prediction [Smith, 1981] among

many more. In fact, these speculative techniques lead to behaviors similar to those commonly found in the relaxed memory models area, and hence, our formalization enables the description of relaxed memory models as well. Although the framework of Chapter 3 is very general, not *all* relaxed consistency models can be formalized by means of speculations. In particular, the model assumes a single global store, to which all writes are eventually updated. Models such as Release Consistency Gharachorloo et al. [1990] go beyond this assumption, and thus cannot be modeled by our framework.

One of the key contributions of Chapter 3 is the definition of the *validity* property, which characterizes the notion of correct speculative computations, which is in general implicit in the literature of speculative computations.

In terms of memory model relaxations, the speculative semantics we develop allows the reordering of a write followed by a read on a different memory location ($\mathbf{W} \rightarrow \mathbf{R}$), or a write followed by another write ($\mathbf{W} \rightarrow \mathbf{W}$) on a different memory location, and the the reading of "own writes early"; all relaxations allowed by the formalization of Chapter 2. Furthermore, the speculative framework allows the reordering of a read followed by another read ($\mathbf{R} \rightarrow \mathbf{R}$) regardless of the locations involved, and the reordering of a read with respect to a subsequent write on a different location ($\mathbf{R} \rightarrow \mathbf{W}$). These two last relaxations where not permitted in the framework of write buffers. We can see then, that the framework of speculations is more general than the one with write buffers.

Regarding the programming language, we considered in Chapter 3 two different synchronization disciplines (or synchronization models according to [Adve and Hill, 1990]) that use different syntax. The first language supports high-level locks to guarantee mutual exclusion. The second language, which is at a lower level, provides only barriers and a simple compare-and-swap construct. For both of these languages we identify robustness properties that guarantee that speculative computations of robust programs exhibit sequentially consistent behaviors.

Finally, in Chapter 4 we exercise the frameworks of Chapters 2 and 3 by instantiating them to deal with the Sparc family of memory models [SPARC, 1994]. The TSO and PSO variations of Sparc can be easily encoded in the framework of write buffers, and all the Sparc memory models can be encoded with the framework of speculative computations. Interestingly, having formalizations of TSO and PSO in both frameworks allows for their comparison. The main result of Chapter 4 is a correspondence result between these two formalizations. In particular, our proof of correspondence provides support to the claim, to be found in the Sparc architecture specification [SPARC, 1994], that the Sparc family of memory models supports write buffering.

5.1 Future Prospects

Some rather immediate research directions are considering the speculative framework of Chapter 3 with "stricter" semantics. We conjecture that disallowing the speculation of write events within the branch of a conditional construct suffices to ensure that checking whether a program is SDRF can be done simply by considering the sequentially consistent computations of the program (cf. the DRF guarantee). However, this restriction applies only to a high-level language, since the scope of the branch of a conditional construct might not be

obvious in a lower-level language. Moreover, we conjecture that if we further restrict the semantics of locks by disallowing instructions past a lock construct to be speculated (as discussed in Chapter 3) the DRF guarantee holds for this speculative semantics.

Another research directions that we plan to pursue is the formal comparison of our operational semantics with axiomatic ones like [Alglave et al., 2010] for example. To that end we could follow the approach of [Boudol and Castellani, 1994]. We think that "extracting" from the computations generated by our speculative framework the events and orders required to instantiate the definition of [Alglave et al., 2010; Sarkar et al., 2009], and finding a speculative computation that satisfies each valid instantiation of that definition should enable a correspondence result between these approaches.

All of the results presented in this thesis focus mainly on the semantical aspects of relaxed memory models, and leave the optimization concerns apart. In fact we attempt to explain in an operational way the behaviors of relaxed memory models rather than considering program transformations. A question to study is to which extent common compiler optimizations are sound with respect to our semantics. One can find inspiration for this research in the work of Ševčík [2009]. A different angle on the same subject is to identify how the speculative framework could (or should) be adapted to capture standard (desirable) compiler optimizations. Perhaps, considering sound optimizations for the subset of programs satisfying our robustness properties (for both languages) could suggest different robustness properties or novel optimizations. One can think of techniques such as lock elision [Rajwar and Goodman, 2001] for the language with locks, or minimal placement of barriers [Shasha and Snir, 1988] for the language with barriers. We think this research area is vast, and having an operational semantics to explain some optimizations is just a first step towards a theory of safe parallel program transformations.

Other possible research directions include the verification and analysis (which we have already started exploring) of programs running on relaxed memory models. In particular, verifying the correctness of parallel algorithms and data structures running on relaxed memory models requires new techniques for the cases in which the DRF guarantee turns to be insufficient. Relaxed memory model aware program logics, or other verification techniques are interesting candidates for future work. We think that having an operational semantics will prove to be of great help towards these goals.

Of course, the specialization of the framework we considered here to further architectures and memory models is part of our future research directions. Maybe describing with more precision the machine architectures could prove of interest for the specification and the use of multiprocessor architectures.

Finally, this thesis is based on the idea that having operational descriptions of relaxed memory models could shed some light on the nature of relaxed memory models and improve their understanding. We plan to pursue this line of research by considering the interaction of programming languages and relaxed memory models to provide support for safe parallelism, a subject that desperately needs a solution [Adve and Boehm, 2010].

Appendix A

Examples

This Appendix contains a list of the main examples discussed in this thesis and the page numbers in which the examples are discussed.

Dekker's mutual exclusion algorithm appears in pages 3 and 22.

Example 1.1 (Dekker)**.**

$$\begin{array}{lcl} \texttt{flag}_0 := \texttt{false}\,; & \| & \texttt{flag}_1 := \texttt{false}\,; \\ \textsf{if } \texttt{flag}_1 \textsf{ then} & \| & \textsf{if } \texttt{flag}_0 \textsf{ then} \\ \quad \textit{critical section 0}\,; & \| & \quad \textit{critical section 1}\,; \end{array}$$

The Safe Publication example appears in pages 9, 23 and 87.

Example 1.6 (Safe Publication)**.**

$$\begin{array}{lcl} \texttt{data} := 1; & \| & \texttt{while not flag do skip}; \\ \texttt{flag} := \texttt{true} & \| & \texttt{r} := \texttt{data} \end{array}$$

The following example illustrates the effects of the reordering of reads with respect to previous writes. It appears in pages 5, 25, 99 and 110.

Example 1.2 (Write Read Reordering)**.**

$$\begin{bmatrix} p := 1\,; \\ r_0 := (!q) \end{bmatrix} \quad \| \quad \begin{bmatrix} q := 1\,; \\ r_1 := (!p) \end{bmatrix}$$

The following example is similar to the previous one, where barriers are added to prevent the reordering of writes with subsequent reads. It appears in page 111.

Example 4.1.

$$\begin{bmatrix} p := 1\,; \\ \langle \texttt{wr}|\texttt{rd} \rangle\,; \\ r_0 := (!q) \end{bmatrix} \quad \| \quad \begin{bmatrix} q := 1\,; \\ \langle \texttt{wr}|\texttt{rd} \rangle\,; \\ r_1 := (!p) \end{bmatrix}$$

The example below puts in evidence the reordering of two write instructions on different references (assuming that reads cannot be reordered). It appears in pages 5, 25, 43, 91, 99, and 109.

Example 1.3 (Write Write Reordering)**.**

$$\begin{bmatrix} p := 1\,; \\ q := 1 \end{bmatrix} \quad \| \quad \begin{bmatrix} r_0 := (!q)\,; \\ r_1 := (!p) \end{bmatrix}$$

The following example is similar to the previous one where barriers have been added in the writing thread. Nonsequentially consistent behaviors are only possible if the read actions on the left thread can be reordered. The example appears in pages 54, 99, 109, and 117.

Example 2.40 (Read Read Reordering: Barrier)**.**

$$\left[\begin{array}{l} p := 1\,; \\ \langle \mathtt{wr}|\mathtt{wr} \rangle\,; \\ q := 1 \end{array} \right] \quad \| \quad \left[\begin{array}{l} r_0 := (!q)\,; \\ r_1 := (!p) \end{array} \right]$$

This example is used to illustrate the reordering of a read with respect to a subsequent write on a different memory location. It appears in pages 5, 78, 99 and 116.

Example 1.4 (Read Write Reordering)**.**

$$\begin{bmatrix} r_0 := (!q)\,; \\ p := 1 \end{bmatrix} \quad \| \quad \begin{bmatrix} r_1 := (!p)\,; \\ q := 1 \end{bmatrix}$$

The following program is used to illustrate the behaviors produced by write buffering that cannot simply be explained by the reordering of actions. It appears in pages 25 and 118.

Example 2.3 (Read Own Write Early)**.**

$$\left[\begin{array}{l} p := 1\,; \\ r_0 := (!p)\,; \\ r_1 := (!q) \end{array} \right] \quad \| \quad \left[\begin{array}{l} q := 1\,; \\ r_2 := (!q)\,; \\ r_3 := (!p) \end{array} \right]$$

The example that follows is similar to the previous and illustrates that data dependencies and $\langle \mathtt{rd}|\mathtt{rd} \rangle$ barriers are not enough to recover sequential consistency in the presence of write buffers. It appears in pages 76 and 118.

Example 3.19 (Read Own Write Early: Barrier)**.**

$$\left[\begin{array}{l} p := 1\,; \\ r_0 := (!p)\,; \\ \langle \mathtt{rd}|\mathtt{rd} \rangle\,; \\ r_1 := (!q) \end{array} \right] \quad \| \quad \left[\begin{array}{l} q := 1\,; \\ r_2 := (!q)\,; \\ \langle \mathtt{rd}|\mathtt{rd} \rangle\,; \\ r_3 := (!p) \end{array} \right]$$

This code corresponds to a common example allowed by many memory models. It appears in pages 44 and 78.

Example 2.24 (Independent Reads Independent Writes)**.**

$$\Big[p := 1\Big] \| \Big[q := 1\Big] \quad \| \quad \left[\begin{array}{l} r_0 := (!p)\,; \\ r_1 := (!q) \end{array} \right] \| \left[\begin{array}{l} r_2 := (!q)\,; \\ r_3 := (!p) \end{array} \right]$$

We use the following example to show that the speculative semantics of Chapter 3 does not satisfy the DRF guarantee. It appears in page 79.

Example 3.21 (Thin Air Values)**.**

$$\left[\begin{array}{l} p := \mathit{ff}\,; \\ (\mathsf{if}\ !p\ \mathsf{then}\ q := \mathit{tt}\ \mathsf{else}\ ()) \end{array}\right] \parallel \left[\begin{array}{l} q := \mathit{ff}\,; \\ (\mathsf{if}\ !q\ \mathsf{then}\ p := \mathit{tt}\ \mathsf{else}\ ()) \end{array}\right]$$

The following example appears in page 119 to illustrate the need of a global validity condition for the speculative semantics of the models in Chapter 4.

Example 4.7.

$$\left[\begin{array}{l} p := 1\,; \\ r_0 := (!q)\,; \\ r_1 := (!p) \end{array}\right] \quad \parallel \quad \left[\begin{array}{l} \mathsf{let}\ x = (!p)\ \mathsf{in}\ (\\ \qquad q := x\ ) \end{array}\right]$$

Bibliography

Abadi, M., Flanagan, C., and Freund, S. N. (2006). Types for safe locking: Static race detection for Java. *ACM Trans. Program. Lang. Syst.*, 28(2):207–255.

Adve, S. V. and Boehm, H.-J. (2010). Memory Models: A Case for Rethinking Parallel Languages and Hardware. In *Communications of the ACM (CACM)*.

Adve, S. V. and Gharachorloo, K. (1996). Shared Memory Consistency Models: A Tutorial. *Computer*, 29:66–76.

Adve, S. V. and Hill, M. D. (1990). Weak Ordering — a New Definition. In *ISCA '90: Proceedings of the 17th annual international symposium on Computer Architecture*, pages 2–14, New York, NY, USA. ACM.

Adve, S. V., Hill, M. D., Miller, B. P., and Netzer, R. H. B. (1991). Detecting data races on weak memory systems. *SIGARCH Comput. Archit. News*, 19(3):234–243.

Alglave, J., Maranget, L., Sarkar, S., and Sewell, P. (2010). Fences in Weak Memory Models. In *CAV*, pages 258–272.

AMD (2010). *AMD64 Architecture Programmer's Manual, Volume 2: System Programming, Revision 3.17*.

ARM (2008). *ARM Architecture Reference Manual, ARM-v7A & ARM-v7R Edition*.

Atig, M. F., Bouajjani, A., Burckhardt, S., and Musuvathi, M. (2010). On the Verification Problem for Weak Memory Models. In *POPL '10: Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 7–18, New York, NY, USA. ACM.

Bergan, T., Anderson, O., Devietti, J., Ceze, L., and Grossman, D. (2010). CoreDet: a compiler and runtime system for deterministic multithreaded execution. In *ASPLOS '10: Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, pages 53–64, New York, NY, USA. ACM.

Berry, G. and Lévy, J.-J. (1979). Minimal and Optimal Computations of Recursive Programs. *J. ACM*, 26(1):148–175.

Bocchino, Jr., R. L., Adve, V. S., and Adve, Sarita V. andSnir, M. (2009). Parallel Programming Must be Deterministic by Default. In *HotPar'09: Proceedings of the First USENIX conference on Hottopics in parallelism*, pages 4–4, Berkeley, CA, USA. USENIX Association.

Boehm, H.-J. (2007). Reordering Constraints for Pthread-Style Locks. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 173–182, New York, NY, USA. ACM.

Boehm, H.-J. (2009). Simple Thread Semantics Require Race Detection. draft.

Boehm, H.-J. and Adve, S. V. (2008). Foundations of the C++ concurrency memory model. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 68–78, New York, NY, USA. ACM.

Boudol, G. (1986). Computational Semantics of Term Rewriting Systems. *Algebraic methods in semantics*, pages 169–236.

Boudol, G. and Castellani, I. (1988). A Non-interleaving Semantics for CCS Based on Proved Transitions. *Fundamenta Informaticae*, XI:433–452.

Boudol, G. and Castellani, I. (1994). Flow Models of Distributed Computations: Three Equivalent Semantics for CCS. *Inf. Comput.*, 114(2):247–314.

Boudol, G. and Petri, G. (2009). Relaxed Memory Models: an Operational Approach. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 392–403, New York, NY, USA. ACM.

Boudol, G. and Petri, G. (2010). A Theory of Speculative Computation. In *ESOP*, pages 165–184. LNCS.

Boyland, J. (2008). An Operational Semantics including Volatile for Safe Concurrency. In Huisman, M., editor, *ECOOP 2008 Workshop on Formal Techniques for Java-like Programs*.

Briggs, F. A. (1979). Effects of Buffered Memory Requests in Multiprocessor Systems. *SIGSIM Simul. Dig.*, 11(1):73–81.

Brookes, S. (2007). A Semantics for Concurrent Separation Logic. *Theor. Comput. Sci.*, 375(1-3):227–270.

Burckhardt, S., Musuvathi, M., and Singh, V. (2010). Verifying Local Transformations on Relaxed Memory Models. In *CC*, pages 104–123.

Burton, F. W. (1985). Speculative Computation, Parallelism, and Functional Programming. In *IEEE Trans. on Computers, Vol. C-34, No. 12*, pages 1190–1193.

Cenciarelli, P. and Knapp, A. (2007). Personal communication.

Cenciarelli, P., Knapp, A., Reus, B., and Wirsing, M. (1999). An Event-Based Structural Operational Semantics of Multi-Threaded Java. In *Formal Syntax and Semantics of Java*, pages 157–200, London, UK. Springer-Verlag.

Cenciarelli, P., Knapp, A., and Sibilio, E. (2007). The Java Memory Model: Operationally, Denotationally,Axiomatically. In *ESOP*, volume 4421 of *LNCS*, pages 331–346. Springer.

Clarke, E. M., Grumberg, O., and Peled, D. A. (1999). *Model Checking*. The MIT Press.

Collier, W. W. (1992). *Reasoning about Parallel Architectures*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Compaq (2002). *Alpha Architecture Reference Manual, Fourth Edition*.

Dubois, M., Scheurich, C., and Briggs, F. (1998). Memory Access Buffering in Multiprocessors. In *ISCA '98: 25 years of the international symposia on Computer architecture (selected papers)*, pages 320–328, New York, NY, USA. ACM.

Felleisen, M. and Hieb, R. (1992). The Revised Report on The Syntactic Theories of Sequential Control and State. *Theor. Comput. Sci.*, 103(2):235–271.

Fisher, J. A. (1981). Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Trans. Comput.*, 30(7):478–490.

Flanagan, C. and Felleisen, M. (1995). The Semantics of Future and Its Use in Program Optimization. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 209–220, New York, NY, USA. ACM.

Flanagan, C., Sabry, A., Duba, B. F., and Felleisen, M. (1993). The Essence of Compiling with Continuations. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 237–247, New York, NY, USA. ACM.

Gao, G. R. and Sarkar, V. (1997). On the Importance of an End-To-End View of Memory Consistency in Future Computer Systems. In *ISHPC '97: Proceedings of the International Symposium on High Performance Computing*, pages 30–41, London, UK. Springer-Verlag.

Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A., and Hennessy, J. (1990). Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *ISCA '90: Proceedings of the 17th annual international symposium on Computer Architecture*, pages 15–26, New York, NY, USA. ACM.

Hennessy, J. L. and Patterson, D. A. (1996). *Computer Architecture: a Quantitative Approach, 2nd edition*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Hill, M. D. (1998). Multiprocessors Should Support Simple Memory-Consistency Models. *Computer*, 31(8):28–34.

Huet, G. (1980). Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems: Abstract Properties and Applications to Term Rewriting Systems. *J. ACM*, 27(4):797–821.

Huisman, M. and Petri, G. (2007). The Java Memory Model: a Formal Explanation. *Verification and Analysis of Multi-threaded Java-like Programs (VAMP'07)*.

Huisman, M. and Petri, G. (2008). BicolanoMT: a formalization of multi-threaded Java at bytecode level. In *Bytecode 2008*, Electronic Notes in Theoretical Computer Science.

Intel Corporation (2007). Intel 64 Architecture Memory Ordering White Paper. SKU 318147-001.

Jagadeesan, R., Pitcher, C., and Riely, J. (2010). Generative Operational Semantics for Relaxed Memory Models. In *ESOP*, pages 307–326.

Jones, C. B. (1983). Tentative Steps Toward a Development Method for Interfering Programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619.

Knight, T. (1986). An Architecture for Mostly Functional Languages. In *LFP '86: Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 105–112, New York, NY, USA. ACM.

Kroft, D. (1981). Lockup-Free Instruction Fetch/Prefetch Cache Organization. In *ISCA '81: Proceedings of the 8th annual symposium on Computer Architecture*, pages 81–87, Los Alamitos, CA, USA. IEEE Computer Society Press.

Lamport, L. (1977). Proving the Correctness of Multiprocess Programs. *IEEE Trans. Softw. Eng.*, 3(2):125–143.

Lamport, L. (1978). Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565.

Lamport, L. (1979). How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programm. *IEEE Trans. Comput.*, 28(9):690–691.

Lamport, L. (1987). A fast Mutual Exclusion Algorithm. *ACM Trans. Comput. Syst.*, 5:1–11.

Ledgard, H. (1983). *Reference Manual for the ADA Programming Language*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Lévy, J.-J. (1980). Optimal Reductions in The Lambda Calculus. *in To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism (J. P. Seldin, J. R. Hindley, Eds), Academic Press*, pages 159–191.

Manson, J. (2004). *The Java Memory Model*. PhD thesis, University of Maryland.

Manson, J., Pugh, W., and Adve, S. V. (2005). The Java memory model. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 378–391, New York, NY, USA. ACM.

Marino, D., Singh, A., Millstein, T., Musuvathi, M., and Narayanasamy, S. (2010). DRFx: a simple and efficient memory model for concurrent programming languages. *SIGPLAN Not.*, 45(6):351–362.

Naik, M., Aiken, A., and Whaley, J. (2006). Effective Static Race Detection for Java. *SIGPLAN Not.*, 41(6):308–319.

Owens, S., Sarkar, S., and Sewell, P. (2009). A Better x86 Memory Model: x86-TSO. In *TPHOLs '09: Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, pages 391–407, Berlin, Heidelberg. Springer-Verlag.

Owicki, S. and Gries, D. (1976). Verifying Properties of Parallel Programs: an Axiomatic Approach. *Commun. ACM*, 19(5):279–285.

Plotkin, G. D. (1975). Call-by-name, Call-by-value and the Lambda-Calculus. *Theoretical Computer Science*, 1(2):125–159.

Plotkin, G. D. (1981). A Structural Approach to Operational Semantics. *DAIMI FN-19*.

PowerPC (2009). Power Instruction Set Architecture, v2.06.

Rajwar, R. and Goodman, J. R. (2001). Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *MICRO 34: Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 294–305, Washington, DC, USA. IEEE Computer Society.

Reynolds, J. C. (2004). Toward a Grainless Semantics for Shared-Variable Concurrency. In *FSTTCS*, pages 35–48.

Saraswat, V. A. (2004). Concurrent Constraint-Based Memory Machines: A Framework for Java Memory Models. In *ASIAN*, pages 494–508.

Saraswat, V. A., Jagadeesan, R., Michael, M., and von Praun, C. (2007). A Theory of Memory Models. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 161–172, New York, NY, USA. ACM.

Sarkar, S., Sewell, P., Nardelli, F. Z., Owens, S., Ridge, T., Braibant, T., Myreen, M. O., and Alglave, J. (2009). The Semantics of x86-CC Multiprocessor Machine Code. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 379–391, New York, NY, USA. ACM.

Sewell, P., Sarkar, S., Owens, S., Nardelli, F. Z., and Myreen, M. O. (2010). x86-TSO: a Rigorous and Usable Programmer's Model for x86 Multiprocessors. *Commun. ACM*, 53(7):89–97.

Shasha, D. and Snir, M. (1988). Efficient and Correct Execution of Parallel Programs that Share Memory. *ACM Trans. Program. Lang. Syst.*, 10(2):282–312.

Shen, X., Arvind, and Rudolph, L. (1999). Commit-Reconcile & Fences (CRF): a New Memory Model for Architects and Compiler Writers. *SIGARCH Comput. Archit. News*, 27(2):150–161.

Smith, J. E. (1981). A Study of Branch Prediction Strategies. In *ISCA '81: Proceedings of the 8th annual symposium on Computer Architecture*, pages 135–148, Los Alamitos, CA, USA. IEEE Computer Society Press.

SPARC, Inc. CORPORATE. (1994). *The SPARC Architecture Manual (version 9)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Ševčík, J. (2009). *Program Transformations in Weak Memory Models*. PhD thesis, The University of Edinburgh.

Ševčík, J. and Aspinall, D. (2008). On Validity of Program Transformations in the Java Memory Model. In *ECOOP '08: Proceedings of the 22nd European conference on Object-Oriented Programming*, pages 27–51, Berlin, Heidelberg. Springer-Verlag.

Zappa Nardelli, F., Sewell, P., Ševčík, J., Sarkar, S., Owens, S., Maranget, L., Batty, M., and Alglave, J. (2009). Relaxed memory models must be rigorous. In *(EC)* ^{2}*. CAV 2009 Workshop*, Grenoble, France.