УДК 004.421.6

A Survey of Programming Language Memory Models

© 2021 г. Evgenii Moiseenko[*,‡], Anton Podkopaev[†‡], Dmitrii Koznov[*]
[*] *Saint Petersburg State University*
198504 Saint Petersburg, Petergof, Universitetskii ave. 28
[†] *HSE University*
194100 Saint Petersburg, Kantemirovskaya str. 3 b. 1
[‡] *JetBrains Research*
197342 Saint Petersburg, Kantemirovskaya str. 2
E-mail: e.moiseenko@2012.spbu.ru, apodkopaev@hse.ru, d.koznov@spbu.ru

Поступила в редакцию 17.05.2021

Abstract

A memory model defines the semantics of concurrent programs operating on a shared memory. The most well-known and intuitive memory model, sequential consistency, is too *strong* for modern languages as it forbids many outcomes observable on modern hardware as a result of compiler and CPU optimizations. This gave rise to so-called *weak* or *relaxed* memory models. In recent years dozens of (weak) memory models for programming languages were proposed making different compromises with respect to programmability and the optimization potential. The goal of this paper is to survey and classify these models as well as to provide practical recommendations for language and compiler designers regarding a choice of a memory model.

To achieve this goal we picked over 2000 research items from Google Scholar with keywords “Relaxed Memory Models”, “Weak Memory Models”, and “Weak Memory Consistency”. Then, we narrowed down this list to 40 papers having as a contribution a programming language memory model. We divide these models to six main classes and analyze their properties and limitations. We conclude with a discussion on how a choice of a memory model is affected by desired features of a language and suggest several possible directions for future researh in the field of weak memory models.

1 Introduction

The main challenge in concurrent programming is to establish proper synchronization between threads executed in parallel. Usually it is done with the help of synchronization primitives provided by a programming language or libraries, for example locks, barriers, channels, *etc.* Sometimes, however, the usage of these primitives is impossible or undesirable. Examples of such cases are the implementation of synchronization primitives themselves or lock-free data structures. In these cases one has to resort to lower-level programming and use shared variables. At this point things get complicated.

Let us consider a concrete example. Here is a simplified version of Dekker's Lock [1]:

```
x := 1                      ||  y := 1
r1 := y                     ||  r2 := x
if r1 = 0 {                 ||  if r2 = 0 {
    // critical section     ||      // critical section
}                           ||  }
```

(Dekker's Lock)

In this program, two threads compete to enter the critical section. In order to indicate their intention, the threads set variables x and y correspondingly.[1] The one who manages to set the variable first and

[1]We distinguish shared variables (denoted as x, y, z) and thread local registers (denoted as r_1, r_2, r_3, *etc.*).

read the other variable before it is set enters the critical section. The algorithm relies on the fact that both threads cannot read value `0`.[2] Otherwise, the two threads would have been able to enter the critical section simultaneously, thus breaking the correctness of the algorithm.

Indeed, running this program on a multi-core system, one would expect to see one of the following outcomes: $[r_1 = 0, r_2 = 1]$, $[r_1 = 1, r_2 = 0]$, or $[r_1 = 1, r_2 = 1]$. These outcomes are *sequential consistent* [2] meaning that they may be obtained via a sequential execution of some interleaving of the threads' instructions.

However, not all behaviors which are observable on real concurrent systems are sequentially consistent. For example, if one ports Dekker's Lock from the pseudo code to the C language, compile it with the GCC compiler, and run on a processor from the x86/x64 family, they may observe non sequentially consistent outcome $[r_1 = 0, r_2 = 0]$. Such outcomes are called *weak*.

Weak outcomes appear because of compiler and CPU optimizations. For example, given the Dekker's Lock, the optimizer may observe that the store to x and the load from y in the left thread are independent instructions and thus they can be reordered (this optimization is valid for single-threaded programs). For the optimized program, the outcome $[r_1 = 0, r_2 = 0]$ is sequentially consistent.

The exact set of allowed outcomes for a given program is defined by a semantics of a concurrent system, or a *memory model*. The memory model permitting only sequentially consistent outcomes is called *sequential consistency* (SC). Memory models admitting weak behaviors are called *weak memory models*.

Neither modern hardware, nor programming languages guarantee sequential consistency since this model forbids many important optimizations. The main question then is how *weak* their memory models should be, *i.e.,* how big is the set of allowed weak behaviors for a given program. A stronger model allows less behaviors, thus giving more guarantees to a programmer and simplifying reasoning about programs, but a weaker model permits more optimizations, thus allowing a compiler to produce more efficient code.

It turns out that this question is challenging especially in the context of programming language (PL) memory models. Thus over the last two decades a plenty of memory models for various languages have been proposed, *e.g.,* for Java [3, 4], C/C++ [5], LLVM [6], JavaScript [7], OCaml [3], Haskell [8], *etc.* These models have different design goals, trade-offs, and limitations. Moreover, the research on weak memory models continues to develop rapidly. According to our findings, in the last decade at least 50 papers on the subject are published each year.[3] For those unfamiliar with all the subtleties of weak memory models, it is hard to navigate in this large zoo. Despite the long history of the field and recent progress made, there is no single source that summarizes the prior knowledge and give a comparison of different memory models of programming languages. The aim of this paper is to close this gap.

We provide an overview of existing approaches to programming languages' memory models, discuss their design choices, trade-offs, and limitations. Besides that, we compare existing memory models in terms of what optimizations opportunities and what guarantees for reasoning they provide.

We hope that our work will be useful to programming language researchers who want to dive into the theory of weakly consistent memory models, and also to system-level developers, who are working on new programming languages, compilers, or virtual machines, and have to choose a memory model for their system.

The rest of the paper is organized as follows. In §2 we overview related work. In §3 we describe the methodology of our study. In §4 we introduce common criteria of programming language memory models, namely optimality of compilation mappings, soundness of program transformations, and provided reasoning guarantees. Next, in §5 we explain how we compare memory models by these criteria. In §6 we present a classification of memory models based on their properties, and discuss each class. Additionally, in appendix A we further describe each particular memory model considered in our study. In §7 we present a short guide on

[2]From here and through the rest of the paper we assume that all variables are initialized with zeros, unless we explicitly state otherwise

[3]This claim is supported by our data acquired from Google Scholar, see §3 for details

the design of memory model for researchers and system developers and, as an example of using the guide, propose a memory model for the Kotlin[4] language. Finally, §8 concludes and outlines possible directions for future research in the field.

2 Related Work

Weak memory models can be partitioned into two significantly different subclasses: models for hardware architectures and models for programming languages. The main difference between them is that memory models for programming languages are expected to support compiler optimizations and may need to support compilation to different architectures.

To this day, hardware weak memory models are relatively well-studied and understood. All major architectures have formally defined memory models: x86 [9], IBM POWER [10–12]), Arm [10, 12–15]) and RISC-V [14]. Among those, x86 and POWER architectures have stable memory models which did not change in the last years, whereas the Arm memory model changed with transition from Armv7 [12] to Armv8 [14] to accommodate new instructions for shared memory access. RISC-V[5] was introduced in 2010 and recently adopted a memory model [14] which is almost the same as the Armv8 model.

All of the aforementioned models have representations in the framework of *declarative* (or *axiomatic*) memory models [12]. This framework became a standard for defining weak memory models, and it has tools for testing and verification [12]. It is also used for defining some programming language memory models: C/C++ [5], JavaScript [7], Java [3] and many others. However, the framework does not fully solve the problem for programming languages like C/C++ (even though it is currently used for defining C/C++ memory models in their standards [5]) which are oriented on zero cost abstraction over architectures like Arm and POWER and support for compiler optimizations which may eliminate syntactic dependencies (for example, constant folding).

[4]https://kotlinlang.org/

[5]https://riscv.org/

The problem is that the framework does not allow one to properly distinguish real (*semantic*) dependencies between instructions in programs from fake ones. For example, there is a real dependency between instructions in the snippet on the left and a fake dependency between instructions on the right:

$$\begin{array}{rcl|rcl} r & := & x & r & := & x \\ y & := & r & y & := & r * 0 \end{array}$$

For hardware models, it is not a problem since they respect all syntactic dependencies, whereas, in the case of a programming language memory model, if we want to support a compiler optimization which replaces $r * 0$ with 0 in the code on the right, we have to either distinguish them and respect only real dependencies or ignore dependencies completely (as the model of C/C++ does). Ignoring of dependencies in combination with support of *load-buffering behavior* of Arm and POWER leads to notorious *Out Of Thin-Air* (OOTA) executions [16] which break even basic reasoning principles about programs (discussed in more detail in §4.3.4).

Many memory models using significantly different approaches and frameworks were proposed to solve the problem without a performance penalty: Java Memory Model (JMM) [3], Promising semantics [17, 18], Weakestmo [19], Modular Relaxed Dependencies (MRD) [20]. Others like RC11 [21] and the OCaml memory model [22] decided to solve the problem and provide more guarantees to a developer at the cost of some slowdown [23].

Even though dozens of memory models with different compromises and features were proposed for programming languages, there are no detailed surveys of them to the best of our knowledge. That motivated our work on this paper.

3 Methodology

The main purpose of our work was to study trade-offs in the design of a memory model for a programming language. Stronger memory models give more reasoning guarantees to programmers, while weaker models provide more optimization opportunities. We wanted to answer the following research question.

- How reasoning guarantees provided by a memory model to an end-user of a programming language narrow optimization opportunities for a language implementation?

To answer this question, we consulted the existing research studies in the field of programming language memory models. Our goal was to identify existing proposed memory models and classify them.

In order to compare memory models we used standard criteria developed in the literature.

C.1 An *optimality of compilation scheme.* A language with a memory model supporting optimal compilation schemes can be efficiently implemented on modern hardware. Contrary, the usage of nonoptimal compilation mappings induces a slowdown during an execution of a program, but can prevent an apperance of certain weak behaviors permitted by the hardware.

C.2 A *soundness of common program transformations.* During the optimiztions passes a compiler of a programming language performs various source-to-source transformations. The more transformations a memory model of the language supports; the more compiler optimizations are potentially applicable to programs written in this language.

C.3 Support of various *reasoning guarantees*, which simplify the reasoning about the correctness of concurrent programs written in a given programming language.

In order to select memory models for our study, we performed the following search procedure. On the *first stage*, we manually selected 10 peer-reviewed research papers [3, 5, 7, 17, 19–22, 24, 25] whose main contribution was a proposal of a new PL weak memory model, and which were presented at highly ranked programming language conferences, such as “Symposium on Principles of Programming Languages” (POPL), “Conference on Programming Language Design and Implementation” (PLDI), and others. We then took the list of keywords from these papers. From this list of keywords we manually excluded those that were either too broad or too narrow. As a result we get three keyword phrases:

- Relaxed Memory Models;
- Weak Memory Models;
- Weak Memory Consistency.

On the *second stage* we used these phrases as search queries for the Google Scholar[6] search engine. For each query we took first 1000 entries from the search result.[7] We got a list of 2493 research items in total. As a sanity check, we verified that each of the 10 initially selected papers was contained in the selection.

On the *third stage* we removed from the selection duplicates and non-peer-reviewed papers. Also we decided to remove technical reports, theses, non-English publications, as well as short papers (up to 4 page long). After this stage 1077 research items have left.

Next, on the *fourth stage*, we further filtered the selection by consulting titles and abstracts of the papers. We included only the papers which are directly related to the topic of PL memory models and whose main focus is memory models themselves, as opposed to papers that only use established results about PL memory models, or papers related to adjacent topics. In particular, we filtered out papers related to the following adjacent topics:

- memory models of hardware, heterogeneous systems, and distributed systems;
- semantics of transactions and persistency;
- verification techniques for weak memory.

As a result we got 105 research items.

Finally, on the *fifth stage*, we carefully examined the remaining articles. In our final selection, we have included only those whose contribution was claimed to include:

- a new PL memory model;
- a study of an existing PL memory model;
- a refinement of an existing PL memory model.

In the end we got 40 research papers.

[6]https://scholar.google.com/

[7]All search queries were performed on 24 September 2020

4 Criteria for Memory Models

In this section we will have a closer look on criteria for programming language memory models, namely optimality of compilation scheme **C.1**, soundness of common program transformations **C.2**, and provided reasoning guarantees **C.3**. The criteria are bound to common programming primitives provided by the shared memory abstraction. Thus, we first introduce these programming primitives.

Programming Primitives A memory model defines the semantics of the shared memory in the presence of concurrently executing threads. The shared memory consists of individual variables, each having a unique address.[8] Threads access these variables by performing loads or stores.[9]

Most programming language memory models distinguish *non-atomic* (sometimes also called *plain*) and *atomic* variables. The former generally should not be accessed concurrently from parallel threads. Depending on the particular programming language, concurrent accesses to non-atomic variables can be either prohibited by a type-system (*e.g.,* Haskell [8, 26], Rust [27]), have undefined behavior (*e.g.,* C/C++ [5, 28]), or being defined but have very weak semantics with almost no guarantees on the order in which concurrent threads can observe these accesses (*e.g.,* Java [3]).

In turn, atomics are designed for concurrent accesses. Some memory models further distinguish several kinds of accesses to atomic variables. In these models the accesses to shared memory are annotated by so-called *access modes.* For example, the C/C++ model (and a later revision of the Java [4] model), distinguish three modes: *relaxed* (*opaque* in Java terminology), *acquire/release*, and *sequentially consistent* (*volatile* in Java). They are denoted as `rlx`, `acq`, `rel`, and `sc` correspondingly. Note that `acq` mode is only applicable to load operations, while `rel` is only applicable to store operations. Non-atomic accesses are often considered to be the fourth access mode `na`.

The access modes are ordered by the guarantees they provide in exchange of optimization opportunities, as the following diagram shows.

$$\begin{tikzcd}
 & & \texttt{rel} \arrow[dr, phantom, "\sqsubset" sloped] & \\
\texttt{na} \arrow[r, phantom, "\sqsubset"] & \texttt{rlx} \arrow[ur, phantom, "\sqsubset" sloped] \arrow[dr, phantom, "\sqsubset" sloped] & & \texttt{sc} \\
 & & \texttt{acq} \arrow[ur, phantom, "\sqsubset" sloped] &
\end{tikzcd}$$

On the one end of the spectrum are sequentially consistent accesses. They guarantee to restore the sequentially consistent semantics, if used properly (see §4.3.1 for details). Non-atomic accesses, as we have already discussed, give little guarantee. Relaxed accesses also have very weak semantics, usually they provide only the *coherence* property (see §4.3.2 for details). Finally, in the middle there are the acquire/release accesses. They are designed to support the message passing idiom [29]. A thread sends the message by performing a release write, another thread expecting a message can perform an acquire read. If the acquire read observes the release write, the two threads synchronize their views on shared memory.

Memory models also provide atomic *read-modify-write* operations. These include *compare-and-swap*, *exchange*, and variations of atomic increment, *e.g., fetch-and-add*, *fetch-and-sub*, *etc.* Compare-and-swap (CAS) operation takes a shared variable, expected and desired values. It reads from the variable and compares the result with the expected value. If the two are equal, it substitutes the value of the variable to the desired value. In either case, the value read from the variable is returned as a result. Note that the above operations are guaranteed to be performed atomically, no other write to the shared variable can happen in-between the read and write parts of CAS. Exchange operation atomically replaces the value of the variable and returns the previously held value. Fetch-and-add and similar primitives perform the operation (addition, subtraction, *etc.*) atomically and unconditionally, returning the content of the shared variable prior to modification.

Locks sometimes considered to be a part of a memory model [3], as well as memory fence operations [5], which are related to hardware fence instructions (see §4.1 for details).

Finally, a memory model can treat the shared memory not as a set of disjoint typed variables, but rather as a raw byte sequence, and permit so-called

[8]Throughout the rest of paper, we use terms memory address and memory location interchangeably

[9]We use terms load/stores and reads/writes interchangeably

mixed-size concurrent accesses [30]. For example, in a mixed-size model it is allowed for an 8 byte load instruction to read from two concurrent adjacent 4 byte stores.

4.1 Compilation Scheme

We next consider the first criterion **C.1** for programming language memory models—optimality of the compilation scheme. A *compilation scheme* is a mapping of programming language primitives into instructions of particular hardware architecture. In our setting we consider the primitives mentioned in §4. The hardware architectures provide a similar set of instructions which usually contain plain load and stores,[10] read-modify-write operations, and also various memory fences.

A compilation scheme should be *sound.* In the context of this paper it means that a set of outcomes permitted by the hardware memory model for a compiled program should be a subset of outcomes permitted by the programming language model for the original program.

Let us consider an example. The program SB below is a variant of Dekker's Lock from §1.

$$\begin{array}{l|l} x := 1 & y := 1 \\ r_1 := y & r_2 := x \end{array} \qquad \text{(SB)}$$

Assume that the memory model of the programming language is sequential consistency, and it should be compiled to x86 hardware. If one would compile all loads and stores to plain load and store instructions of x86,[11] the outcome $[r_1 = 0, r_2 = 0]$ would be allowed for the compiled program (and it can actually be observed in practice), since the memory model of x86 permits this outcome. It can be obtained as a result of *store buffering* optimization (hence the name of the program SB). The store $x := 1$ can be buffered and executed after all other instructions of the program. Yet the outcome $[r_1 = 0, r_2 = 0]$ is not sequentially consistent. Therefore the proposed compilation scheme, which maps all loads and stores to the plain load and stores is *unsound.* Unsoundness of a compilation scheme has dramatic consequences as it may break the correctness of a program.

[10]Some architectures also provide various types of load and stores matching the access modes annotations, *e.g.,* `lda` — load acquire, and `stl` — store release on Armv8.

[11]On x86 `MOV` instruction is used as both plain load from memory and plain store to memory instructions.

A sound compilation scheme for the sequential consistency can compile a store as a plain store followed by the `mfence` instruction [5, 9] as demonstrated below:

$$\begin{array}{l|l} x := 1 & y := 1 \\ \mathsf{mfence} & \mathsf{mfence} \\ r_1 := y & r_2 := x \end{array} \qquad \text{(SB+MFENCE)}$$

The `mfence` is a special memory fence instruction of x86 architecture that flushes the store buffer of the thread. For the program SB+MFENCE the outcome $[r_1 = 0, r_2 = 0]$ is forbidden by the x86 memory model.

Although the modified compilation scheme is sound for SC, it is not *optimal* [31], in a sense that it requires to use memory fence instructions, which usually induce a performance penalty of about 10-30% [32,33] (see appendix A.1 for details). Unfortunately, it is not possible to have compilation mapping to modern hardware architectures for the SC model which is both *sound and optimal.* This fact makes the SC memory model unsuitable for high-performance programming languages and serves as the stimulu for weakening of memory models.

In this paper, when speaking about compilation schemes, we will consider the following hardware memory models: x86, Armv7, Armv8, and POWER, for two main reasons. First, these hardware architectures are the most widespread today. Second, they have received a lot of attention from the research community recently. As a result of this effort, there were developed rigorous formal specifications of these models [9,11,14,15].

4.2 Program Transformations

The next criterion **C.2** for memory models is the soundness of program transformations, which are source-to-source transformations of code which applied during optimization passes of a compiler.

Sound transformations should preserve the semantics of a program. In our context, similarly to the soundness of compilation scheme, it means that a set of outcomes of the transformed program should be a subset of outcomes of the original one.

Going back to the SB example, assume the sequential consistency model again and consider a transformation that reorders the store instruction past the following load instructions in the left thread, assuming the load and store operate on the disjoint memory locations:

$$\begin{array}{c||c} x := 1 & y := 1 \\ r_1 := y & r_2 := x \end{array} \rightsquigarrow \begin{array}{c||c} r_1 := y & y := 1 \\ x := 1 & r_2 := x \end{array}$$

For the transformed version of the program (on the right), the outcome $[r_1 = 0, r_2 = 0]$ is sequentially consistent. Yet for the original one (on the left) it is not. It means that the aforementioned program transformation is unsound for SC.

We next present a list of various program transformations considered in the literature on weak memory models with a short description of each one. Note that the list is far from being complete regarding to transformations used in optimizing compilers [34]. For example, it lacks common loop optimizations, because the theory of relaxed memory models still struggles with problems of liveness properties [35], needed for studying these transformations formally.

The transformations we consider can be split into two subcategories: *local* and *global.* Local transformations rewrite a small piece of code within a single thread. Global transformations may need to consider a whole program (or a large part of it) spanning multiple threads in order to perform a rewriting.

4.2.1 Local Transformations

Reordering of Independent Instructions This transformation reorders two adjacent independent memory accessing instructions operating on different memory locations. Depending on a particular pair of instructions it can be further split into store/load, store/store, load/load, and load/store reorderings.

$$\begin{array}{lcll} x := v;\ r := y & \rightsquigarrow & r := y;\ x := v & \text{store/load} \\ x := v;\ y := u & \rightsquigarrow & y := u;\ x := v & \text{store/store} \\ r := x;\ s := y & \rightsquigarrow & s := y;\ r := x & \text{load/load} \\ r := x;\ y := v & \rightsquigarrow & y := v;\ r := x & \text{load/store} \end{array}$$

Elimination of Redundant Access In a pair of two adjacent instructions accessing same memory location one of them can be eliminated if its effect is subsumed by another. For example, two stores writing to the same variable can be replaced by a single store. Similarly to the reorderings, there exist four kinds of eliminations depicted below.

$$\begin{array}{lcll} x := v;\ r := x & \rightsquigarrow & x := v;\ r := v & \text{store/load} \\ r := x;\ s := x & \rightsquigarrow & r := x;\ s := r & \text{load/load} \\ r := x;\ x := r & \rightsquigarrow & r := x & \text{load/store} \\ x := v;\ x := u & \rightsquigarrow & x := u & \text{store/store} \end{array}$$

Irrelevant Load Elimination Yet another elimination transformation which removes a load instruction if its result is never used.

$$r := x \ \rightsquigarrow\ \epsilon \quad \mid r \text{ is never used}$$

Speculative Load Introduction An inverse to the previous transformation, the load introduction inserts a load instruction in an arbitrary place of a program.

$$\epsilon \ \rightsquigarrow\ r := x \quad \mid r \text{ is never used}$$

It can be used in combination with the load/load elimination to move a load instruction out from one branch of a conditional:

$$\mathsf{if}\ (e)\ \mathsf{then}\ \{r := x\} \ \rightsquigarrow\ s := x;\ \mathsf{if}\ (e)\ \mathsf{then}\ \{r := s\} \quad \mid s \text{ is never used}$$

Roach Motel Reordering This class of reorderings moves memory access instructions into synchronization blocks. For example, a store can be moved past a lock acquisition. Intuitively, such reorderings can only increase synchronization of a program, which means that the transformed program should exhibit less non-determinism and have fewer outcomes.

Non-atomic accesses can be moved freely inside a critical section, *i.e.,* past a lock acquisition or prior a lock release. Besides that, a store can be moved after a `lock`, and load can be moved prior an `unlock`. Similar rules apply to reorderings around acquire and release accesses and fences, where an acquire operation behaves similarly to `lock`, and release operation similarly to `unlock`.

$$\begin{array}{rcl} r :=_{\mathtt{na}} x;\ \mathsf{lock}(l) & \rightsquigarrow & \mathsf{lock}(l);\ r :=_{\mathtt{na}} x \\ x :=_o v;\ \mathsf{lock}(l) & \rightsquigarrow & \mathsf{lock}(l);\ x :=_o v \\ \mathsf{unlock}(l);\ x :=_{\mathtt{na}} v & \rightsquigarrow & x :=_{\mathtt{na}} v;\ \mathsf{unlock}(l) \\ \mathsf{unlock}(l);\ r :=_o x & \rightsquigarrow & r :=_o x;\ \mathsf{unlock}(l) \end{array}$$

Strengthening Similarly to the roach motel reordering, the strengthening transformation increases synchronization by replacing an access mode of an operation by a stronger one. For example, a non-atomic access can be replaced by a sequentially consistent access:

$$\begin{array}{lll} r :=_o x & \leadsto \; r :=_{o'} x & \mid o \sqsubset o' \\ x :=_o v & \leadsto \; x :=_{o'} v & \mid o \sqsubset o' \end{array}$$

Trace Preserving Transformations This wide class contains all local transformations which do not change a set of traces of a thread [36]. Trace is a sequence of visible side-effects performed by a thread (loads and stores to shared memory are also viewed as side-effects). An example is the classic *constant folding* [34,37] transformation. Here is a particular example of the constant folding application:

$$x := 0 + v \;\leadsto\; x := v$$

Common Subexpression Elimination CSE is yet another classic transformation [34] which searches for instances of identical expressions and removes redundant computations. Here is an example:

$$r_1 := x + y;\ r_2 := x + y \;\leadsto\; r_1 := x + y;\ r_2 := r_1$$

4.2.2 Global Transformations

Register Promotion If a compiler can determine that a shared variable is accessed only from a single thread, it can replace the variable by a thread-local register.

$$\begin{array}{l} x := v;\ r := x \;\leadsto\; s := v;\ r := s \\ \qquad\qquad\qquad \mid \texttt{x} \text{ is not accessed from other threads} \\ \qquad\qquad\qquad \mid \texttt{s} \text{ is a fresh register} \end{array}$$

Thread Inlining Sequentialization or thread inlining is a transformation that merges two threads into one. Quite surprisingly, this seemingly harmless transformation is challenging for many memory models.

$$P \parallel Q \;\leadsto\; P\ ;\ Q$$

Value Range Based Transformations Transformations of this class can be applied if a program satisfies some invariant deduced by a global value-range analysis. For example, in a program below the conditional statement can be eliminated since a static analysis can deduce invariant $\mathsf{x} \geq 0$.

$$\left.\begin{array}{l} r_1 := x \\ \mathsf{if}\ (r_1 \geq 0)\ \mathsf{then} \\ \quad y := 1 \end{array}\right\Vert \begin{array}{l} r_2 := x \\ y := r_2 \end{array} \;\leadsto\; \left.\begin{array}{l} r_1 := x \\ y := 1 \end{array}\right\Vert \begin{array}{l} r_2 := x \\ y := r_2 \end{array}$$

4.3 Reasoning Guarantees

Finally, we discuss the third criterion **C.3**—reasoning guarantees provided by memory models.

4.3.1 DRF Theorems

When reasoning about concurrent code, most programmers assume sequentially consistent memory model. Of course, it would be improper to require from programmers to always keep in mind all the intricacy of weak memory models, as it only complicates an already difficult task of establishing the correctness of concurrent programs. The *data-race freedom* [3] property, DRF for short, is designed to solve this problem. It guarantees that well-synchronized programs have only sequentially consistent outcomes. In other words, it allows programmers to assume simpler sequentially consistent model if they properly use synchronization primitives.

Let us consider an example. Remember the SB program from §4.1. As we demonstrated, under a weak memory model this program can have the weak outcome $[r_1 = 0, r_2 = 0]$. Nevertheless, one can restore the SC semantics. One way to do this is to use locks, as the following listing demonstrates:

$$\left.\begin{array}{l} \mathsf{lock}(l) \\ x := 1 \\ r_1 := y \\ \mathsf{unlock}(l) \end{array}\right\Vert \begin{array}{l} \mathsf{lock}(l) \\ y := 1 \\ r_2 := x \\ \mathsf{unlock}(l) \end{array} \qquad \text{(SB+LOCK)}$$

A DRF compliant weak memory model should guarantee that this program has only sequentially consistent outcomes: $[r_1 = 0, r_2 = 1]$, $[r_1 = 1, r_2 = 0]$, or $[r_1 = 1, r_2 = 1]$.

Alternatively, if model provides `sc` access mode, a programmer can annotate all memory accesses by this mode to restore sequential consistency:

$$\begin{array}{l} x \ :=_{\mathsf{sc}}\ 1 \\ r_1\ :=_{\mathsf{sc}}\ y \end{array} \left\| \begin{array}{l} y\ :=_{\mathsf{sc}}\ 1 \\ r_2\ :=_{\mathsf{sc}}\ x \end{array} \right. \qquad \text{(SB+SC)}$$

More formally, DRF theorem for a weak model M states that a program has only sequentially consistent outcomes under M if it has no data-races under sequentially consistent memory model (or all accesses participating in such race are annotated by sc).

The DRF theorem allows one to reduce reasoning under a weak memory model to reasoning under the sequential consistency. It is sufficient to prove that a program has no data-races under the SC in order to derive that this program has only SC outcomes.

The DRF theorem in the formulation given above is sometimes called *external data-race freedom* (eDRF), in order to distinguish it from the *internal data-race freedom* (iDRF) [7, 38]. The latter guarantees the SC semantics for a program under weak model M only if the program has no races under **model M itself**. Note that the internal DRF gives a weaker guarantee compared to the external DRF. It does not allow to completely avoid the reasoning in term of the weak memory model, because one has to first show that the program is race-free under relaxed model. As we will demonstrate later (see §6.3) the internal DRF is a compromise for a certain class of memory models which does not admit the external DRF.

4.3.2 Coherence

As we demonstrated, memory models of modern hardware architectures do not provide the sequentially consistent semantics. Yet they usually provide a weaker property called *sequential consistency per location*, also known as *coherence* [12]. Following hardware models many programming language level memory models also provide this property.

The coherence property ensures that all stores to each particular location can be totally ordered and that the resulting order, the *coherence order*, reflects the order in which stores propagate from threads into the main memory. In particular, coherence implies that programs consisting only of accesses to a single memory location have sequentially consistent semantics. For example, consider the following program:

$$\begin{array}{l} x\ :=\ 1 \\ r_1\ :=\ x \end{array} \left\| \begin{array}{l} x\ :=\ 2 \\ r_2\ :=\ x \end{array} \right. \qquad \text{(COH)}$$

The coherence prescribes memory model to assign to this program only the sequentially consistent outcomes: $[r_1 = 1, r_2 = 2]$, $[r_1 = 1, r_2 = 1]$, or $[r_1 = 2, r_2 = 2]$. A non-coherent model additionally may permit the outcome $[r_1 = 2, r_2 = 1]$. For example, the Java memory model actually allows this outcome [3].

4.3.3 Undefined Behavior

As we already briefly mentioned, some memory models, *e.g.,* C/C++, treat racy programs as having *undefined behavior* [28] if at least one of the accesses participating in a race is a non-atomic access. In other words, for these programs any outcome is possible. This property is also sometimes called the *catch-fire semantics.*

The practical payoff of this approach is that it enables the optimal compilation scheme for non-atomic accesses and makes any sequentially valid transformation applicable to them. Indeed, effects of hardware and compiler optimizationz can only be observed due to racy accesses from concurrent threads. If such accesses are said to imply undefined behavior and give no guarantee, effects of these optimizations become indistinguishable.

4.3.4 Speculative Execution and Out of Thin-Air Values

In order to introduce the last two properties, we turn to an example:

$$\begin{array}{l} r_1\ :=\ x \\ y\ :=\ 1 \end{array} \left\| \begin{array}{l} r_2\ :=\ y \\ x\ :=\ r_2 \end{array} \right. \qquad \text{(LB)}$$

Assume a weak memory model admitting the outcome $[r_1 = 1, r_2 = 1]$ for this program. For example, hardware memory models of Armv7, Armv8, and POWER allow this outcome, and it can even be actually observed on some Armv7 machines [39].

The outcome $[r_1 = 1, r_2 = 1]$ cannot be obtained by some *in-order* execution of the program. To enable this kind of behaviors for programs, a memory model has to utilize some form of *speculative execution* [16, 40]. That is, during the execution, the load $r_1\ :=\ x$ needs to be buffered and the store $y := 1$ needs to be executed out of order (hence the name of the program LB — *load buffering*).

However, unrestricted speculations can lead to disruptive results. A store executed out of order can turn into a self-fulfilling prophecy [16]. Consider the following variation of the load buffering program.

$$\begin{array}{l||l} r_1 := x & r_2 := y \\ y := r_1 & x := r_2 \end{array} \qquad \text{(LB+data)}$$

Here, a hypothetical abstract machine can speculate to perform a store of value `1` into the variable `y` from the left thread, then read this value in the right thread, write it to the variable `x` and then read it back in the left thread closing the paradoxical causality cycle. The value `1` in the example above appears *out of thin-air* and then justifies itself leading to the confusing outcome $[r_1 = 1, r_2 = 1]$.

As we will see in §6, speculative execution is required to enable certain program transformations. However, speculations should be properly constrained in order to prevent thin-air values. In §6.4-§6.6 we will see how various memory models deal with this problem.

5 Comparison

We performed a comparison of the memory models found via the search procedure described in §3 by the criteria given in §4. A particular challenge of this comparison was the fact that consulted research papers often use different terminology, have incomplete information about models, and sometimes they even contradict each other. We tried to approach these challenges by the following means. First, we used consistent terminology to denote the properties of the memory models, as presented in §4. Second, we complemented the information about each particular memory model from different sources. If after this procedure some particular property was still unclear, we left it as unknown.

Based on our comparison of the memory models, we identified six classes of them: sequentially consistent models, models with total or partial order on stores, program order preserving models, syntactic dependency preserving models, semantic dependency preserving models, and models with out of thin-air values. The models from the same class have the similar compilation mappings, set of sound program transformations, and provided reasoning guarantees. We first present the result of our comparison on a per-class basis (see table 1 and §6), and then give a more detailed comparison with respect to individual models (see table 2 and appendix A). Thus we have an opportunity to first discuss common principles behind programming language relaxed memory models in general, and then dive deeper into the details of each particular model.

In both table 1 and table 2 we order the memory models by their strength. The strongest models are located at the top rows of the tables, while the weakest are at the bottom.

The columns of both tables correspond to the properties of the memory models. In order to be concise, we chose a binary classification for all properties, *i.e.,* the model is either said to satisfy a given property or not. We also split properties into several subgroups.

The first group is devoted to an optimality of compilation mappings to target hardware architectures. We classify a compilation scheme as either optimal or not, in the following sense. We chose the weakest possible access mode supported by a model and consider the compilation scheme for memory accesses annotated by this mode. For memory models that treat racy non-atomic accesses as undefined behavior, we consider the compilation mapping for the weakest atomic access mode that model provides. It is because the catch-fire semantics for racy non-atomics trivially permits the optimal compilation mapping (see §4.3.3). We say that the compilation scheme is *optimal* if accesses annotated by the chosen mode can be compiled just as plain load and store instructions of a given hardware architecture (*i.e.,* without use of memory fences or other auxiliary code).

The second group is dedicated to a soundness of various program transformations. The classification is also binary: a transformation is either sound or unsound in a given memory model (in the sense stated in §4.2). Again, to be concise, we do not consider all combinations of program transformations and memory access modes. Instead, we consider the weakest possible accesses which have fully defined semantics. We further

Class	# Models	Compilation				Transformations																			Reasoning				
						Local																Global							
		x86	POWER	Armv7	Armv8	Reordering				Elimination				ILE	SLI	RM	S	TP	CSE	RP	TI	VR	eDRF	COH	no-UB	In-Order	no-OOTA
						SL	SS	LL	LS	SL	SS	LL	LS														
Sequential Consistency	2	−	−	−	−	−	−	−	−	+	+	+	+	+	+	+	+	+	−	+	+	−	+	+	+	+	+
Total/Partial Store Order	2	+	−	−	−	+	+	−	−	+	+	+	−	+	+	−	−	+	−	−	−	−	+	+	+	+	+
Program Order Preserving	3	+	−	−	−	+	+	+	−	+	+	+	−	−	−	+	+	+	±	+	+	−	+	+	+	+	+
Syntax. Dep. Preserving	2	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	−	−	−	−	−	−	+	+	+	−	+
Semantic Dep. Preserving	7	+	+	+	+	+	+	+	+	+	+	+	±	+	+	+	+	+	±	+	−	+	+	+	+	−	+
Out of Thin-Air	5	+	+	+	+	+	+	+	+	+	+	+	−	−	−	+	+	+	−	−	+	−	−	+	+	−	−

Table 1.: Classes of memory models and their properties

split the transformations into global and local as in §4.2.

The third group corresponds to reasoning principles guaranteed by the model. In particular, we check whether a model provides the external DRF guarantee (see §4.3.1), whether it provides the coherence property (see §4.3.2), whether it has fully defined semantics for all types of accesses, *i.e.,* the model does not treat racy non-atomic accesses as undefined behavior (see §4.3.3), whether the model utilizes in-order execution (as opposed to speculative out-of-order executon), and whether it forbids out of thin-air values (see §4.3.4).

In table 2 each row corresponds to a specific memory model, denoted by its abbreviation, and thus each cell describes a particular property of that particular model. We marked a cell by + if the corresponding model satisfies the given property, and we marked it by − otherwise. If the property was not studied in the research papers, we color the cell in gray ⬜.

Each row of the table 1 corresponds to a class of memory models. We marked a cell by + if the majority of models in the given class satisfy the property. If less than the majority of models satisfy the property we mark the corresponding cell by ±. Finally, if none of the models satisfy the property, we mark the cell by −. Note that when counting the majority, we omit the unknowns. Also, if a given property was not studied in the context of some class of models (*i.e.,* in table 2 it is marked by gray color for all models in this class) in table 1 we mark the corresponding cell by −. That is, in table 1 symbols + and ± denote positive knowledge, while − denotes negative knowledge or an absence of information.

Besides tables 1 and 2 which describe properties of the memory models, we also present table 3 that provides a list of features supported by the models. In this table each row corresponds to a particular memory model. Columns correspond to supported features. In particular, we check what types of access modes are supported: non-atomic (NA), relaxed (RLX), release/acquire (RA), sequentially-consistent (SC); what types of fences are supported: release/acquire (F-RA) and sequentially-consistent (F-SC); whether the atomic read-modify-write operations are supported (RMW), whether the model handles locks explicitly (LK), and whether it supports mixed-size accesses (MIX).

6 Analysis

In this section we discuss each identified class of memory models in more detail. Based on the data from table 1, we derive a relationship between a compilation scheme optimality, a soundness of transformations and reasoning guarantees. In particular, we show how the support of some

Class	Model	Compilation				Transformations																	Reasoning				
						Local														Global							
		x86	POWER	Armv7	Armv8	Reordering				Elimination				ILE	SLI	RM	S	TP	CSE	RP	TI	VR	eDRF	COH	no-UB	In-Order	no-OOTA
						SL	SS	LL	LS	SL	SS	LL	LS														
Sequential Consistency	SC [8, 32, 33, 41, 42]	−	−	−	−	−	−	−	−	+	+	+	+	+	+	+	+	+	−	+	+		+	+	+	+	+
	DRFx [43]	−	−	−	−	−	−	−	−	+	+	+	+	+	−	+	+	+	−	+	+		+	+	−	+	+
Total/Partial Store Order	BMM [44]	+	−	−	−	+	−	−	−	+	+	+	−	+	+	−		+	−		−		+	+	+	+	+
	RMMOA [45]	+	−	−	−	+	+	−	−														+		+	+	+
Program Order Preserving	RC11 [21, 46–48]	+	−	−	−	+	+	+	−	+	+	+	−		−	+	+	+	−	+	+		+	+	−	+	+
	OCMM [22]	+	−	−	−	+	+	+	−	+	+	+	−					+	+				+	−	+	+	+
	JAM [4]	+	−	−	−												+		−				+	+	+	+	+
Syntax. Dep. Preserving	LKMM [49]	+	+	+	+	+	+	+	+									−	−					+	+	−	+
	OHMM [50]					+	+	+	+	+	+	+	+	+	+	+		−					+		+	−	+
Semantic Dep. Preserving	JMM [3, 36, 51]	+	−	+	+	+	+	+	+	+	+	−	−	+	−	−		+	−		−		+	−	+	−	+
	PRM [17, 18]	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	−	+	+	+	+	−	+
	WMO [6, 19]	+	+	+	+	+	+	+	+	+	+	+	−		+	−	+		−		−		+	+	−	−	+
	CSRA [25]	+	+	−	−	+	+	+	+	+	+	+	−	+	−	+	−		−		−	+		+	−	−	+
	WJES [24]		−	−	−			−				−	−										+	−	+	−	+
	MRD [20]	+	+	+	+																		+	+	+	−	+
	GOS [52]																						+		+	−	+
Out of Thin-Air	C11 [5, 29, 30, 38, 53–57]	+	+	+	+	+	+	+	+	+	+	+	−		−	+	+	+	−		+		−	+	−	−	−
	JSMM [7]	+	+	+	+																		−	−	+	−	−
	RMC [58]	+	+	+	+																		−	+	+	−	−
	RAO [59]																							−	+	−	−
	TSC [40]																						−		+	−	−

Table 2.: Memory models and their properties

reasoning guarantees disables some program transformations and requires a more heavyweight compilation mapping to hardware.

We stat with a discussion of the class of sequentially consistent models §6.1. Then we proceed to the class of totally and partially store ordered models §6.2. After that we switch to the class of very weak models permitting thin-air values §6.3. Then we describe various solutions tackling the problem of thin-air values, namely program order preserving models §6.4, syntactic dependency preserving models §6.5, and semantic dependency preserving models §6.6.

In §6.7 we also discuss the particular properties of models, namely the coherence and the catch-fire semantics, which are orthogonal to the main partitioning into classes, but nonetheless they affect the soundness of certain program transformations.

6.1 Sequential Consistency

Sequential Consistency (SC) is one of the most intuitive models of concurrency. Under this model, one can represent a state of the memory as a simple mapping from memory locations to their values. Then each outcome can be obtained by sequential execution of some interleaving of threads' instructions.

Class	Model	Features								
		NA	RLX	RA	SC	F-RA	F-SC	RMW	LK	MIX
Sequential Consistency	EtE-SC [32,41]	−	−	−	+	−	−	−	−	−
	VbD [33,42]	+	−	−	+	−	−	+	+	−
	SC-Hs [8]	+	−	−	+	−	−	+	−	−
	DRFx [43]	+	−	−	+	−	−	−	−	−
Total/Partial Store Order	BMM [44]	+	−	−	+	−	−	−	+	−
	RMMOA [45]	+	−	−	−	−	−	−	+	−
Program Order Preserving	RC11 [21]	+	+	+	+	+	+	+	−	−
	ORC11 [48]	+	+	+	−	+	−	+	−	−
	RAR [47]	−	+	+	−	−	−	+	−	−
	CRC [46]	+	−	+	−	−	+	+	−	−
	OCMM [22]	+	−	−	+	−	−	+	−	−
	JAM [4]	+	+	+	+	+	+	+	−	−
Syntax. Dep. Preserving	LKMM [49]	−	+	+	−	+	+	+	−	−
	OHMM [50]	+	−	−	+	−	−	−	+	−
Semantic Dep. Preserving	JMM [3]	+	−	−	+	−	−	+	+	−
	PRM [17]	+	+	+	−	+	+	+	+	−
	WMO [19]	+	+	+	+	+	+	+	−	−
	CSRA [25]	+	+	+	−	−	−	+	+	−
	WJES [24]	−	+	−	−	−	−	+	+	−
	MRD [20]	−	+	−	−	−	−	−	+	−
	GOS [52]	+	−	−	−	−	−	−	+	−
Out of Thin-Air	C11 [5]	+	+	+	+	+	+	+	+	+
	JSMM [7]	+	−	−	+	−	−	+	+	+
	RMC [58]	−	+	+	+	+	+	+	−	−
	RAO [59]	+	−	−	+	−	−	−	−	−
	TSC [40]	+	−	−	+	−	−	−	+	−

Table 3.: Features supported by memory models

SC renders many common transformations unsound, including all kinds of instruction reorderings and common subexpression elimination [32, 36]. The fact that instruction reorderings are forbidden makes the model expensive to implement on modern hardware since even the relatively strong hardware model of x86 performs store/load reordering. Therefore, in order to preserve the sequential consistency during compilation, a compiler need to emit heavyweight memory fences between store and load instructions, which makes the compilation mappings far from being optimal.

In terms of reasoning guarantees, however, SC is quite a pleasant model. It gives the external DRF and the coherence properties for free, because it assigns to programs only sequentially consistent outcomes by definition.

The conceptual simplicity of SC have inspired many researchers to adopt it and to try to mitigate the induced performance penalty. The recurring idea was to somehow separate thread-local and shared mutable memory. Accesses to thread-local memory can be compiled without memory fences and are subject to a wider range of local transformations. To safely distinguish between two types of memory researches proposed to utilize a type-system [8], static [41] or dynamic [42] analysis, a hardware support [41,43] or some combination of the above.

Despite these efforts SC still induces considerable slowdown, especially on weak hardware. For instance, on Armv8 machines the slowdown can be up to 70% [42] (see appendix A.1 for details). Moreover, while these optimization typically reduce penalties on thread-local accesses (a common case), they are likely to have a lesser impact on specific applications which heavily utilize concurrency, for example, lock-free data structures. Finally, modern compilers usually require a significant amount of engineering work and rewrite in order to preserve SC [32,42].

6.2 Total or Partial Store Order

The next class of PL memory models we consider was inspired by the *total store order* (TSO) and the *partial store order* (PSO) models. TSO and PSO are memory models of x86 [9] and SPARC [60] hardware correspondingly. In these models threads are equipped with *store buffers*. All store operations go to these buffers before they propagate into the main memory.

Models of this class can be compiled down to the x86 hardware without any performance penalty, since x86 implements TSO model itself. However, on weaker hardware like POWER a compiler need to emit essentially as many fences as to enforce SC [61].

This class permits more program transformations than SC. Store buffers enable store/load reorderings in case of TSO, and additionally store/store reorderings in case of PSO. The TSO and PSO models are weaker than the SC, but they are still relatively strong — the external DRF and the coherence properties hold.

Therefore these models do not have any significant benefits in terms of the reasoning guarantees compared to SC, but induce a similar performance penalty on architectures weaker than x86. Hence a choice of TSO and PSO as a programming language level memory model is reasonable only if the language targets the x86 hardware solely.

6.3 Out of Thin-Air Values

We next move on to the other end of a memory models' spectrum. We consider the class uniting the weakest models from our list. These models enable efficient compilation mappings and almost all reasonable program transformations, but at a cost of introducing thin-air values (§4.3.4).

Let us revisit the load buffering program:

$$\underset{\text{(LB)}}{\begin{array}{l||l} r_1 := x & r_2 := y \\ y := 1 & x := r_2 \end{array}} \rightsquigarrow \underset{\text{(LBtr)}}{\begin{array}{l||l} y := 1 & r_2 := y \\ r_1 := x & x := r_2 \end{array}}$$

Program on the right LBtr can be obtained from the program on the left LB via the load/store reordering. The outcome $[r_1 = 1, r_2 = 1]$ is valid for LBtr. Therefore under a memory model where the load/store reordering is a sound transformation, this outcome should also be valid for LB. As we demonstrated in §4.3.4 in order to allow this outcome, memory models needs to utilize speculative execution.

We also demonstrated that unrestricted speculations can lead to so called thin-air values which break fundamental reasoning principles [16, 38]. In the presence of thin-air values type safety and security guarantees can be violated, and compositional reasoning is impossible. Moreover, they are also incompatible with the external DRF property. To see this consider yet another variation of the load buffering program:

$$\begin{array}{l||l} r_1 := x & r_2 := y \\ \text{if } (r_1)\ \{ & \text{if } (r_2)\ \{ \\ \quad y := 1 & \quad x := 1 \\ \} & \} \end{array} \qquad \text{(LB+ctrl)}$$

For a memory model admitting thin-air values (as *e.g.,* C11 [5]), the outcome $[r_1 = 1, r_2 = 1]$ is valid (a justification is the same as for the LB+data example from §4.3.4). Not only this outcome is completely unintuitive, but it also contradicts the external DRF guarantee. Indeed, under SC the program above has a single valid execution with the outcome $[r_1 = 0, r_2 = 0]$ containing no data-races, thus under a DRF compliant model it should also have this sole outcome.

The counter-intuitive behavior of OOTA models, together with the fact that they break important reasoning principles, has lead over the time to the consensus in the research community that these models are not suited well for the role of programming languages memory models [16, 38]. A lot of effort has been put to forbid problematic thin-air outcomes, while still keep the compilation scheme as efficient as possible and enable as many transformations as possible. In the rest of this section we describe various proposals tackling the thin-air problem.

6.4 Program Order Preserving

The most straightforward way to forbid thin-air values was proposed by Boehm and Demsky [16] The idea is to simply prohibit any kind of speculative execution, which can be achieved by forbidding load/store reorderings altogether. This fix not only restores the external DRF and other reasoning guarantees [21], but also leads to a much simpler model. The abstract machine implementing the memory model does not need to resort to speculative execution and can perform threads' instructions in-order. A memory storage can be implemented as a monotonically growing history of messages, with each thread having its own view on a frontier of this history [22, 47].

Lahav *et al.* [21] formalized this approach to the thin-air problem and studied it extensively. The authors were shown that many program transformations are still sound in this setting, with the obvious exception of the load/store reordering itself (see table 1 for details).

The compilation mapping to x86 remain efficient, since this architecture already guarantee to preserve order between loads and subsequent stores. However, weaker architectures (Arm, POWER) do not guarantee that, and thus additional measures are required. Boehm and Demsky [16] proposed to compile every relaxed load as a plain load followed by a spurious conditional jump instruction, which introduces a dependency between the load and subsequent stores. Arm and POWER hardware preserves this dependency, and thus it also retains the load/store ordering. Ou and Demsky [23] studied a performance penalty required to preserve load/store ordering between **relaxed atomics only** on the Armv8 hardware and reported a negligible overhead of 0% on average and 6.3% in maximum on a set of benchmarks implementing various concurrent data-structures, *e.g.,* locks, stacks, queues, deques, maps, *etc.* (see A.3 for details). Note that the overhead is expected to be greater if the compilation scheme is required to preserve the ordering between non-atomic accesses as well.

6.5 Syntactic Dependencies Preserving

The alternative conceptually simple solution to thin-air values problem is to preserve *syntactic dependencies* [16, 49]. Under this approach reordering of independent load/store pairs is allowed. However, reordering is forbidden if a store depends on the value read by a load either because this value was used to compute the value written by the store (*data dependency*), or it was used to compute the memory address of the store (*address dependency*), or else a control-flow path lead to the store was dependent on this value (*control dependency*). For example, giving the program LB+data the store $y := r_1$ depends on the load $x := r_1$ since it writes the value read by the load.

Note that these kind of dependencies are computed following the syntax of a program (hence the name) as opposed to *semantic dependencies.* For example, giving the modified version of the LB+data program below, the store to y in the left thread is still considered to have syntactic dependency on the previous load.

$$\begin{array}{l} r_1 := x \\ y := 1 + 0 * r_1 \end{array} \left\| \begin{array}{l} r_2 := y \\ x := r_2 \end{array} \right. \qquad \text{(LB+fakedata)}$$

Here the syntactic dependency can be eliminated by the *constant folding* transformation — the expression $1 + 0 * r_1$ can be reduced to value 1. Under a syntactic dependency preserving memory model a compiler, however, is prohibited to perform this optimization. Indeed, once a dependency is removed, nothing prevents to reorder a store before a preceding load. Even if a compiler itself does not perform this reordering, after the compilation hardware can do this during the execution.

This subtlety reveals the main drawback of syntactic dependency tracking models — trace preserving transformations (*e.g.,* constant folding) are unsound in these models. Constant folding is one of the classic optimizations that any compiler might want to apply, and the fact that it is unsound makes an adoption of this class of models problematic. Note that hardware memory models apply a similar approach and usually have a notion of syntactic dependencies between memory operations [11, 12, 14]. Yet in this setting the unsoundness of trace preserving transformations is not a problem, since hardware does not perform such complex optimizations.

Ou and Demsky [23] examined the performance penalty of a syntactic dependency tracking compiler. They adjusted compiler optimization passes to preserve dependencies between **non-atomic and relaxed** accesses. They evaluated the dependency preserving version of the LLVM compiler infrastructure on Armv8 hardware using SPEC CPU2006 benchmarks and reported a moderate slowdown of 3.1% on average and 17.6% in maximum. (see A.4 for details).

6.6 Semantic Dependencies Preserving

The last approach to tackle thin-air problem is to construct a notion of *semantic dependencies*, which would precisely characterize what load/store pairs are independent and rule out fake dependencies like the one in LB+fakedata. A practical payoff of this approach is that it does not require significant modifications to existing compilers or hardware, and thus should not impose performance penalties. The ultimate goal is to enable the optimal compilation mappings, preserve most of the existing compiler optimizations, and at the same time maintain the important reasoning guarantees like external DRF.

It turns out that this task is quite challenging and to this date there is no strong consensus on how to achieve it. In order to give a satisfactory definition of semantic dependencies researchers had to resort to conceptually complex memory models [17, 19, 20, 24, 25, 52]. The main challenge in this line of work was to formally prove that these complex models indeed satisfy all the desired properties.

Currently the most complete approach of this class is the Promising semantics [17,18]. This model was proven to enable the optimal compilation schemes [62], and permit most local and global program transformations (with a notable exception of the thread inlining), while still preserving the external DRF guarantee.

6.7 Secondary Classes

We also identified an alternative division of memory models into groups, which correspond to particular properties of a memory model, namely the coherence and the catch-fire semantics which treats racy programs as erroneous. We demonstrate how the presence of these properties affects the compilation mappings and the soundness of certain program transformations.

6.7.1 Coherent Models

The coherence property (*i.e.,* SC-per-location, §4.3.2) has a subtle effect on the common subexpression elimination optimization (CSE), which was was first observed in the context of an early version of the Java memory model [63]. To see the problem, consider the program below (on the left) and the transformed version of this program after application of CSE (on the right). Note that the optimization has replaced the second access to variable x by a read from a register.

$$\begin{array}{l||l} r_1 := x & \\ r_2 := y & y := 1 \\ r_3 := x & \end{array} \quad \rightsquigarrow \quad \begin{array}{l||l} r_1 := x & \\ r_2 := y & y := 1 \\ r_3 := r_1 & \end{array}$$

Now assume that variables x and y point to the same memory location. Under this assumption the outcome $[r_1 = 0, r_2 = 1, r_3 = 0]$ is forbidden for a memory model respecting coherence. Indeed, the coherence guarantees sequential consistency per location, which means that for programs consisting of accesses to a single memory location (as the one above in the presence of aliasing) only the sequentially consistent outcomes are allowed. The outcome $[r_1 = 0, r_2 = 1, r_3 = 0]$ cannot be obtained as the interleaving of instructions, and thus it should be forbidden. However, this outcome is allowed for the optimized version of the program.

Note that the compiler still can apply CSE to the program above, but only if it is able to prove that variables x and y point to disjoint memory locations, which can be achieved by alias analysis [64]. In fact, in this case CSE can be seen as a combination of instruction reordering and elimination transformations.

Therefore, the coherence property in general is not compatible with the common subexpression elimination. As for compilation schemes, coherence does not require any changes here and thus does not impose any performance penalty. It is because hardware models already guarantee coherence [9, 11,12,21].

6.7.2 Catch-Fire Models

A catch-fire semantics that treats racy non-atomic accesses as undefined behavior also affects soundness of a program transformations. As we already briefly discussed, it enables the optimal compilation mapping for non-atomic accesses and makes sequentially valid transformations applicable to them, but its impact is not limited to this observation. A catch-fire semantics also has an interesting interplay with the speculative load introduction.

Consider the following example:

$$\begin{array}{l||l} r := x & \\ \mathsf{if}\ (r)\ \{ & y := 1 \\ \quad s := y & \\ \} & \end{array} \quad \rightsquigarrow \quad \begin{array}{l||l} r_1 := x & \\ t := y & \\ \mathsf{if}\ (r)\ \{ & y := 1 \\ \quad s := t & \\ \} & \end{array}$$

As we mentioned in §4.2, the speculative load introduction can be used in combination with the load/load elimination to move a load instruction out of one branch of a conditional. In more detail, giving the example above, the speculative load introduction can be applied to add the load $t := y$ before the if statement, and then the load/load elimination can be used to replace the second load with an assignment.

The subtle point here is that while the left program is race free even under SC, the right program is racy under SC semantics, because of the race between load and store to y. This fact implies that if all accesses in the programs above are non-atomic, then a catch-fire semantics should treat the right program as having undefined behavior. In other words, the right program allows any outcome, while the left program allows only the outcome $[r = 0]$. Soundness of program transformation requires a set of outcomes of a transformed program to be a subset of outcomes of the original program. This condition is clearly violated in our example.

Put simply, speculative load introduction in general is unsound in catch-fire memory models, because it can bring data-races into otherwise race-free programs. Since catch-fire semantics is sensitive to the presence of data-races it is incompatible with this transformation.

Note that this problem cannot be mitigated by forbidding only non-atomic load introduction and allowing atomic load introduction. Indeed, an introduced atomic load access still can race with some non-atomic load or store located elsewhere in a program.

7 Guide for Choosing a Model

In this section we provide a summary of our findings and present a short guide for researchers and system-level developers on how to choose a memory model based on design principles of a programming language.

A language that seeks to provide a clear semantics and high-level programming abstractions at the cost of some performance losses most definitely should adhere to simple memory models like sequential consistency.

Programming languages focusing on efficiency of compiled code, for example, C/C++, have to resort to weakest models admitting the optimal compilation mapping and wide range of program transformations. For these languages it would be natural to pick semantic dependency preserving models. However, these models are the most complicated ones which makes reasoning about programs' correctness under them challenging [65].

In the middle between two extremes are program order preserving models. Those are a reasonable choice for programming languages which may afford a moderate performance overhead in exchange of a simpler and more predictable program behavior [23].

A nice property of program order preserving models, compared to a syntax or semantic dependency preserving models, is that they do not utilize any form of speculative execution. This further simplifies the reasoning about the correctness of concurrent programs and better reflects the expectations of programmers. The price for this simplicity is that these models require sub-optimal compilation mappings to the Arm and POWER hardware. In contrast, syntactic dependency preserving models can be efficiently implemented on Arm and POWER hardware, but these models do not support trace preserving transformations, including constant folding, and utilize speculative execution, leading to a more complicated semantics.

For languages adopting stronger models which require non-optimal compilation mappings and forbid certain program transformations there are some general optimization techniques and design decisions which may partly mitigate the induced performance penalties.

A type system can serve as a great help in this task. Languages like Haskell, OCaml, Rust that statically distinguish and isolate memory regions which can be accessed and modified concurrently have a great advantage. These languages can identify precisely immutable and thread local variables and compile accesses to them without insertion of fences. Moreover, memory accesses to local variables are subject to a wide range of program transformations proven to be sound for single threaded programs.

Languages like Java which cannot utilize the type system to fully prevent racy accesses to non-atomic variables because of the backward compatibility, still can approximate a set of thread local variables using a conservative static escape analysis [66] or various dynamic techniques [42], and then apply similar optimizations to them.

Functional programming languages encourage programmers to use immutable data whenever possible. This style of programming minimizes the use shared memory and mitigates the performance impact of a strong memory model [8].

Finally, if the language tolerates undefined behavior, as C/C++ does, an alternative to a complex semantic dependency preserving model could be a program order preserving model which treats data races on non-atomic accesses as undefined behavior [16, 23]. In this case a compiler can use optimal compilation mappings and apply a wide range of transformations to non-atomics and at the same time have a simpler semantics for atomics.

Choosing a Memory Model for Kotlin As an example, consider Kotlin,[12] a general-purpose programming language, which does not have a standardized memory model yet. Currently, Kotlin can be compiled to Java bytecode, to JavaScript code, or to native code via LLVM (for Linux, Windows, macOS, iOS, and other platforms).

The language is not oriented to system-level programming, that is, it does not have to provide zero cost abstraction over target hardware for shared memory accesses. Therefore a memory model which either preserves program order or syntax dependencies is suitable for Kotlin. Both approaches have moderate performance penalties. However, program order preservation works better for languages tolerating undefined behavior for data races involving non-atomic accesses (see [23]) since it allows to compile non-atomics as plain accesses to architectures like Arm and POWER which do not guarantee to preserve the program order. Even though having undefined behavior for Kotlin in general might be undesirable, it is practically unavoidable because data races on non-atomics have undefined behavior under LLVM [6].

Among the two classes of models the most well-studied and feature-rich model is RC11 [21], which adds program order preservation to the C11 memory model [5]. RC11 supports a superset of access types presented in JMM [3] and its extension JAM [4], and it is very close to the memory models of JavaScript [7] and LLVM [6] since both of them were inspired by C11.

That makes RC11 a good starting point for development of a memory model for Kotlin.

8 Conclusion and Future Work

In this work we surveyed memory models proposed for various programming language. We compared them based on the common set of criteria developed in the literature and identified six main classes of memory models. We also presented a short guide on how to choose a suitable memory model based on the design of a programming language. We hope our work will be helpful for programming language researches and implementors, and will serve as a gentle introduction to the complex topic of weak memory models. Based on our analysis, we can suggest several possible directions for future work in the field.

The problems of the optimality of compilation schemes and the soundness of local program transformations are relatively well-studied. More recent memory models, RC11 [21], OCaml MM [22], Promising [17, 18], and Weakestmo [19], support a wide range of local transformations and have clear trade-offs in terms of compilation mappings. An exception is local transformations involving loops and recursion, their soundness was not studied formally. Global transformations also received a little attention so far, with few notable exceptions [18, 25]. The exact impact of these transformations on the design of memory models is yet to be discovered.

Whole program data-race freedom guarantees were also studied extensively [3, 17, 21, 38]. In contrast, the local data-race freedom [22] is a relatively new concept. We expect it as well as local reasoning guarantees in general [46, 67, 68] to receive more attention in the near future.

Mixed size accesses [30], which are already used in the JavaScript memory model [7] and real-world applications, for example, in the Linux kernel codebase [30], are understudied even for hardware. Proper understanding of them is an important direction for the community.

Semantic dependency preserving models are still an active area of research [17–20, 67, 68]. We expect those to be a subject to further refinement. An interesting line of work here would be the development of new reasoning principles beyond data-race freedom, which could improve the meta-theory of these models and simplify reasoning about the correctness of programs.

Finally, comprehensive quantitative studies of the performance penalties induced by memory models are quite valuable. Although there is some work in this direction, [8, 22, 23, 33, 41, 42], the full picture is still unclear.

[12] https://kotlinlang.org/

Acknowledgments

We thank Ori Lahav for his comments on the draft of this paper.

The reported study was funded by RFBR, project number 20-31-90088.

References

1. E. W. Dijkstra, "Cooperating sequential processes," in *The origin of concurrent programming*, pp. 65–138, Springer, 1968.

2. L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Trans. Computers*, vol. 28, no. 9, pp. 690–691, 1979.

3. J. Manson, W. Pugh, and S. V. Adve, "The Java memory model," in *POPL 2005*, pp. 378–391, ACM, 2005.

4. J. Bender and J. Palsberg, "A formalization of Java's concurrent access modes," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–28, 2019.

5. M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber, "Mathematizing C++ concurrency," in *POPL 2011*, pp. 55–66, ACM, 2011.

6. S. Chakraborty and V. Vafeiadis, "Formalizing the concurrency semantics of an LLVM fragment," in *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 100–110, IEEE, 2017.

7. C. Watt, C. Pulte, A. Podkopaev, G. Barbier, S. Dolan, S. Flur, J. Pichon-Pharabod, and S.-y. Guo, "Repairing and mechanising the JavaScript relaxed memory model," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 346–361, 2020.

8. M. Vollmer, R. G. Scott, M. Musuvathi, and R. R. Newton, "SC-Haskell: Sequential consistency in languages that minimize mutable shared heap," *ACM SIGPLAN Notices*, vol. 52, no. 8, pp. 283–298, 2017.

9. P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen, "x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors," *Commun. ACM*, vol. 53, no. 7, pp. 89–97, 2010.

10. J. Alglave, A. Fox, S. Ishtiaq, M. O. Myreen, S. Sarkar, P. Sewell, and F. Z. Nardelli, "The semantics of Power and ARM multiprocessor machine code," in *Proceedings of the 4th workshop on Declarative aspects of multicore programming*, pp. 13–24, 2009.

11. S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams, "Understanding POWER multiprocessors," in *PLDI 2011*, pp. 175–186, ACM, 2011.

12. J. Alglave, L. Maranget, and M. Tautschnig, "Herding cats: Modelling, simulation, testing, and data mining for weak memory," *ACM Trans. Program. Lang. Syst.*, vol. 36, no. 2, pp. 7:1–7:74, 2014.

13. N. Chong and S. Ishtiaq, "Reasoning about the ARM weakly consistent memory model," in *Proceedings of the 2008 ACM SIGPLAN workshop on Memory systems performance and correctness: held in conjunction with the Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'08)*, pp. 16–19, 2008.

14. C. Pulte, S. Flur, W. Deacon, J. French, S. Sarkar, and P. Sewell, "Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8," *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, pp. 1–29, 2018.

15. S. Flur, K. E. Gray, C. Pulte, S. Sarkar, A. Sezgin, L. Maranget, W. Deacon, and P. Sewell, "Modelling the ARMv8 architecture, operationally: Concurrency and ISA," in *POPL 2016*, pp. 608–621, ACM, 2016.

16. H.-J. Boehm and B. Demsky, "Outlawing ghosts: Avoiding out-of-thin-air results," in *MSPC 2014*, pp. 7:1–7:6, ACM, 2014.

17. J. Kang, C.-K. Hur, O. Lahav, V. Vafeiadis, and D. Dreyer, "A promising semantics for relaxed-memory concurrency," in *POPL 2017*, ACM, 2017.

18. S.-H. Lee, M. Cho, A. Podkopaev, S. Chakraborty, C.-K. Hur, O. Lahav, and V. Vafeiadis, "Promising 2.0: global optimizations in relaxed memory concurrency," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 362–376, 2020.

19. S. Chakraborty and V. Vafeiadis, "Grounding thin-air reads with event structures," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–28, 2019.

20. M. Paviotti, S. Cooksey, A. Paradis, D. Wright, S. Owens, and M. Batty, "Modular relaxed dependencies in weak memory concurrency," in *European Symposium on Programming*, pp. 599–625, Springer, Cham, 2020.

21. O. Lahav, V. Vafeiadis, J. Kang, C.-K. Hur, and D. Dreyer, "Repairing sequential consistency in C/C++11," in *PLDI 2017*, ACM, 2017.

22. S. Dolan, K. Sivaramakrishnan, and A. Madhavapeddy, "Bounding data races in space and time," *ACM SIGPLAN Notices*, vol. 53, no. 4, pp. 242–255, 2018.

23. P. Ou and B. Demsky, "Towards understanding the costs of avoiding out-of-thin-air results," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–29, 2018.

24. A. Jeffrey and J. Riely, "On thin air reads: Towards an event structures model of relaxed memory," in *LICS 2016*, IEEE, 2016.

25. J. Pichon-Pharabod and P. Sewell, "A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions," in *POPL 2016*, pp. 622–633, ACM, 2016.

26. S. Marlow *et al.*, "Haskell 2010 language report," *Available on: https://www. haskell. org/onlinereport/haskell2010*, 2010.

27. S. Klabnik and C. Nichols, *The Rust Programming Language (Covers Rust 2018).* No Starch Press, 2019.

28. H.-J. Boehm and S. V. Adve, "Foundations of the C++ concurrency memory model," *ACM SIGPLAN Notices*, vol. 43, no. 6, pp. 68–78, 2008.

29. O. Lahav, N. Giannarakis, and V. Vafeiadis, "Taming release-acquire consistency," *ACM SIGPLAN Notices*, vol. 51, no. 1, pp. 649–662, 2016.

30. S. Flur, S. Sarkar, C. Pulte, K. Nienhuis, L. Maranget, K. E. Gray, A. Sezgin, M. Batty, and P. Sewell, "Mixed-size concurrency: ARM, Power, C/C++ 11, and SC," *ACM SIGPLAN Notices*, vol. 52, no. 1, pp. 429–442, 2017.

31. "C/C++11 mappings to processors," 2011. Available at `https://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html` [Online; accessed 26-April-2021].

32. D. Marino, A. Singh, T. Millstein, M. Musuvathi, and S. Narayanasamy, "A case for an SC-preserving compiler," *ACM SIGPLAN Notices*, vol. 46, no. 6, pp. 199–210, 2011.

33. L. Liu, T. Millstein, and M. Musuvathi, "A volatile-by-default JVM for server applications," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–25, 2017.

34. S. Muchnick, *Advanced compiler design and implementation.* Morgan kaufmann, 1997.

35. O. Lahav, E. Namakonov, J. Oberhauser, A. Podkopaev, and V. Vafeiadis, "Making weak memory models fair," *arXiv preprint arXiv:2012.01067*, 2020.

36. J. Ševčík and D. Aspinall, "On validity of program transformations in the Java memory model," in *European Conference on Object-Oriented Programming*, pp. 27–51, Springer, 2008.

37. M. N. Wegman and F. K. Zadeck, "Constant propagation with conditional branches," *ACM*

Transactions on Programming Languages and Systems (TOPLAS), vol. 13, no. 2, pp. 181–210, 1991.

38. M. Batty, K. Memarian, K. Nienhuis, J. Pichon-Pharabod, and P. Sewell, "The problem of programming language concurrency semantics," in *ESOP*, vol. 9032 of *LNCS*, pp. 283–307, Springer, 2015.

39. L. Maranget, S. Sarkar, and P. Sewell, "A tutorial introduction to the ARM and POWER relaxed memory models," 2012. Available at `https://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf` [Online; accessed 30-April-2021].

40. G. Boudol and G. Petri, "A theory of speculative computation," in *European Symposium on Programming*, pp. 165–184, Springer, 2010.

41. A. Singh, S. Narayanasamy, D. Marino, T. Millstein, and M. Musuvathi, "End-to-end sequential consistency," in *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pp. 524–535, IEEE, 2012.

42. L. Liu, T. Millstein, and M. Musuvathi, "Accelerating sequential consistency for Java with speculative compilation," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 16–30, 2019.

43. D. Marino, A. Singh, T. Millstein, M. Musuvathi, and S. Narayanasamy, "DRFx: A simple and efficient memory model for concurrent programming languages," in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 351–362, 2010.

44. D. Demange, V. Laporte, L. Zhao, S. Jagannathan, D. Pichardie, and J. Vitek, "Plan B: A buffered memory model for Java," in *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 329–342, 2013.

45. G. Boudol and G. Petri, "Relaxed memory models: an operational approach," *ACM SIGPLAN Notices*, vol. 44, no. 1, pp. 392–403, 2009.

46. M. Dodds, M. Batty, and A. Gotsman, "Compositional verification of compiler optimisations on relaxed memory," in *European Symposium on Programming*, pp. 1027–1055, Springer, 2018.

47. S. Doherty, B. Dongol, H. Wehrheim, and J. Derrick, "Verifying C11 programs operationally," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, pp. 355–365, 2019.

48. H.-H. Dang, J.-H. Jourdan, J.-O. Kaiser, and D. Dreyer, "RustBelt meets relaxed memory," *Proceedings of the ACM on Programming Languages*, vol. 4, no. POPL, pp. 1–29, 2019.

49. J. Alglave, L. Maranget, P. E. McKenney, A. Parri, and A. Stern, "Frightening small children and disconcerting grown-ups: Concurrency in the Linux kernel," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 405–418, 2018.

50. Y. Zhang and X. Feng, "An operational happens-before memory model," *Frontiers of Computer Science*, vol. 10, no. 1, pp. 54–81, 2016.

51. M. Huisman and G. Petri, "The Java memory model: a formal explanation," *VAMP*, vol. 7, pp. 81–96, 2007.

52. R. Jagadeesan, C. Pitcher, and J. Riely, "Generative operational semantics for relaxed memory models," in *European Symposium on Programming*, pp. 307–326, Springer, 2010.

53. S. Sarkar, K. Memarian, S. Owens, M. Batty, P. Sewell, L. Maranget, J. Alglave, and D. Williams, "Synchronising C/C++ and POWER," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 311–322, 2012.

54. M. Batty, K. Memarian, S. Owens, S. Sarkar, and P. Sewell, "Clarifying and compiling

C/C++ concurrency: from C++ 11 to POWER," *ACM SIGPLAN Notices*, vol. 47, no. 1, pp. 509–520, 2012.

55. V. Vafeiadis, T. Balabonski, S. Chakraborty, R. Morisset, and F. Z. Nardelli, "Common Compiler Optimisations are Invalid in the C11 Memory Model and what we can do about it," in *POPL 2015*, pp. 209–220, ACM, 2015.

56. M. Batty, A. F. Donaldson, and J. Wickerson, "Overhauling SC atomics in C11 and OpenCL," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 634–648, 2016.

57. K. Nienhuis, K. Memarian, and P. Sewell, "An operational semantics for C/C++ 11 concurrency," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 111–128, 2016.

58. K. Crary and M. J. Sullivan, "A calculus for relaxed memory," in *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 623–636, 2015.

59. V. A. Saraswat, R. Jagadeesan, M. Michael, and C. von Praun, "A theory of memory models," in *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 161–172, 2007.

60. S. I. Inc and D. L. Weaver, *The SPARC architecture manual.* Prentice-Hall, 1994.

61. D. Lustig, C. Trippel, M. Pellauer, and M. Martonosi, "ArMOR: Defending against memory consistency model mismatches in heterogeneous architectures," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pp. 388–400, 2015.

62. A. Podkopaev, O. Lahav, and V. Vafeiadis, "Bridging the gap between programming languages and hardware weak memory models," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–31, 2019.

63. W. Pugh, "Fixing the Java memory model," in *Proceedings of the ACM 1999 conference on Java Grande*, pp. 89–98, 1999.

64. A. Diwan, K. S. McKinley, and J. E. B. Moss, "Type-based alias analysis," *ACM Sigplan Notices*, vol. 33, no. 5, pp. 106–117, 1998.

65. K. Svendsen, J. Pichon-Pharabod, M. Doko, O. Lahav, and V. Vafeiadis, "A separation logic for a promising semantics," in *Programming Languages and Systems* (A. Ahmed, ed.), (Cham), pp. 357–384, Springer International Publishing, 2018.

66. J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff, "Escape analysis for Java," *Acm Sigplan Notices*, vol. 34, no. 10, pp. 1–19, 1999.

67. R. Jagadeesan, A. Jeffrey, and J. Riely, "Pomsets with preconditions: a simple model of relaxed memory," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–30, 2020.

68. M. Cho, S.-H. Lee, C.-K. Hur, and O. Lahav, "Modular data-race-freedom guarantees in the Promising semantics," in *Proceedings of the 42st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2021.

69. D. Marino, A. Singh, T. Millstein, M. Musuvathi, and S. Narayanasamy, "Drf x: An understandable, high performance, and flexible memory model for concurrent languages," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 38, no. 4, pp. 1–40, 2016.

70. J. Ševčík, V. Vafeiadis, F. Zappa Nardelli, S. Jagannathan, and P. Sewell, "CompCertTSO: A verified compiler for relaxed-memory concurrency," *Journal of the ACM (JACM)*, vol. 60, no. 3, pp. 1–50, 2013.

71. F. Pizlo, L. Ziarek, E. Blanton, P. Maj, and J. Vitek, "High-level programming of embedded hard real-time devices," in *Proceedings of the 5th European conference on Computer systems*, pp. 69–82, 2010.

72. M. Khiszinsky, "CDS C++ library," 2017. Available at `https://github.com/khizmax/libcds` [Online; accessed 26-April-2021].

73. "Folly: Facebook open-source library," 2018. Available at `https://github.com/facebook/folly` [Online; accessed 26-April-2021].

74. J. Preshing, "Junction — a library of concurrent data structures in C++," 2018. Available at `https://github.com/preshing/junction` [Online; accessed 26-April-2021].

75. "VarHandle API Docs.," 2017. Available at `https://docs.oracle.com/javase/9/docs/api/java/lang/invoke/VarHandle.html` [Online; accessed 22-Feb-2021].

76. D. Lea, "JEP 193: Variable handles," 2017. Available at `http://openjdk.java.net/jeps/193`[Online; accessed 22-Feb-2021].

77. D. Lea, "Using JDK 9 memory order modes," 2018. Available at `http://gee.cs.oswego.edu/dl/html/j9mm.html`[Online; accessed 22-Feb-2021].

78. D. Howells, P. E. McKenney, W. Deacon, and P. Zijlstra, "Linux kernel memory barriers," 2017. Available at `https://www.kernel.org/doc/Documentation/memory-barriers.txt` [Online; accessed 22-Feb-2021].

79. P. E. McKenney, "Proper care and feeding of return values from rcu_dereference()," 2017. Available at `https://www.kernel.org/doc/Documentation/RCU/rcu_dereference.txt` [Online; accessed 22-Feb-2021].

80. P. E. McKenney and J. Walpole, "What is RCU, fundamentally?," *Linux Weekly News (LWN.net)*, 2007.

81. G. Winskel, "Event structures," in *Advanced Course on Petri Nets*, pp. 325–392, Springer, 1986.

82. A. Podkopaev, O. Lahav, and V. Vafeiadis, "Promising compilation to ARMv8 POP," in *ECOOP 2017*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

83. E. Moiseenko, A. Podkopaev, O. Lahav, O. Melkonian, and V. Vafeiadis, "Reconciling event structures with modern multiprocessors," in *34th European Conference on Object-Oriented Programming*, 2020.

84. J. Y. A. Pichon-Pharabod, *A no-thin-air memory model for programming languages.* PhD thesis, University of Cambridge, 2018.

A Catalog of Memory Models

In this section we dive inside the classes of memory models given in §6, and present a more detailed view on each memory model we consider in our study. It contains an overview of each PL memory model, a recap of its distinguished properties, and references to the research papers studying this model.

A.1 Sequential Consistency

We start with a description of several attempts to adopt the sequential consistency as a memory model for existing languages and runtimes. Most of the proposed solutions share similar properties, and thus in table 2 we unite them in a single row under the name SC. The only exception is DRFx model which implements a catch-fire semantics for racy programs and thus has slightly different properties.

End-to-end Sequential Consistency Marino *et al.* [32, 41] examined the performance penalties to ensure end-to-end SC (EtE-SC) enforced by (1) a modified SC-preserving version of LLVM compiler infrastructure and (2) a modified version of x86 hardware. To mitigate the induced overhead the authors utilized the observation that hardware need to enforce SC only for memory accesses to shared mutable variables. To classify memory regions as either thread-local, shared immutable, or shared mutable they have used a combination of a static compiler analysis and a dynamic analysis powered by modified hardware. They evaluated their approach on a number of benchmarks and reported performance overhead of 6.2% on average and 17% in maximum, compared to the stock LLVM compiler and x86 hardware.

Volatile-by-default Liu *et al.* [33, 42] studied sequential consistency in the context of the Java language. They proposed the *volatile-by-default* semantics (VbD), where each memory access is treated as volatile (*i.e.,* sequentially consistent) by default. The experiments showed a considerable performance penalty: 28% slowdown on average with 81% in maximum on x86 hardware, and 57% slowdown on average with 157% in maximum on Armv8 hardware. The authors tried to mitigate performance overhead and presented a novel optimization technique for the language-level SC compatible with *just-in-time* compilation. They propose to treat each object as thread-local speculatively and compile memory accesses without fences. If later during the execution concurrent accesses to the object are detected, the code is recompiled to place necessary fences. A modified version of the JVM which implements the technique described above was reported to incur 37% slowdown on average with 73% in maximum on Armv8 hardware.

SC-Haskell SC-Hs memory model [8] was inspired by the similar idea of separation between thread-local and shared mutable memory. To safely distinguish between the two the authors utilized the powerful strong type system of Haskell. The consequence of this approach is that programmers need to follow a stricter discipline in order to please the type checker. The authors modified the GHC to preserve SC and then run 1,279 benchmarks on x86-64 hardware to measure the performance penalties. They reported 0.4% geometric mean slowdown, and noticed that only 12 benchmarks experienced slowdown greater than 10%. These promising results can be partly explained by the fact that Haskell encourages a purely functional programming style, which minimizes the usage of shared mutable memory.

DRFx The DRFx [43,69] is another SC preserving memory model. In this model a runtime system is guaranteed to raise an exception if a program has data-races, and yield only sequentially consistent outcomes otherwise. In order to make the runtime data-race detection feasible in practice, the authors propose several modifications to existing hardware.

The authors claim that any sequentially valid optimization (*e.g.,* instruction reorderings or common subexpression elimination), is sound in DRFx model, the only limitation is that these transformations can only be performed withing bounds of compiler-designated program regions. Besides that any transformation that introduces speculative reads or writes is unsound, since speculative optimizations can bring data-races into otherwise race-free programs.

Note that in table 2 we still do not consider many of the transformations that claimed to be sound by the authors as actually being sound because of our convention described in §5. We consider transformations sound only if they are sound in a fully-defined semantics. The DRFx model does not meet this criterion as it provides catch-fire semantics.

The expected performance overhead of the model is reported to be 3.25% on average assuming an efficient implementation of the data-race detection in hardware. (compared to stock compiler and x86 hardware).

A.2 Total and Partial Store Order

In this section we consider PL memory models inspired by TSO and PSO.

Buffered Memory Model Demange *et al.* [44] presented the *Buffered Memory Model*, BMM for short, as a candidate model for the Java language. Their motivation, however, stemmed not from the desire to fully replace the Java memory model, but rather from a goal to build a verified version of Java Virtual Machine (akin to CompCertTSO project [70] for the C language). A more simple yet pragmatic TSO like memory model was considered as a first step to achieve this goal.

The authors proved soundness of several program transformations (including the store/load reordering, the speculative load introduction, and several others, see table 2) and the external DRF theorem. They also modified existing open-source implementation of JVM [71] to preserve BMM and reported only 1% average overhead compared to original version of the virtual machine. Again, the authors used only x86 hardware in their experiments, and the performance penalties are expected to be more significant on weaker hardware.

Relaxed Memory Models: an Operational Approach Boudol and Petri [45] proposed an operational approach to formal semantics of relaxed memory models (RMMOA) based on an abstract machine with a main memory and a hierarchical structure of store buffers with stores to different locations possibly propagating to the main memory out-of-order (similarly to PSO model). The authors present a proof of the external DRF theorem, but do not provide an extensive study of the soundness of program transformations.

A.3 Program Order Preserving

In this section we describe the memory models that preserve program order and forbid any kind of speculative executions to tackle the problem of thin-air values. In particular, we consider RC11 and several derivatives of this model, as well as the memory model of OCaml, and the proposed revised model of Java.

RC11 Lahav *et al.* [21] formalized a version of the C11 that preserves order between load/store pairs, and also repairs the semantics of sequentially-consistent accesses. The resulting model is commonly referred to as RC11.

The authors proved soundness of several program transformations (see table 2 for details). Among the unsound transformation, the load/store reordering is forbidden for an obvious reasons, the speculative load introduction is not supported because of the catch-fire semantics for racy programs, the CSE is not supported because relaxed accesses enforce coherence (non-atomic accesses support this transformation, but their usage entails undefined behavior in the presence of races).

The compilation mapping to x86 is not affected and remain optimal. One of the possible compilation mappings to the Arm and POWER architectures requires to compile a relaxed load as a plain load followed by a spurious conditional branch. Ou and Demsky [23] have studied the performance penalty of this mapping on Armv8 hardware. They modified the LLVM compiler framework to enforce the RC11 memory model by (1) adjusting the compiler optimization passes and (2) changing the compilation mappings. Several compilation schemes were considered, among them the one that uses a spurious conditional branch as described above has demonstrated the most promising results. The authors measured the running time on a set of benchmarks implementing concurrent data-structures (*e.g.,* locks, stacks, queues, deques, maps from various open source libraries [72–74]) and reported an overhead of 0% on average and 6.3% in maximum, compared to the unmodified version of the compiler.

RAR Doherty *et al.* [47] developed an operational version of the RC11 supporting release-acquire and relaxed accesses (RAR). On top of it they built proof calculus for invariant-based reasoning and verified correctness of mutual exclusion algorithms.

Operational RC11 Dang *et al.* [48] developed yet another operational version of the RC11 which they called ORC11. Their motivation was to then develop a new program logic and show it's soundness with respect to the ORC11 memory model. The program logic itself was utilized to prove correctness of several synchronization primitives from the standard library of the Rust [27].

Compositional Relaxed Concurrency Dodds *et al.* [46] proposed the compositional relaxed concurrency semantics (CRC) for the fragment of the C11 memory model, including non-atomic accesses with catch-fire semantics, release-acquire accesses, and sequentially-consistent fences. Based on this semantics the authors developed a tool for automatic verification of program transformations in the considered fragment of the C11 model. Since the relaxed fragment was not included, the authors avoided problems with thin-air values.

OCaml Memory Model Dolan *et al.* [22] developed a new memory model for the Multicore OCaml project. An important divergence of the OCaml memory model (OCMM) from the C11-like models is that the former has a weaker notion of the coherence. The choice of the weaker coherence was deliberate with the purpose to enable the common subexpression elimination (see §6.7.1 for details).

The authors also were the first to propose the local DRF property (lDRF), a strengthening of the external DRF (eDRF). While the latter requires an absence of data-races for a whole program as a prerequisite, the former bounds the effect of races to a portion of the program, thus enabling the compositional reasoning about the behavior of the program. The authors discovered that the lDRF property is not compatible with the load/store reordering. This fact forced them to forbid this transformation and adapt similar compilation scheme as for RC11.

Java Access Modes Bender and Palsberg [4] formalized a new revision of the Java Memory Model [75–77], which was developed to overcome the difficulties of the previous one [3] (see A.5 for details). The new version of the model was inspired by the RC11. It introduced a system of annotations on memory accesses, called "Java Access Modes" (hence the name of the model — JAM), similar to those present in the C11 like models. The new model adopted the RC11 solution to OOTA problem. It forbids load/store reorderings on the level of opaque (an analog of C/C++ relaxed) or stronger accesses. The model does not tackle the problem of thin-air values on the level of plain (*i.e.,* non-atomic) accesses.

A.4 Syntactic Dependencies Preserving

Next we discuss the programming language memory models that track syntactic dependencies.

Linux Kernel Memory Model LKMM [49] has adopted the idea to track syntactic dependencies in order to forbid thin-air values. Despite this choice limits the list of supported trace preserving transformations, in the context of the OS kernel development it can be justified by the following arguments. First, the Linux kernel targets a wide range of hardware architectures with a diverse set of memory models. To simplify the reasoning about the code, it is reasonable to pick a syntactic dependency preserving model which is conceptually close to those of hardware. Second, kernel developers already utilize various techniques to prevent certain compiler optimizations [49, 78, 79].

The authors of the model have empirically tested soundness of compilation mappings to x86, Armv7, Armv8, and POWER hardware. They also formalized the read-copy-update synchronization mechanism (RCU) [80] extensively used in the Linux kernel development, and proved soundness of its implementation with respect to their model.

Operational Happens-Before Model In attempt to repair the Java Memory Model (see appendix A.5) Zhang and Feng have proposed the operational happens-before model OHMM [50]. Their abstract machine consists of a global event buffer, where events might be reordered before they propagate into a global history based memory, and a replay mechanism used to simulate speculative executions. To avoid thin-air outcomes the model tracks syntactic dependencies between events and forbids the reordering of dependent events. The authors proved the external DRF and the soundness of several program transformations (see table 2).

Dependency Preserving Compiler Ou and Demsky [23] studied the performance penalty induced by dependency preserving compiler. Again, they modified the LLVM compiler infrastructure and run benchmarks from SPEC CPU2006 suite on Armv8 hardware. They have observed 3.1% overhead on average and 17.6% in maximum.

A.5 Semantic Dependencies Preserving

Next we discuss the memory models which try to tackle the thin-air values problem by developing a notion of semantic dependencies. In particular, this class includes the original Java Memory Model, the Promising semantics, and several models based on the *event structures* [81].

Java Memory Model The original version of the Java memory model JMM [3] was a pioneering work in the area of programming language memory models. In order to forbid thin-air outcomes, the memory model used a notion of *commit sequence*, *i.e.,* a sequence of partial execution graphs. The model was shown to adhere the external DRF [51]. However, the model failed to justify some program transformations that were expected to be sound [36] (*e.g.,* redundant load after load elimination, roach motel reordering, and others, see table 2 for details).

Generative operational semantics Jagadeesan *et al.* [52] attempted to fix JMM and proposed the generative operational semantics with speculative execution (GOS). To avoid thin-air values they have put stratification constraints on speculations. The authors prove the external DRF theorem. Also they verified a few program transformations (store/store reordering, load/load elimination, and roach motel reordering), but overall their study of transformations was not systematic.

Promising Semantics Kang *et al.* [17, 18] developed the Promising semantics (PRM). It is the most complete to this day model of the class of semantic dependency preserving models. Its key ingredient is a promising and certification machinery. During an execution, the abstract machine can non-deterministically *promise* to perform some store, it then has to *certify* the promise is feasible. The certification mechanism is defined in a way that forbids thin-air values to appear. The authors of the model have proven formally that Promising semantics admits many local and global program transformations, with a notable exception of the thread inlining (see table 2 for details).

Podkopaev *et al.* [62, 82] proved formally the soundness of standard optimal compilation mappings to x86, Armv7, Armv8, and POWER.

The model has a fully defined semantics for plain accesses. Plain and relaxed accesses, however, have different semantics. In particular, coherence is enforced only for relaxed accesses. This design choice, in particular, allows to support CSE on the level of plain accesses.

One of a few limitations of the Promising semantics is that it does not support sequentially consistent accesses.

Weakestmo Chakraborty and Vafeiadis [6, 19] developed a memory model based on the event structures (WMO). They utilize the event structures' capability to simultaneously encode multiple conflicting executions in order to model speculative executions. The model was shown to admit optimal compilation mappings [83], several program transformation, and the external DRF. Unlike Promising semantics it also supports sequentially consistent accesses.

A Concurrency Semantics for Relaxed Atomics Pichon-Pharabod and Sewell [25] presented the operational memory model (CSRA) consisting of a memory subsystem inspired by the POWER model and a thread subsystem, where each thread is represented as an event structure. At each step the abstract machine is allowed to either commit an event to the storage, or perform a transformation on one of the event structures. The authors have shown soundness of optimal compilation mappings to x86 and POWER, as well as soundness of several program transformations. It was later revealed though that the compilation scheme to Armv7 and Armv8 is not optimal [84].

Well-Justified Event Structures Jeffrey and Riely [24] proposed the memory model (WJES) based on event structures and a notion of *well-justification* of events inspired by the game semantics. Well-justification is used to prevent thin-air values and prove the external DRF. The authors do not study the soundness of program transformations in their model. They show, however, a counterexample demonstrating that the load/load reordering is unsound. This fact also implies that the compilation mappings to Armv7, Armv8, and POWER are not optimal.

Modular Relaxed Dependencies Paviotti *et al.* [20] constructed the denotational semantics based on the event structures (MRD). They employ the event structures to capture semantic dependencies between memory access events, which are in turn used to rule out thin-air outcomes. The authors prove the external DRF and the soundness of optimal compilation mappings, also they present a refinement relation which can be used to reason about validity of program transformations. However, they have not studied soundness of particular transformations.

A.6 Out of Thin-Air Values

Finally, we discuss memory models admitting thin-air values.

C11 The most notable member of the OOTA class is the C11 model [5]. The main purpose of the C11 model was to adhere to the fundamental principle of C/C++, *i.e.,* to provide so-called zero-cost abstraction. In other words, the memory model was meant to provide efficient compilation mappings and support as many transformation as possible. It was later revealed that the formal model partially fails to achive these goals.

Vafeiadis *et al.* [55] showed that several program transformation (load/store elimination, strengthening, roach motel reorderings, sequentialization) that deemed to be correct are actually unsound according to the formal model. They proposed several local fixes to the model which partly repair soundness of transformations and improve its meta-theoretical properties.

Batty *et al.* [38] showed that the model also fails to provide the external DRF guarantee, and that it is ultimately not possible to provide this guarantee at all within the style of the C11 formal semantics. Only the internal DRF can be proved for it.

A lot of work [5, 53, 54, 56] was dedicated to prove soundness of optimal compilation mappings with respect to formal models of hardware, and there the results were mostly positive. Besides that, Flur *et al.* [30] have extended the model to support mixed-size accesses. Finally, Nienhuis *et al.* [57] presented a formal executable semantics in terms of an abstract machine, equivalent to the C11 model.

JavaScript Memory Model The JSMM is based on the C11 model. Like the latter, it also has the problem of thin-air values and thus can only provide the internal DRF guarantee. Contrary to the C11, the JavaScript model does not treat racy non-atomic accesses as undefined behavior.

The main language primitive provided by the JSMM is `SharedArrayBuffer`, that is a linear mutable byte buffer. Thus the model naturally supports mixed-size accesses.

A calculus for relaxed memory Crary and Sullivan [58] proposed an alternative approach to the relaxed shared memory concurrency, which they called *Relaxed Memory Calculus* (RMC). Instead of deriving ordering constraints from annotations on memory accesses, they propose to directly specify the ordering between memory accesses in a source code. Their approach is highly generic and subsumes the traditional memory order annotations in the style of C11. Their model is very weak and permits thin-air values. Yet the authors proved the internal DRF theorem.

Relaxed Atomic + Ordering Saraswat *et al.* [59] presented the RAO memory model where relaxed behaviors are explained through transformations over a sequentially consistent execution. Authors claim their model provides the external DRF, but at the same time permits thin-air values. These two properties known to be incompatible [38]. We suppose that the external DRF can be achieved in their model only because of the fundamental restrictions on the input programming language (*e.g.,* the general conditional statements are not supported [25]).

A theory of speculative computation Boudol and Petri [40] proposed a general framework to study effects of speculative execution in shared memory setting (TSC). They have also noticed that the external DRF does not hold in the presence of unrestricted speculations, yet the internal DRF theorem still can be proven.