Extending the C/C++ Memory Model with Inline Assembly

PAULO EMÍLIO DE VILHENA, Imperial College London, United Kingdom
ORI LAHAV, Tel Aviv University, Israel
VIKTOR VAFEIADIS, MPI-SWS, Germany
AZALEA RAAD, Imperial College London, United Kingdom

Programs written in C/C++ often include *inline assembly*: a snippet of architecture-specific assembly code used to access low-level functionalities that are impossible or expensive to simulate in the source language. Although inline assembly is widely used, its semantics has not yet been formally studied.

In this paper, we overcome this deficiency by investigating the effect of inline assembly to the *consistency* semantics of C/C++ programs. We propose the first memory model of the C++ Programming Language with support for inline assembly for Intel's x86 including *non-temporal stores* and *store fences*. We argue that previous provably correct compiler optimizations and correct compiler mappings should remain correct under such an extended model and we prove that this requirement is met by our proposed model.

1 INTRODUCTION

Large software applications are rarely written in only one language. While the bulk of an application is typically written in a general-purpose programming language, such as C++, some parts are invariably written in higher-level domain-specific languages (for example, lexers and parsers, which generate C++ code) and others directly in assembly code of the underlying architecture(s).

The latter kind is directly supported by mainstream C/C++ compilers through *inline-assembly* blocks, which can be used (1) to expose some hardware instructions that are inaccessible or difficult to simulate in the source language, (2) to write prolog and epilog code of *naked* functions, and (3) to keep the ordering of instructions at compile time [18, 21]. As such, inline assembly constitutes an important tool of C/C++, whose significance is further attested by major projects such as the Linux kernel-based virtual machine (KVM) [6], and the GNU Compiler Collection (GCC) [22], both counting with thousands of occurrences of inline assembly.

Unlike some of the key features of C/C++, such as synchronization primitives, which have been the subject of many research papers [4, 14], and despite the extensive use of inline assembly, inline assembly lacks a *formal semantics*: a precise unambiguous specification.

In this paper, we overcome this deficiency and propose the first formal account of inline assembly. We distinguish three classes of inline-assembly instructions:

(1) Instructions, such as complex arithmetic and bit-manipulating operations and *single instruction/multiple data* [8] (SIMD) instructions, whose effect can be expressed in the source language (typically, as a sequence of arithmetic operations).
(2) Instructions accessing memory and/or enforcing ordering between instructions (such as *store fences* [11, Vol. 2B, §4]), whose effect cannot be expressed in the source language. Such instructions are commonly used in libraries for parallel and persistent programming, for efficient moving of data, and for communicating with external devices.
(3) Instructions that have a global effect and may completely change the semantics of the subsequent program, such as raising an interrupt, writing to the stack pointer register or to the page table entries [1, 28], and flushing the *translation lookaside buffer* [11, Vol. 2A, §3].

Authors' addresses: Paulo Emílio de Vilhena, p.de-vilhena@imperial.ac.uk, Imperial College London, London, United Kingdom; Ori Lahav, orilahav@tau.ac.il, Tel Aviv University, Tel Aviv, Israel; Viktor Vafeiadis, viktor@mpi-sws.org, MPI-SWS, Kaiserslautern, Germany; Azalea Raad, azalea.raad@imperial.ac.uk, Imperial College London, London, United Kingdom.

We narrow our scope to the second class of instructions for the Intel's x86 architecture, whose consistency and persistency semantics have been formalized by Raad et al. [24] in a model known as Ex86. We argue that supporting the first class of instructions is straightforward, raising no challenges beyond that of providing accurate semantics for the individual hardware instructions. In contrast, the second class affects the *memory consistency model* of the programming language, governing how concurrent programs are allowed to interact through shared memory. As we shall see, the effect of this class of instructions on the language's model leads to interesting semantic challenges. As for the third class of instructions, we declare them to be beyond the scope of this paper.

A particularly interesting use case of inline assembly are x86 *non-temporal stores* [11, Vol. 1, §10.4.6.2], an x86-specific feature that allows writing to memory while bypassing the cache. Non-temporal stores are used in cases of bulk memory writes [24], whose relative order is immaterial, such as initializing a memory page with zeros.

Unlike regular x86 stores, non-temporal stores can be reordered with other stores, and so the following C/C++ program with inline assembly, when compiled with `gcc` or `clang`, can exhibit the following quite surprising outcome (here and henceforth, we use pseudocode syntax with $x, y, \ldots$ being shared locations and $a, b, \ldots$ being thread-local registers; we assume that all locations are initialized to 0):

$$\begin{array}{l} \texttt{asm}\{[x] :=_{\texttt{nt}} 1\} \\ [y]^{\texttt{rel}} := 1 \end{array} \;\Big\|\; \begin{array}{l} a := [y]^{\texttt{acq}} \;//\; 1 \\ b := [x]^{\texttt{rlx}} \;//\; 0 \end{array} \quad \xrightarrow{\text{compile}} \quad \begin{array}{l} \texttt{movnt } [x], 1 \\ \texttt{mov } [y], 1 \end{array} \;\Big\|\; \begin{array}{l} \texttt{mov } a, [y] \;//\; 1 \\ \texttt{mov } b, [x] \;//\; 0 \end{array} \qquad \text{(MP-NT)}$$

Normally, C/C++ release-acquire accesses induce synchronization and thus anything executed before a release write is deemed to have happened before everything after an acquire read fulfilled by this write. Yet, this is no longer the case with inline assembly. Applying the standard compilation scheme of mapping C/C++ release/acquire/relaxed accesses to regular x86 accesses results in a x86 program that can read $a = 1 \wedge b = 0$. The only way to prevent the weak outcome is to add an appropriate instruction working as a fence between the two store instructions: a store fence (`sfence`) suffices, but one may also use a *memory fence* (`mfence`), a read-modify-write operation, or a plain x86 store to x. However, without a formal specification, such observations are unclear to developers, who naturally expect release/acquire synchronization to apply to all kinds of accesses. [1]

The question is how to provide an appropriate semantics for C/C++ programs with inline assembly, such as the previous example of MP-NT. In §2, we show that devising an appropriate semantics is by no means trivial. At the very least, one would require a solution that is:

- *flexible*, i.e. allowing arbitrary mixing of C/C++ and inline-assembly accesses with no partition on threads or memory locations that can or cannot use x86 instructions, since such restriction is not respected by most use cases of inline assembly;
- *supporting a representative set of x86 and C/C++ features* that have to do with accessing memory in a possibly concurrent setting;
- *preserving* the correctness of the existing *C/C++ compilation schemes* to x86 and of *local source-to-source code transformations*, since these are readily performed by C/C++ compilers;
- *precisely matching the x86 (resp. C/C++) model* for programs consisting purely of x86 (resp. C/C++) constructs. This last criterion acts as a sanity check ensuring that the semantics of existing C/C++ programs (without inline assembly) will not be affected by our proposed extension of the C/C++ concurrency model.

[1] Indeed, Program MP-NT illustrates one of the concerns in a recent Rust bug report: https://github.com/rust-lang/rust/issues/114582.

In addition, we would like our semantics to provide useful guarantees for common correct uses of inline assembly, such as the following variant of MP-NT, which rules out the weak outcome by inserting a store fence between the non-temporal store to x and the release write to y:

$$\begin{array}{l}\texttt{asm}\{[x]:=_{\texttt{nt}}1\}\\ \texttt{asm}\{\texttt{sfence}\}\\ [y]^{\texttt{rel}}:=1\end{array}\;\Big\|\;\begin{array}{l}a:=[y]^{\texttt{acq}}\;//\;\underline{1}\\ b:=[x]^{\texttt{rlx}}\;//\;\underline{0}\end{array}\quad\xrightarrow{\text{compile}}\quad\begin{array}{l}\texttt{movnt}\;[x],1\\ \texttt{sfence}\\ \texttt{mov}\;[y],1\end{array}\;\Big\|\;\begin{array}{l}\texttt{mov}\;a,[y]\;//\;\underline{1}\\ \texttt{mov}\;b,[x]\;//\;\underline{0}\end{array}\qquad\text{(MP-NT-SF)}$$

(In our examples, certain read instructions are followed by comments. When every comment is displayed in green, as $//\,v$, the annotated outcome can be observed on some architecture and should therefore be allowed by the model. When every comment is underlined and displayed in red, as $//\,\underline{v}$, the annotated outcome cannot be observed and should therefore be forbidden.)

As we explain in §2, many direct approaches to the problem of defining an appropriate semantics for C/C++ with inline Ex86-assembly fail one or more of the stated requirements.

In response, in §3, we develop a carefully designed extension of the C/C++ consistency model with support for the user-mode Ex86 inline-assembly instructions that access memory: namely, plain loads and stores, non-temporal stores, read-modify-write operations, and fences. We prove that our model is an extension of the Ex86 and C/C++ models, in the sense that plain x86 and plain C/C++ programs have unchanged semantics.

In §4, we prove that the established sound compilation schemes from C++ to Ex86 remain sound in spite of the presence of inline-assembly blocks, and that, similarly, so do the sound local source-to-source code transformations, such as reordering of independent memory loads. In addition, we introduce a new, provably sound, compilation scheme to Ex86, which compiles relaxed writes to non-temporal stores for the price of including some additional store fences (Definition 4.2).

2 OVERVIEW

In this section, we provide a gentle introduction to §3, where we formalize our contributions. To this end, in §2.1, we establish a series of desired properties that a model for C/C++ with inline assembly should enjoy. Then, in §2.2, we show why direct approaches for devising such a model do not work. Finally, in §2.3 and §2.4, we present an intuitive overview of our proposed model, showing how it satisfies the established desiderata.

2.1 Desiderata for a Hybrid Consistency Model for C/C++ and x86 Assembly

We argue that tentative "hybrid models" for C/C++ with support for inline Ex86-assembly should enjoy the following properties:

P0: Flexibility. As a first minimal requirement, we ask the hybrid model to support all the features of the respective C/C++ and x86 models, and to allow free mixing of the two. That is, we want to be able to write programs where threads can mix both C/C++ and inline-assembly instructions and where memory locations can be accessed using both types of instructions, as we have seen in the MP-NT and MP-NT-SF programs.

P1: Correctness of compiler mappings. In the weak-memory literature, a *compiler mapping*, or a *compilation scheme*, maps the memory operations of the source language to sequences of instructions of the target language that implement the corresponding high-level memory operation. Two standard compilation schemes from C/C++ to x86 exist [4, 14]: the *fence-after*-`sc`-*write* scheme, which places memory fences after `sc` writes; and the *fence-before*-`sc`-*read* scheme, which places memory fences before `sc` reads. Both schemes have been proven *correct* with respect to RC11 [14]: the compilation of a C/C++ program p following one of these schemes can only exhibit behaviors that are assigned to p by RC11. These schemes can be easily extended with support for inline

Ex86-assembly by simply mapping an inline-assembly instruction **asm**{s} to s. This mapping is in agreement with how current C/C++ compilers handle such instructions [15, Chapter 6.6]. It is therefore desirable that these schemes remain correct with respect to a hybrid model for C/C++ with inline Ex86-assembly.

P2: Correctness of standard compiler optimizations. To improve program performance, C/C++ compilers perform a sequence of local source-to-source transformations, whose correctness (in the absence of inline assembly) has been established by prior work [14, 30]. C/C++ compilers readily perform these transformation even when the program contains inline assembly. It is thus important that these transformations remain correct in any C/C++ model extended with inline assembly.

P3: Extension of source. For programs that do not use inline assembly, we want our model to coincide with the model of the source language. Concretely, we consider RC11 as the source model, and say that a model M is an *extension of RC11* if the semantics given by M to plain C/C++ programs agrees with the semantics given by RC11. If this property did not hold of a candidate hybrid model M, then plain C/C++ and C/C++ with support for inline assembly should be seen as different programming languages, because programs could have different semantics depending on whether RC11 or the hybrid model M is used. We see this distinction as artificial and compromising to the language.

P4: Extension of target. Analogously, we argue that a candidate hybrid model M should be an *extension of Ex86*: the semantics given by M to a C/C++ program p written entirely using inline Ex86-assembly should agree with the semantics given by Ex86 (to the obvious Ex86 program corresponding to p). The model M cannot give a stronger semantics to p than Ex86 because the compilation scheme of inline assembly is the straightforward identity map. Therefore, if there was a mismatch, then the model M would be necessarily assigning a more relaxed semantics to p than Ex86. This weakness in reasoning is undesirable.

P5: Architecture-specific guarantees for mixed programs. The RC11 model is sufficiently relaxed so as to support efficient compilation to multiple hardware architectures. This generality has the downside that RC11 may allow behaviors that cannot be observed by most implementations. The following program, for example, depicts such a behavior (known as *independent reads from independent writes* - IRIW):

$$[x]^{\mathtt{rel}} := 1 \;\Big\|\; \begin{array}{l} a := [x]^{\mathtt{acq}} \,//\, 1 \\ b := [y]^{\mathtt{rlx}} \,//\, 0 \end{array} \;\Big\|\; \begin{array}{l} c := [y]^{\mathtt{acq}} \,//\, 1 \\ d := [x]^{\mathtt{rlx}} \,//\, 0 \end{array} \;\Big\|\; [y]^{\mathtt{rel}} := 1 \qquad \text{(IRIW)}$$

This behavior is allowed by RC11 and observed when the program is run on the POWER [2] architecture. It illustrates that the two independent writes in the first and fourth threads can be observed in different orders by the second and third threads, even though the accesses in these two middle threads have to be executed in order (the acq access mode prevents reordering with subsequent accesses).

When, however, the IRIW program is compiled to x86 and to recent versions of Armv8 [23], the annotated weak outcome cannot be observed because these target architecture models provide the *multi-copy atomicity* guarantee, which postulates that any two writes must be observed by all threads, except the ones performing the two writes, in the *same* order. This multi-copy atomicity guarantee is a key property of the x86 and Armv8 architectures. It can be exploited to simplify reasoning about the correctness of a given program and, in some cases, to write more efficient ones.

The problem is that the RC11 model does not provide an efficient way of enforcing multi-copy atomicity even when the target architecture provides this guarantee. RC11, in fact, provides only two

	P0	P1	P2	P3	P4	P5
Hardware	★	⯪	☆	☆	★	★
Branching	★	⯪	☆	★	★	★
TSO-as-RA	☆	★	★	★	☆	☆
Projection	★	★	★	★	★	☆
Goens et al. [10]	☆	★	★	★	★	☆
Approach of §2.3	★	★	☆	★	★	★
Our approach	★	★	★	★	★	★

P0 - Flexibility
P1 - Correctness of compiler mappings
P2 - Correctness of compiler optimizations
P3 - Extension of RC11
P4 - Extension of Ex86
P5 - Strong guarantees for mixed programs

Fig. 1. Comparison of approaches according to several desired properties.

ways to forbid the weak behavior of IRIW, both of which incur an non-negligible implementation cost on x86. One can either (1) strengthen all access modes to sc, or (2) insert an sc fence between the two pairs of read operations. In the context of x86, both solutions are unsatisfactory, as they involve additional unnecessary fences. With the support for inline Ex86-assembly, one could imagine a third solution that consists in strengthening the first read operation of each thread as follows:

$$[x]^{\mathtt{rel}} := 1 \;\Big\|\; \begin{array}{l}\mathtt{asm}\{a := [x]\}\\ b := [y]^{\mathtt{rlx}}\end{array} \;\Big\|\; \begin{array}{l}\mathtt{asm}\{c := [y]\}\\ d := [x]^{\mathtt{rlx}}\end{array} \;\Big\|\; [y]^{\mathtt{rel}} := 1 \qquad \text{(IRIW-TSO)}$$

One would expect this solution to work because (similar to acq accesses) Ex86 disallows the reordering of a read operation with any other subsequent operation. This solution avoids the emission of fences and highlights the reliance on an architecture-specific guarantee.

2.2 Evaluation of Candidate Models

We now consider multiple tentative hybrid models and evaluate them according to our established criteria. Figure 1 contains a summary of our discussion. The candidate models are organized by lines, and the desired properties by columns. A full star means that a model enjoys the corresponding property; an empty star means that it does not; a half star means that the property is partially met.

Hardware approach. The hardware approach is perhaps the first and simplest solution that comes to mind: it consists of using the hardware model Ex86 itself as the hybrid model. This seems like a plausible solution, because a program that uses inline Ex86-assembly can only be executed on this specific architecture. However, one immediate deficiency of this approach is that the Ex86 model is not directly applicable to a C/C++ program; one would first have to consider its compilation to Ex86 and only then apply the hardware model. As a consequence, one would have to commit to one of the compilation schemes to Ex86. Therefore, under this approach, the correctness of standard compilation mappings would not hold in general. Another downside is that this model is not an extension of RC11: the semantics of a program under Ex86 can clearly disagree from that given by RC11. Finally, this approach would not validate standard compilation optimizations as many of them, such as reordering of independent reads, is unsound under Ex86.

Branching approach. A slight refinement of the hardware approach is to branch on whether the program uses inline assembly: if it does, then the semantics is given by Ex86; otherwise, the semantics is given by RC11. This approach improves on the previous one by constituting an extension of RC11, however most compiler optimizations would still be unsound in programs with inline assembly.

The TSO-as-RA approach. The next approach is to keep the RC11 model, and to simply map each inline assembly instruction to an existing C/C++ construct with the same or slightly weaker semantics. In particular, plain Ex86 stores can be mapped to RC11 `rel` stores, plain Ex86 loads can be mapped to RC11 `acq` loads, Ex86 memory fences to RC11 `sc` fences, and Ex86 store fences to RC11 `acqrel` fences.

This approach has three major downsides. First, it does not give any semantic benefit to using inline assembly (P5). Second, it does not match the Ex86 semantics for programs consisting purely of inline assembly (P3). For example, consider a version of IRIW written entirely using inline assembly; that is, using inline-assembly reads and writes instead of C++ reads and writes. According to the TSO-as-RA approach, this inline-assembly version of IRIW can exhibit the annotated behavior of IRIW, even though, in practice, it can never be observed. Third, the TSO-as-RA approach cannot model all relevant Ex86 features. In particular, it cannot model Ex86 non-temporal stores because there is no corresponding RC11 store construct that permits the weak behavior of MP-NT from §1.

Projection approach. Given that neither Ex86 nor RC11 alone are appropriate for ascribing semantics to C/C++ programs with inline assembly, a natural choice is to use both models together.

At a very high level, the two models seem compatible: they are defined in a *declarative style* as a set of constraints that program executions should satisfy. For instance, RC11 states that a read operation cannot *happen before* the write instruction from which it reads. An instruction is said to happen before another one (1) if it appears earlier in the same thread, or (2) if it appears before some release-acquire synchronization, such as seen in the example of MP-NT. Ex86, on the other hand, imposes multi-copy atomicity: the order in which independent writes are observed is the same across all threads (except the ones performing those writes as they may observe their own writes early).

A natural definition for a combined model would be to take the conjunction of the constraints of the two models, each applied only to the instructions of the corresponding model. In other words, to apply the Ex86 constraints to the inline-assembly instructions and the RC11 constraints to the RC11 accesses. Such a definition is clearly an extension of RC11 and Ex86. Moreover, it supports the existing compilation schemes and compiler optimizations. It fails, however, to provide useful semantics for programs with inline assembly: for instance, it does not rule out the weak behaviors of the MP-NT-SF and IRIW-TSO programs, because it does not rule out cycles with accesses from both models.

Compound memory model approach. Goens et al. [10] propose another way of combining two memory models based on operational semantics, where each thread follows a single operational memory model. Their approach is, however, not applicable to the setting of inline assembly because it is too inflexible: it does not allow the use of both x86 and C/C++ instructions in the same thread.

2.3 Towards a Good Hybrid Model

From the approaches seen so far, only the projection approach came close to achieving our desiderata for a hybrid memory consistency model. To arrive at a good hybrid model, we will therefore start with the projection approach and refine it to strengthen the guarantees given to programs containing both C/C++ accesses and inline x86-assembly.

Supporting correct message-passing patterns. The first necessary strengthening comes from carefully inspecting the MP-NT and MP-NT-SF examples. RC11 forbids the weak behavior of the corresponding programs with only C/C++ accesses with its *coherence* condition, which says that the *extended coherence order* (eco) cannot contradict the model's *happens-before* relation (hb).

The extended coherence order, eco, orders accesses at a given memory location in the order they appear to have executed. For instance, it places all writes to the same location, say x, in a total order. A read r to x is placed by eco after the write w from which r reads and before every other write that follows w according to eco itself. In the executions leading to the annotated outcomes of MP-NT and MP-NT-SF, eco orders the write to x before the read to x (as the latter reads the initialization value, 0) and orders the write to y after the read to y (as the latter reads from the former).

The happens-before relation, hb, defined as $(\mathrm{po} \cup \mathrm{sw})^+$, is given as the transitive closure of the union of two components: program-order edges (po, relating instructions of the same thread in the order they appear in the program) and synchronization edges (sw) between threads, when one thread reads from another in a synchronizing fashion (e.g., using `rel`/`acq` accesses). In our example, the write to y synchronizes with the read to y, and thus the previous write to x happens before the read to x according to RC11, and so the read to x cannot read 0.

Clearly, to regain soundness in the model with inline assembly, we need to adapt the definition of hb to exclude program-order edges from non-temporal stores to subsequent stores because these can be reordered by x86. Blindly restricting the definition of hb to relate only C/C++ events (as in the projection approach) is too weak because the behavior of MP-NT-SF would then be allowed. A suitable definition is thus to remove from hb only the po edges between a non-temporal store and any later instruction that is not a fence. That is, we redefine hb as $(\mathrm{po}_{\mathrm{RC11}} \cup \mathrm{sw})^+$, where the relation $\mathrm{po}_{\mathrm{RC11}}$ excludes such po edges (see §3).

Supporting stronger architecture-specific behaviors. Next, we also need to strengthen the model to support the IRIW-TSO example. If all accesses in the example were x86 accesses, Ex86 would forbid this outcome by its general acyclicity condition which forbids cycles consisting of external eco edges (i.e. ones between accesses from different threads) and its *preserved program order* (ppo), which includes the program-order edges between instructions whose ordering is guaranteed on x86 (e.g., from x86 reads to all subsequent memory instructions).

A minimal way to extend the applicability of this condition would be to require the cycle to contain at least one inline-x86-assembly instruction. Requiring at least one assembly instruction in the cycle prevents this new condition from breaking Property P3: the additional condition simply does not apply to program without inline assembly. Moreover, it ascribes the intended semantics to the IRIW-TSO program, forbidding its annotated weak outcome.

Sadly, however, this minimal way of adapting the Ex86 model is flawed as it does not validate compiler optimizations. To see this, consider the following variant of IRIW-TSO:

$$[x]^{\mathtt{rlx}} := 1 \;\Big\|\; \begin{array}{l}\mathbf{asm}\{a := [x]\} \;//\; \underline{1}\\ b := [y]^{\mathtt{rlx}} \;//\; \underline{0}\end{array} \;\Big\|\; \begin{array}{l}c := [y]^{\mathtt{rlx}} \;//\; \underline{1}\\ d := [x]^{\mathtt{rlx}} \;//\; \underline{0}\end{array} \;\Big\|\; [y]^{\mathtt{rlx}} := 1 \qquad \text{(IRIW-TSO-2)}$$

The annotated behavior is disallowed under this model because the cycle contains one inline-assembly instruction. However, a C/C++ compiler can reorder the accesses of the third thread and arrive at the following program:

$$[x]^{\mathtt{rlx}} := 1 \;\Big\|\; \begin{array}{l}\mathbf{asm}\{a := [x]\} \;//\; 1\\ b := [y]^{\mathtt{rlx}} \;//\; 0\end{array} \;\Big\|\; \begin{array}{l}d := [x]^{\mathtt{rlx}} \;//\; 0\\ c := [y]^{\mathtt{rlx}} \;//\; 1\end{array} \;\Big\|\; [y]^{\mathtt{rlx}} := 1$$

The depicted outcome is now allowed: first $d := [x]^{\mathtt{rlx}}$ reads 0, then the first and second threads execute, then the fourth thread writes 1 to y, which is finally read by the third thread.

2.4 Our Approach

The IRIW-TSO-2 counterexample shows that it is too strong to stipulate the absence of cycles that violate Ex86-consistency and contain at least one Ex86 event. The weak behavior of IRIW-TSO-2

should be allowed by our model so as to validate the reordering of RC11 relaxed accesses on the third thread of the program.

In order to allow the annotated behavior of IRIW-TSO-2, our idea is to insist that *all* ppo *edges* in the (ppo ∪ eco)-cycle contain at least one x86 instruction or a C/C++sc fence. This is because neither x86 instructions nor C/C++sc fences can be optimized away by the compiler in a thread-local fashion. Therefore, the third thread of IRIW-TSO-2 cannot contribute to the cycle that violates Ex86-consistency, because it contains only plain C/C++ instructions.

Extending RC11 with this refined condition leads to a hybrid model that enjoys all our established desiderata: (1) it supports the established compilation schemes to Ex86; (2) it supports all existing local compiler optimizations, because these only affect C/C++ operations, and thus do not affect our model's preserved program order relation, which must include an assembly instruction or a sc fence; (3) it extends both RC11 and Ex86; and (4) it provides the intended semantics to Program MP-NT and to all variants of Program IRIW that we have encountered.

3 THE EXTENDED MODEL

In this section, we present our extension of C++'s memory model with support for inline Ex86-assembly. We use RC11 [14] as the memory model for C++. With the interest of recalling the basic notions of RC11 and setting up notation and useful definitions for the next subsections, we start with a brief presentation of RC11. We mainly follow its original introduction by Lahav et al. [14]. We also rely on Podkopaev et al. [20] for the precise definition of the construction of *execution graphs*.

3.1 The RC11 Memory Model

RC11 defines the semantics of multithreaded C/C++ programs. More specifically, RC11 formalizes how the memory, which initially maps every location to a default value (usually the integer 0), is updated after the execution of a program. To account for non-determinism (for example, due to the concurrent execution of threads), the model associates a program p not with a single final memory, but with the set of states in which the memory can be found after the execution of p.

The RC11 model follows the *declarative approach*. In the declarative approach, the set of final memory states associated with a program p is defined in three steps. The first step consists in an operational semantics; that is, a formalization of program execution. However, this formalization does not strive to capture exactly how the program p runs. Instead, it follows a simple *thread-interleaving* semantics where threads non-deterministically take turns and contribute to the construction of an abstract structure called an execution graph. An execution graph stores, in the form of nodes, the memory operations (such as writes, reads, and synchronization barriers) issued by threads. These nodes are also called *events*. The result of the first step is thus the construction of a set of execution graphs associated with p. The second step is the collection, among this resulting set of execution graphs, of every *consistent* execution graph. A consistent execution graph is one whose nodes can be connected by extra relations in a way that satisfies conditions postulated by the model in question. These conditions capture how the model deviates from one that would tolerate only sequentially consistent behaviors. The third and final step amounts to mapping every consistent execution graph to the memory state it represents.

To illustrate the RC11 model, we introduce RC11-lang, a simple concurrent imperative programming language with support for C++'s memory-access modes. Opting for a simple set of programming constructs allows us to concentrate on the key aspect of the memory model: the definition of the semantics of memory operations such as read, writes, and synchronization barriers.

Figure 2 shows the syntax of RC11-lang. The language is parametric on a set of registers, *Reg*, and introduces a set of (preallocated) memory locations, *Loc*, defined as the set of natural numbers.

Syntax of expressions, commands, and access modes

$$\begin{aligned} \mathit{Expr} \ni e &::= n\ (\in \mathbb{N}) \mid r\ (\in \mathit{Reg}) \mid \ell\ (\in \mathit{Loc} \triangleq \mathbb{N}) \\ &\quad \mid e + e \mid e - e \mid e * e \\ \mathit{Cmd} \ni s &::= r := [e]^{md} \mid [e]^{md} := e \mid r := \mathtt{rmw}_{md}\,([e],\, e,\, e) \\ &\quad \mid \mathtt{fence}_{md} \mid \mathtt{if}\ e\ \{\ s\ \} \mid \mathtt{while}\ e\ \{\ s\ \} \mid s; s \mid \mathtt{skip} \\ \mathit{Mode} \ni md &::= \mathtt{na} \mid \mathtt{rlx} \mid \mathtt{rel} \mid \mathtt{acq} \mid \mathtt{acqrel} \mid \mathtt{sc} \end{aligned}$$

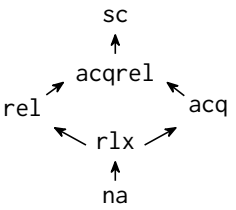


Fig. 2. Syntax of RC11-lang.

Expressions e are used to compute numbers n or locations ℓ by reading numbers stored in registers r and performing arithmetic operations. The syntactic category of commands, *Cmd*, includes `if` branching, `while` loops, sequential composition, a `skip` instruction, and memory operations, such as reads, writes, read-modify-writes (RMWs), and fences. The notation `[`e`]` is used to indicate that e denotes a memory location rather than a number. Every memory operation carries an access mode *md*. Access modes are ordered according to the diagram depicted in Figure 2. To give an (over-simplistic) intuitive explanation of access modes, we can say that `sc` operations follow a sequentially consistent semantics, and operations with a weaker access mode *md* follow a semantics that deviates from sequential consistency to a degree that is proportional to how distant *md* is from `sc`. Only certain access modes are permitted per operation:

- Modes `na`, `rlx`, `rel`, and `sc` apply to writes.
- Modes `na`, `rlx`, `acq`, and `sc` apply to reads.
- Modes `acq`, `rel`, `acqrel`, and `sc` apply to fences.
- Modes `rlx`, `acq`, `rel`, `acqrel`, and `sc` apply to RMWs.

Finally, a program $p \in \mathit{Prog}$ is defined as a collection of commands, represented as a map from numbers (or *thread identifiers*) to commands: $\mathit{Prog} \triangleq \mathbb{N} \xrightarrow{\text{fin}} \mathit{Cmd}$.

We formalize an *event* as either an *initialization event* $I(\ell)$, representing the initialization of ℓ with the default value 0; or a pair of natural numbers $(i,\ j)$, where i is a thread identifier and j is the order of this event with respect to the events emitted by thread i. (These numbers are used, for example, in the definition of the *program-order* relation.) An execution graph is represented as a pair of a set of events `E` and a map `lab` from events to *labels*. A label specifies both the type of a memory event (whether it is a read, a write, a read-modify-write, or a fence) and its arguments. A read label is represented as $\mathtt{R}^{md}(\ell,\ n)$; a write label is represented as $\mathtt{W}^{md}(\ell,\ n)$; a fence is represented as $\mathtt{F}^{md}$; and a read-modify-write label is represented as $\mathtt{RMW}^{md}(\ell,\ n,\ m^?)$, where $m^?$ denotes either a number or the marker $\bot$ representing the case of a failed read-modify-write operation. We are often lax about the distinction between events and labels, and use them interchangeably. Moreover, we write `R`, `W`, `F`, and `RMW` to denote respectively the sets of events whose label is a read, a write, a fence, and a read-modify-write. We further distinguish `RMW` into the set of successful read-modify-writes `RMW-s` and failed read-modify-writes `RMW-f`.

The construction of the set of execution graphs associated with a program relies on the notions of *threads* and *thread pools*. A thread pool is modeled as a finite map from thread identifiers to threads. A thread, in its turn, is modeled as a tuple containing the following fields: `reg_st`, which maps a register to the number it stores; `ev_counter`, which stores the number of events issued by the thread; and `next_cmd`, which stores the next command to be executed by the thread. In sum, here is the definition of the set of threads, *Thread*, and the set of thread pools, *Pool*:

$$P \in \mathit{Pool} \triangleq \mathbb{N} \xrightarrow{\text{fin}} \mathit{Thread} \qquad t \in \mathit{Thread} \triangleq \begin{Bmatrix} \mathtt{reg\_st} : \mathit{Reg} \to \mathbb{N}; \\ \mathtt{ev\_counter} : \mathbb{N}; \\ \mathtt{next\_cmd} : \mathit{Cmd} \end{Bmatrix}$$

Pool reduction $\boxed{P \,/\, G \longrightarrow P \,/\, G}$

$$\textsc{ReadStep}\quad \frac{P[i].\begin{cases}\texttt{reg\_st} & = \phi\\ \texttt{ev\_counter} & = j\\ \texttt{next\_cmd} & = r := [e]^{md};\, s\end{cases} \qquad \ell = [\![\, e \,]\!]_\phi \qquad a = (i,\, j) \qquad P' = P\left[i := P[i].\begin{cases}\texttt{reg\_st} & := \phi[r := n]\\ \texttt{ev\_counter} & := j + 1\\ \texttt{next\_cmd} & := s\end{cases}\right] \qquad G' = G.\begin{cases}\mathsf{E} & := G.\mathsf{E} \uplus \{a\}\\ \texttt{lab} & := G.\texttt{lab}[a := \mathsf{R}^{md}(\ell,\, n)]\end{cases}}{P \,/\, G \longrightarrow P' \,/\, G'}$$

$$\textsc{TerminateStep}\quad \frac{P[i].\texttt{next\_cmd} = \texttt{skip} \qquad P' = \lambda j \in \mathit{dom}(P) \setminus \{i\}.\ P[j]}{P \,/\, G \longrightarrow P' \,/\, G}$$

Fig. 3. Definition of pool reduction.

On top of these definitions, the set of candidate execution graphs associated with a program is captured by the *pool reduction* relation, a relation between pairs of pools and execution graphs. It is noted $P \,/\, G \longrightarrow P' \,/\, G'$. Intuitively, the statement $\mathit{toPool}(p) \,/\, \mathit{Init} \longrightarrow^* \emptyset \,/\, G$ express that G is an execution graph associated with p. The graph Init in this statement denotes the *initial execution graph*, a graph where $\mathit{Init}.\mathsf{E}$ contains initialization events $I(\ell)$ for every location ℓ, and $\mathit{Init}.\texttt{lab}$ maps $I(\ell)$ to $\mathsf{W}^{\mathsf{na}}(\ell,\, 0)$. The pool $\emptyset$ denotes a thread pool of empty domain. The pool $\mathit{toPool}(p)$ denotes a thread pool in its initial state:

$$\mathit{toPool}(p) \;\triangleq\; \lambda i \in \mathit{dom}(p).\ \left\{\texttt{reg\_st} = \lambda\_.0;\ \texttt{ev\_counter} = 0;\ \texttt{next\_cmd} = \mathit{prog}(i);\ \texttt{skip}\right\}$$

Figure 3 includes some illustrative cases of the pool reduction relation. The complete definition can be found in the Appendix (§A). Some cases rely on the interpretation of an expression e under a map ϕ from registers to numbers. This interpretation, noted $[\![\, e \,]\!]_\phi$, is simply defined as the interpretation of the syntactic arithmetic operators as their mathematical counterpart. Rule ReadStep shows how a new read event a is added to the execution graph when a read operation is executed. There is no restriction on the value n returned by the read operation. It is only at the level of execution graphs that consistency conditions are imposed and certain values are ruled out. Rule TerminateStep shows how completed threads are discarded from the pool. Eventually, all threads complete their execution and the pool degenerates to $\emptyset$.

To define RC11's notion of a consistent execution graph, we need to introduce the program-order relation po and we need to consider the extension of an execution graph with a *reads-from* relation rf and a *modification-order* relation mo. We are often lax about the distinction between the execution graph G and the extended tuple $(G,\, \mathsf{rf},\, \mathsf{mo})$.

Notation. The metavariables a, b, c, d, and e range over events, which, as we recall, are formalized as either initialization events $I(\ell)$ or as pairs of natural numbers $(i,\, j)$, where i is a thread identifier and j is the order of the event. The terms $a.1$ and $a.2$ denote the first and second projections of a in the case a is a pair. The relation R^{-1} is the *inverse relation* of R: $(b,\, a) \in R^{-1} \iff (a,\, b) \in R$. The relation $R_1;\, R_2$ is the *sequential composition* of R_1 and R_2: $(a,\, c) \in R_1;\, R_2 \iff \exists b.\ (a,\, b) \in R_1 \wedge (b,\, c) \in R_2$. The relation $[S]$ is the smallest reflexive relation on a set S; it is defined as $\{(s,\, s) \mid s \in S\}$. The relations $R^?$, R^+, and R^* respectively denote the reflexive closure, the transitive closure, and the reflexive-and-transitive closure of R. The relations R_i and R_e are the *internal* and *external*

components of R: $(a, b) \in R_i \iff (a, b) \in R \wedge a.1 = b.1$, and, $R_e = R \setminus R_i$. Given a graph G, the relation R_ℓ is the *at-ℓ* restriction of R: it restricts R to events a such that $G.\mathtt{lab}(a)$ accesses ℓ. The term $a.\mathtt{loc}$ denotes the location accessed by a. The relation $R|_{\mathsf{loc}}$ is the *per-location* restriction of R: $(a, b) \in R|_{\mathsf{loc}} \iff (a, b) \in R \wedge a.\mathtt{loc} = b.\mathtt{loc}$. The relation $R|_{\neq\mathsf{loc}}$ is the *distinct-locations* restriction of R: $R|_{\neq\mathsf{loc}} = R \setminus R|_{\mathsf{loc}}$.All these restrictions can be similarly applied to sets. The graph G is usually clear from the context and left implicit.

Program order. The program order reflects the order in which events were emitted by a given thread: $(a, b) \in \mathsf{po} \iff (a = I(\_) \wedge b \neq I(\_)) \vee (a.1 = b.1 \wedge a.2 < b.2)$.

Reads-from. The reads-from relation relates write events to read events, $\mathsf{rf} \subseteq (\mathtt{W} \cup \mathtt{RMW\text{-}S}) \times (\mathtt{R} \cup \mathtt{RMW})$. It captures how information flows from a write to a read on the same location. There are two conditions. First, for every read b, there must be a unique write a such that $(a, b) \in \mathsf{rf}$. (This condition can be equivalently stated as the conjunction of the inclusions $[\mathit{codom}(\mathsf{rf})] \subseteq \mathsf{rf}^{-1}; \mathsf{rf}$ and $\mathsf{rf}; \mathsf{rf}^{-1} \subseteq [\mathit{dom}(\mathsf{rf})]$.) Second, for every pair $(a, b) \in \mathsf{rf}$, the events a and b must act on the same location and the value read by b must be equal to the value written by a.

Modification order. The modification order is a relation on write and successful read-modify-write events, $\mathsf{mo} \subseteq (\mathtt{W} \cup \mathtt{RMW\text{-}S}) \times (\mathtt{W} \cup \mathtt{RMW\text{-}S})$. Intuitively, it describes how single memory cells have been observed to evolve during program execution. The mo relation is equal to the disjoint union of the relations mo_ℓ, defined as the restriction of mo to events in ℓ: $\mathsf{mo} = \biguplus_{\ell \in Loc} \mathsf{mo}_\ell$. Moreover, for every ℓ, the relation mo_ℓ is a *strict total order* (transitive, irreflexive, and total).

We are finally in position to introduce the RC11-consistency conditions:

Definition 3.1 (RC11-Consistency). An execution graph $(G, \mathsf{rf}, \mathsf{mo})$ is *RC11-consistent* if the conditions

- $\mathit{irreflexive}(\mathsf{hb}; \mathsf{eco}^?)$ (Coherence)
- $\mathit{acyclic}(\mathsf{psc})$ (SC)
- $\mathit{irreflexive}(\mathsf{rb}; \mathsf{mo})$ (Atomicity)
- $\mathit{acyclic}(\mathsf{po} \cup \mathsf{rf})$ (No-Thin-Air)

hold, where the relations *happens-before* (hb), *synchronizes-with* (hb), *extended coherence order* (eco), *reads-before* (rb), *partial-SC* (psc), and *SC-before* (scb) are defined as follows:

$$\mathsf{rb} \triangleq (\mathsf{rf}^{-1}; \mathsf{mo}) \setminus [\mathtt{E}] \qquad \mathsf{hb} \triangleq (\mathsf{po} \cup \mathsf{sw})^+ \qquad \mathsf{eco} \triangleq (\mathsf{rf} \cup \mathsf{mo} \cup \mathsf{rb})^+$$

$$\mathsf{sw} \triangleq \begin{cases} [\mathtt{E}^{\sqsupseteq \mathtt{rel}}]; ([\mathtt{F}]; \mathsf{po})^?; [\mathtt{W}^{\sqsupseteq \mathtt{rlx}}]; \\ \quad \mathsf{rf}^+; \\ [\mathtt{R}^{\sqsupseteq \mathtt{rlx}}]; (\mathsf{po}; [\mathtt{F}])^?; [\mathtt{E}^{\sqsupseteq \mathtt{acq}}] \end{cases}$$

$$\mathsf{psc} \triangleq \mathsf{psc}_{\mathsf{base}} \cup \mathsf{psc}_{\mathsf{fence}}$$

$$\mathsf{scb} \triangleq \mathsf{po} \cup \mathsf{po}|_{\neq\mathsf{loc}}; \mathsf{hb}; \mathsf{po}|_{\neq\mathsf{loc}} \cup \mathsf{hb}|_{\mathsf{loc}} \cup \mathsf{mo} \cup \mathsf{rb}$$

$$\mathsf{psc}_{\mathsf{base}} \triangleq ([\mathtt{E}^{\mathtt{sc}}] \cup [\mathtt{F}^{\mathtt{sc}}]; \mathsf{hb}^?); \mathsf{scb}; ([\mathtt{E}^{\mathtt{sc}}] \cup \mathsf{hb}^?; [\mathtt{F}^{\mathtt{sc}}])$$

$$\mathsf{psc}_{\mathsf{fence}} \triangleq [\mathtt{F}^{\mathtt{sc}}]; (\mathsf{hb} \cup \mathsf{hb}; \mathsf{eco}; \mathsf{hb}); [\mathtt{F}^{\mathtt{sc}}]$$

These consistency conditions are equivalent to the ones formulated by Margalit and Lahav [17], who diverge from Lahav et al. [14] only in a minor way: the synchronizes-with relation relies on a simplified notion of *release sequences*, defined as the reflexive-and-transitive closure of rf. This simplification is in agreement with the current model of the C++ programming language [5]. We further adapt the statement of Atomicity according to our design choice of modeling RMWs as single events rather than as pairs of reads and writes related by an extra relation rmw.

To complete the description of RC11, showing how it defines the semantics of a program, we need to introduce the notions of a *data race* and of *undefined behavior* (*UB*):

$$\begin{array}{l} \mathit{Cmd} \ni s ::= \ldots \mid \mathbf{asm}\{r := [e]\} \mid \mathbf{asm}\{[e] := e\} \mid \mathbf{asm}\{r := \mathtt{rmw}\,([e],\, e,\, e)\} \mid \mathbf{asm}\{\mathtt{mfence}\} \\ \qquad\qquad \mid \mathbf{asm}\{[e] :=_{\mathtt{nt}} e\} \mid \mathbf{asm}\{\mathtt{sfence}\} \end{array}$$

Fig. 4. Syntax of RC11$^{\mathsf{Ex86}}$-lang.

Definition 3.2 (Data Race). A pair of events (a, b) forms a *data race* if the following conditions hold: (1) $a \neq b$, (2) $a.\mathtt{loc} = b.\mathtt{loc}$, (3) $\{a.\mathtt{md}, b.\mathtt{md}\} \cap (\mathtt{W} \cup \mathtt{RMW\text{-}s}) \neq \emptyset$, and (4) $(a, b) \notin \mathsf{hb} \cup \mathsf{hb}^{-1}$.

Definition 3.3 (RC11-Behaviors).

$$\begin{array}{l} \left(\mathit{toPool}(p) \;/\; \mathit{Init} \longrightarrow^{*} \emptyset \;/\; G \;\wedge\; (G, \mathsf{rf}, \mathsf{mo})\ \mathit{is\ RC11\text{-}consistent}\right) \vdash p \longrightarrow (G, \mathsf{rf}, \mathsf{mo}) \\ \left(\begin{array}{l} \mathit{toPool}(p) \;/\; \mathit{Init} \longrightarrow^{*} \_ \;/\; G \;\wedge\; (G, \mathsf{rf}, \mathsf{mo})\ \mathit{is\ RC11\text{-}consistent} \\ \qquad \wedge\; (a, b)\ \mathit{forms\ a\ data\ race} \;\wedge\; \mathtt{na} \in \{a.\mathtt{md}, b.\mathtt{md}\} \end{array}\right) \vdash p \longrightarrow \mathit{UB} \end{array}$$

Each consistent execution graph $(G, \mathsf{rf}, \mathsf{mo})$ represents one of the possible final memory states of a program. We use the function *finalSt* to extract this memory state: $\mathit{finalSt}(G, \mathsf{mo})$ denotes the memory where a location ℓ stores the value n of the last write event $\mathtt{W}(\ell, n)$ in G with respect to mo. The memory $\mathit{finalSt}(G, \mathsf{mo})$ is represented as a partial map where a location ℓ belongs to $\mathit{dom}(\mathit{finalSt}(G, \mathsf{mo}))$ iff there exists $a \neq I(\_)$ such that $G.\mathtt{lab}(a) \in \mathtt{W}_\ell \cup \mathtt{RMW\text{-}s}_\ell$.

Definition 3.4 (RC11-lang Semantics). The semantics of a RC11-lang program p is defined as its set of final states:

$$\sigma \in [\![\, p \,]\!]_{\mathrm{RC11}} \iff p \longrightarrow \mathit{UB} \;\vee\; \exists G, \mathsf{rf}, \mathsf{mo}.\ p \longrightarrow (G, \mathsf{rf}, \mathsf{mo}) \;\wedge\; \sigma = \mathit{finalSt}(G, \mathsf{mo})$$

3.2 The RC11$^{\mathsf{Ex86}}$ Memory Model - An Extension of RC11 with Inline Ex86-Assembly

We now introduce RC11$^{\mathsf{Ex86}}$, an extension of RC11 with inline Ex86-assembly. We illustrate the model in an extension of the language RC11-lang with inline assembly, called RC11$^{\mathsf{Ex86}}$-lang.

Figure 4 shows the syntactical increments of RC11$^{\mathsf{Ex86}}$-lang over RC11-lang. The main difference with respect to Figure 2 is the addition of inline-assembly commands, distinguished by the prefix **asm**. They allow one to access the following Ex86-specific instructions: *plain Ex86 reads, writes, and read-modify-writes*; *non-temporal stores*; *store fences*; and *memory fences*.

To give an intuitive operational account of these instructions, we can rely on the formal operational model of Ex86 [24]. In this operational model, every thread contains a local cache where write instructions first take effect before reaching the global main memory, which is shared among all threads. A non-temporal store $[e] :=_{\mathtt{nt}} e'$ bypasses the local cache if the cache contains no writes to the same location. Therefore, a non-temporal store can be reordered with respect to writes or non-temporal stores to different locations. A store fence `sfence` can be used to avoid the reordering of non-temporal stores. A memory fence `mfence` can be used for the same purpose. Additionally, it can be used to stop the reordering of a write followed by a read.

To distinguish events emitted by inline-assembly commands from events emitted by pure RC11-lang commands, we introduce three new access modes:

$$\mathit{Mode} \ni \mathit{md} ::= \ldots \mid \mathtt{nt} \mid \mathtt{sf} \mid \mathtt{tso}$$

Events emitted by plain Ex86 reads, writes, and read-modify-writes carry the mode `tso`: $\mathtt{W}^{\mathtt{tso}}$, $\mathtt{R}^{\mathtt{tso}}$, and $\mathtt{RMW}^{\mathtt{tso}}$. Events emitted by non-temporal stores carry the mode `nt`: $\mathtt{W}^{\mathtt{nt}}$. Events emitted by store fences carry the mode `sf`: $\mathtt{F}^{\mathtt{sf}}$. Events emitted by Ex86 memory fences are indistinguishable from those emitted by `sc` fences, they all carry the mode `sc`. Of course, it would be possible to distinguish events emitted by memory fences by using an extra mode, say `mf`. However, our model assigns the same strength to `sc` fences and to memory fences, so we prefer to simply use the mode `sc`. (In

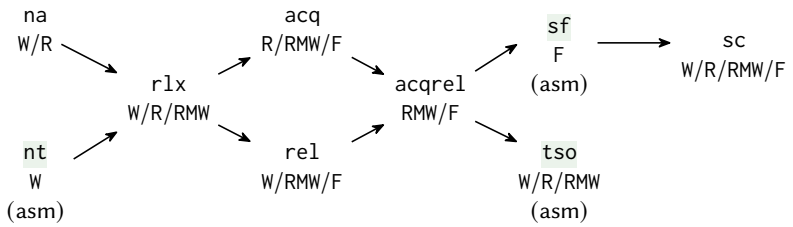


Fig. 5. Diagram of access modes of $\text{RC11}^{\text{Ex86}}$.

other words, in our proposed model, programmers have no good reason to use **asm**{mfence}, as they can equivalently use $\texttt{fence}_{\texttt{sc}}$; we include **asm**{mfence} only for the sake of being complete.)

The following definition introduces $\text{RC11}^{\text{Ex86}}$-consistency. Many of the conditions are identical to those from RC11 (Definition 3.1). Therefore, to avoid repetition, we include only the differences with respect to RC11. For clarity, we highlight these differences using a colored background. Finally, we observe that (in both the new definitions and in those inherited from RC11) the ranges of access modes should be interpreted using the graph from Figure 5; that is, using the order induced by the reflexive-and-transitive closure of the directed-edge relation from Figure 5.

Definition 3.5 ($\text{RC11}^{\text{Ex86}}$-Consistency). An execution graph $(G, \mathsf{rf}, \mathsf{mo})$ is *$\text{RC11}^{\text{Ex86}}$-consistent* if, in addition to the conditions from Definition 3.1 (where Coherence is renamed to Coherence-I), the following conditions hold:

- $\mathit{acyclic}(\mathsf{ppo}_{\text{asm}} \cup \mathsf{eco})$ (Coherence-II)
- $\mathit{irreflexive}([\mathsf{W}^{\mathsf{nt}}]; \mathsf{po}; (\mathsf{rb} \cup \mathsf{mo}))$ (Coherence-III)

where the relations hb, eco, and $\mathsf{ppo}_{\text{asm}}$ are defined as follows:

$$\mathsf{hb} \triangleq (\mathsf{po}_{\text{RC11}} \cup \mathsf{sw})^+ \qquad \mathsf{eco} \triangleq (\mathsf{rf}_e \cup \mathsf{mo} \cup \mathsf{rb})^+$$

$$\mathsf{po}_{\text{RC11}} \triangleq [\mathsf{E} \setminus \mathsf{W}^{\mathsf{nt}}]; \mathsf{po} \cup \mathsf{po}; [\mathsf{RMW}^{\mathsf{tso}} \cup \mathsf{F}^{\sqsupseteq \mathsf{sf}}]$$

$$\begin{aligned}\mathsf{ppo}_{\text{asm}} \triangleq\ & \mathsf{po}; [\mathsf{RMW}^{\mathsf{tso}} \cup \mathsf{F}^{\sqsupseteq \mathsf{sf}}] \\ & \cup [\mathsf{R}^{\mathsf{tso}} \cup \mathsf{RMW}^{\mathsf{tso}} \cup \mathsf{F}^{\mathsf{sc}}]; \mathsf{po} \\ & \cup [\mathsf{F}^{\sqsupseteq \mathsf{sf}}]; \mathsf{po}; [\mathsf{E} \setminus \mathsf{R}] \\ & \cup [\mathsf{W}^{\mathsf{tso}}]; \mathsf{po}; [\mathsf{E} \setminus \mathsf{R} \setminus \mathsf{W}^{\mathsf{nt}}] \\ & \cup [\mathsf{E} \setminus \mathsf{R} \setminus \mathsf{W}^{\mathsf{nt}}]; \mathsf{po}; [\mathsf{W}^{\mathsf{tso}}]\end{aligned}$$

This definition diverges from RC11 in multiple ways:

Diagram of access modes. The diagram of access modes unites RC11 modes and Ex86-inspired modes into the same picture. It is intriguing because it misses some orderings that one would naturally expect, such as $\texttt{tso} \sqsubset \texttt{sc}$ or perhaps even $\texttt{na} \sqsubset \texttt{nt}$. Given that non-temporal stores break release-acquire synchronization, as we shall explain, it is not difficult to understand the absence of the ordering $\texttt{na} \sqsubset \texttt{nt}$. Perhaps more striking is the absence of the ordering $\texttt{tso} \sqsubset \texttt{sc}$. We explain in §3.2.1 that adding such an ordering violates (at least) one of our desiderata.

Definition of hb. Instead of the full po relation, now the definition of hb uses a restricted version of po that excludes edges starting in non-temporal stores, unless they reach a sc fence, a sf fence, or a tso read-modify-write. In §3.2.3, we explain in detail the motivation for this change, but, for now, let us simply say that this relaxation of hb is necessary, for example, to allow the weak behavior of Program MP-NT.

Definition of eco. In RC11, the relation eco can be defined using either the full rf relation or the external restriction rf_e. The two formulations of RC11 are equivalent. In the presence of inline assembly, especially of non-temporal stores, however, the definition of eco must use rf_e: a formulation of $\text{RC11}^{\text{Ex86}}$ where eco is defined using rf is unsound. In §3.2.4, we explain in detail why this is the case.

Consistency Condition - Coherence-II. The consistency conditions now postulate the absence of cycles in $\mathsf{ppo}_{\mathsf{asm}} \cup \mathsf{eco}$. This condition is the key principle that allows one to reason about inline-assembly using our model. In §3.2.2 we shall see that this condition is an adaptation of one of Ex86-consistency conditions. We believe that extensions of RC11 with support for inline assembly for other architectures could be obtained by redefining $\mathsf{ppo}_{\mathsf{asm}}$.

Consistency Condition - Coherence-III. The addition of this condition is a technicality. In RC11, Condition Coherence-I ensures that mo_i and rb_i are included in po. In $\text{RC11}^{\text{Ex86}}$, however, Condition Coherence-I is insufficient to rule out cases that violate these properties, because a po edge that starts with a non-temporal store is not necessarily included in hb. As a consequence, the existence of an event a such that $(a, a) \in [\mathtt{W}^{\mathtt{nt}}]; \mathsf{po}; (\mathsf{rb} \cup \mathsf{mo})$ is not a contradiction to *irreflexive*(hb; eco). This new condition must therefore be included.

3.2.1 Diagram of Access Modes. Let us start by explaining how the mode `sf` fits in Figure 5. It naturally sits between the two strongest modes allowed in a fence: `acqrel` and `sc`. This positioning is natural because an `acqrel` fence is erased by the standard compilation schemes to x86, so they cannot be used to stop the reordering of non-temporal stores. Moreover, a `sc` fence can be used to stop the reordering of a write and a read, for which a store fence is insufficient. This explains the ordering $\mathtt{sf} \sqsubset \mathtt{sc}$.

An interesting implication of the (derived) ordering $\mathtt{rel} \sqsubset \mathtt{sf}$ is that the model allows store fences to establish release-acquire synchronization. In other words, a store fence is allowed in the beginning of a sw edge. It can thus be used to rule out behaviors that contradict the irreflexivity of hb; eco? (Coherence-I). This is exhibited by the following pair of programs:

$$\begin{array}{l}\mathtt{asm}\{[x] :=_{\mathtt{nt}} 1\} \\ \mathtt{fence}_{\mathtt{rel}} \\ [y]^{\mathtt{rlx}} := 1\end{array} \;\Big\|\; \begin{array}{l}a := [y]^{\mathtt{acq}} \;//\; 1 \\ b := [x]^{\mathtt{rel}} \;//\; 0\end{array} \qquad\qquad \begin{array}{l}\mathtt{asm}\{[x] :=_{\mathtt{nt}} 1\} \\ \mathtt{asm}\{\mathtt{sfence}\} \\ [y]^{\mathtt{rlx}} := 1\end{array} \;\Big\|\; \begin{array}{l}a := [y]^{\mathtt{acq}} \;//\; \underline{1} \\ b := [x]^{\mathtt{rel}} \;//\; \underline{0}\end{array}$$

The behavior depicted is allowed by our model in the program on the left, but forbidden in the program on the right. This is in agreement with the behavior exhibited by these programs in Ex86 after compilation, because the `rel` fence would then be erased.

Let us now explain the positioning of `nt` in the diagram. That non-temporal stores are deemed weaker than relaxed writes is easy to understand when we take Program MP-NT into account. Indeed, the weak behavior of MP-NT is disallowed when a `rlx` write is used instead of a non-temporal store:

$$\begin{array}{l}[x]^{\mathtt{rlx}} := 1 \\ [y]^{\mathtt{rel}} := 1\end{array} \;\Big\|\; \begin{array}{l}a := [y]^{\mathtt{acq}} \;//\; \underline{1} \\ b := [x]^{\mathtt{rlx}} \;//\; \underline{0}\end{array}$$

This explains the ordering $\mathtt{nt} \sqsubset \mathtt{rlx}$.

The lack of the ordering $\mathtt{na} \sqsubset \mathtt{nt}$ can be similarly explained:

$$\begin{array}{l}[x]^{\mathtt{na}} := 1 \\ [y]^{\mathtt{rel}} := 1\end{array} \;\Big\|\; \begin{array}{l}a := [y]^{\mathtt{acq}} \;//\; \underline{1} \\ \mathtt{if}\ (a == 1)\ \{ \\ \quad b := [x]^{\mathtt{rlx}} \;//\; \underline{0} \\ \}\end{array} \qquad\qquad \begin{array}{l}\mathtt{asm}\{[x] :=_{\mathtt{nt}} 1\} \\ [y]^{\mathtt{rel}} := 1\end{array} \;\Big\|\; \begin{array}{l}a := [y]^{\mathtt{acq}} \;//\; 1 \\ \mathtt{if}\ (a == 1)\ \{ \\ \quad b := [x]^{\mathtt{rlx}} \;//\; 0 \\ \}\end{array}$$

The `if`-branching is just to prevent a data race between the `na` write and the `rlx` read to x: it makes sure that, when the read is issued, it is preceded by the write with respect to hb. The

program on the left cannot exhibit the depicted behavior because of a cycle in hb; rb, forbidden in both RC11 and RC11$^{\text{Ex86}}$ (since it is an extension of RC11). The program on the right can exhibit the annotated behavior because of the reordering of non-temporal stores with writes to distinct locations.

The lack of the ordering $\texttt{nt} \sqsubset \texttt{na}$ is justified by the *catch-fire* semantics of na. A data race makes every behavior allowed by the model:

$$[x]^{\texttt{na}} := 1 \;\Big\|\; \begin{array}{l} a := [x]^{\texttt{rlx}} \\ b := [y]^{\texttt{rlx}} \;//\; 42 \end{array} \qquad\qquad \texttt{asm}\{[x] :=_{\texttt{nt}} 1\} \;\Big\|\; \begin{array}{l} a := [x]^{\texttt{rlx}} \\ b := [y]^{\texttt{rlx}} \;//\; \underline{b \neq 0} \end{array}$$

This example might instigate the reader to ask the question: why do non-temporal stores, or, more generally, inline-assembly accesses, not follow a catch-fire semantics? There are multiple reasons to avoid this approach. First, assigning catch-fire semantics to racy inline-assembly accesses compromises Property P5, because it allows the behavior of IRIW; and Property P4, because the semantics of a racy program written entirely using inline Ex86-assembly would diverge from the semantics given by Ex86. Additionally, the reasons that justify the catch-fire semantics of na accesses do not apply to inline-assembly accesses. Indeed, there are roughly two reasons why the catch-fire semantics of na accesses is necessary: (1) to validate compiler optimizations (e.g., the reordering of na accesses to different locations), and (2) to support the mapping of na accesses to plain accesses in architectures that do not enforce the acyclicity of $\texttt{po} \cup \texttt{rf}$. In our setting, the compiler is not expected to reorder inline assembly, and our compilation schemes are only to Ex86, which enforces the acyclicity of $\texttt{po} \cup \texttt{rf}$.

Finally, let us explain how tso is placed in the diagram. Because the strengthening to tso accesses is one of our desired properties, tso is placed above every non-sc access. The lack of the ordering $\texttt{tso} \sqsubset \texttt{sc}$ however is intriguing, because sequential consistency is stronger than TSO. The problem is that, in general, RC11 does not enforce SC semantics to programs that mix sc and non-sc accesses to the same location. The following pair of examples (inspired by the Z6.U example from [14]) shows that the semantics assigned to sc accesses by RC11 can be weaker than the semantics assigned to tso accesses by our model:

$$\begin{array}{l} \texttt{asm}\{[x] := 1\} \\ [y]^{\texttt{rel}} := 1 \end{array} \;\Big\|\; \begin{array}{l} \texttt{asm}\{a := [y]\} \;//\; \underline{1} \\ b := [z]^{\texttt{rlx}} \;//\; \underline{0} \end{array} \;\Big\|\; \begin{array}{l} [z]^{\texttt{sc}} := 1 \\ \texttt{fence}_{\texttt{sc}} \\ c := [x]^{\texttt{rlx}} \;//\; \underline{0} \end{array}$$

$$\begin{array}{l} [x]^{\texttt{sc}} := 1 \\ [y]^{\texttt{rel}} := 1 \end{array} \;\Big\|\; \begin{array}{l} a := [y]^{\texttt{sc}} \;//\; 1 \\ b := [z]^{\texttt{rlx}} \;//\; 0 \end{array} \;\Big\|\; \begin{array}{l} [z]^{\texttt{sc}} := 1 \\ \texttt{fence}_{\texttt{sc}} \\ c := [x]^{\texttt{rlx}} \;//\; 0 \end{array}$$

3.2.2 Consistency Condition. - Coherence-II. Condition Coherence-II is the key principle that allows one to reason about programs with inline assembly. Ideally, one would like to reason about such instructions using the hardware model, Ex86, by relying on the guarantee that every cycle containing at least one inline assembly instruction should comply to Ex86-consistency. However, as explained in §2.4, such an approach would be too strong, ruling out behaviors that could be introduced by standard the compiler optimizations. We thus argued that a possible solution would be to enforce the guarantee that every cycle in which every pair of po-separated events contains at least one inline assembly instruction should comply to Ex86-consistency. This is the approach implemented by Coherence-II, with some small caveats.

The formulation of Ex86-consistency, as introduced by Raad et al. [24], includes two consistency conditions: an *internal* condition, which applies to cycles confined within single threads by requiring the irreflexivity of po; $(\texttt{rf}_i \cup \texttt{mo}_i \cup \texttt{rb}_i)$; and an *external* condition, which posits the absence of certain types of cycles spanning over multiple threads.

The irreflexivity of po; ($\mathsf{rf}_i \cup \mathsf{mo}_i \cup \mathsf{rb}_i$) is equivalent to the irreflexivity of both po; rf_i and po; ($\mathsf{mo}_i \cup \mathsf{rb}_i$). [2] Condition No-Thin-Air is stronger than the irreflexivity of po; rf_i, and Conditions Coherence-I and Coherence-III together rule out reflexive edges in po; ($\mathsf{mo}_i \cup \mathsf{rb}_i$).

Therefore, Condition Coherence-II focus on integrating the external condition to the model. To recall the definition of Ex86's external condition, and to make its comparison with Coherence-II clear, we include the definition of Ex86's external condition here, putting it side-by-side with Coherence-II:

(RC11$^{\mathsf{Ex86}}$- Coherence-II)

$$
\begin{aligned}
&\mathit{acyclic}(\mathsf{ppo}_{\mathsf{asm}} \cup \mathsf{eco})\\
&\mathsf{ppo}_{\mathsf{asm}} = \mathsf{po};\ [\mathtt{RMW}^{\mathtt{tso}} \cup \mathtt{F}^{\sqsupseteq \mathtt{sf}}]\\
&\qquad \cup\ [\mathtt{R}^{\mathtt{tso}} \cup \mathtt{RMW}^{\mathtt{tso}} \cup \mathtt{F}^{\mathtt{sc}}];\ \mathsf{po}\\
&\qquad \cup\ [\mathtt{F}^{\sqsupseteq \mathtt{sf}}];\ \mathsf{po};\ [\mathtt{E} \setminus \mathtt{R}]\\
&\qquad \cup\ [\mathtt{W}^{\mathtt{tso}}];\ \mathsf{po};\ [\mathtt{E} \setminus \mathtt{R} \setminus \mathtt{W}^{\mathtt{nt}}]\\
&\qquad \cup\ [\mathtt{E} \setminus \mathtt{R} \setminus \mathtt{W}^{\mathtt{nt}}];\ \mathsf{po};\ [\mathtt{W}^{\mathtt{tso}}]
\end{aligned}
$$

(Ex86 - External)

$$
\begin{aligned}
&\mathit{acyclic}(\mathsf{ppo} \cup \mathsf{rf}_e \cup \mathsf{mo}_e \cup \mathsf{rb}_e)\\
&\mathsf{ppo} = \mathsf{po};\ [\mathtt{RMW} \cup \mathtt{MF} \cup \mathtt{SF}]\\
&\qquad \cup\ [\mathtt{R} \cup \mathtt{RMW} \cup \mathtt{MF}];\ \mathsf{po}\\
&\qquad \cup\ [\mathtt{SF}];\ \mathsf{po};\ [\mathtt{E} \setminus \mathtt{R}]\\
&\qquad \cup\ [\mathtt{W}];\ \mathsf{po};\ [\mathtt{W}]\\
&\qquad \cup\ [\mathtt{W} \cup \mathtt{NT}];\ \mathsf{po}|_{\mathsf{loc}};\ [\mathtt{W} \cup \mathtt{NT}]
\end{aligned}
$$

We keep the notation used by [24] in the statement of External, which diverges from ours in two minor ways: (1) instead of a single set of fences, Ex86 introduces one set exclusively for store fences (SF), and one set exclusively for memory fences (MF); (2) analogously, instead of a single set of write events, there is one exclusive set for non-temporal stores (NT) and one for the remaining writes (W).

The side-by-side comparison reinforces the claim that Coherence-II integrates Ex86-consistency into RC11$^{\mathsf{Ex86}}$ under the condition that pairs of po-separated events in a violating cycle include at least one inline-assembly event. Indeed, most cases of $\mathsf{ppo}_{\mathsf{asm}}$ edges either start or end in a event with mode tso, nt, or sf. There is one exception to this case: edges that either either start or end in a sc fence. This is explained by how we model memory fences. The condition therefore rules out certain kinds of cycles with no inline-assembly instructions, provided that the po-separated events include a sc fence. Such cycles however are already ruled out by Condition SC.

To conclude, let us comment on the difference between the statements of the acyclicity conditions: Coherence-II uses eco, which includes the internal edges mo_i and rb_i; whereas External uses $\mathsf{rf}_e \cup \mathsf{mo}_e \cup \mathsf{rb}_e$, which includes only external edges. The absence of internal edges from the statement of External is compensated by the last case of ppo, $[\mathtt{W} \cup \mathtt{NT}];\ \mathsf{po}|_{\mathsf{loc}};\ [\mathtt{W} \cup \mathtt{NT}]$, which coincides with mo_i; and by the inclusion of [R]; po edges, which include rb_i. We can thus omit the "per-location" case in the definition of $\mathsf{ppo}_{\mathsf{asm}}$, and reuse eco in the statement of Coherence-II. The attentive reader might notice that these edges evade the constraint of one inline-assembly event per pair of po-separated events. They however pose no risk to the soundness of compiler optimizations, because no optimization applies to pairs of a read and a write to the same location, so rb_i edges cannot be undone; and because mo_i edges between plain RC11 accesses in a $\mathsf{ppo}_{\mathsf{asm}} \cup \mathsf{eco}$ cycle can always be merged into an edge of type mo_e, rb_e, or $\mathsf{ppo}_{\mathsf{asm}}$.

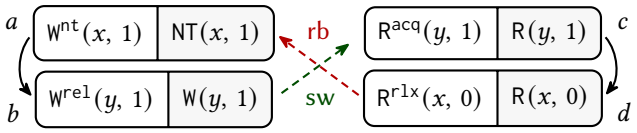
3.2.3 Definition of hb. To see why hb is defined using $\mathsf{po}_{\mathsf{RC11}}$ instead of po, let us consider Program MP-NT. As we shall see, whether the final state σ that maps both x and y to 1 is allowed (i.e. whether $\sigma \in [\![\,\text{MP-NT}\,]\!]$) depends on the definition of hb.

In our model, the final state σ is allowed, thanks to the use of $\mathsf{po}_{\mathsf{RC11}}$ in the definition of hb. If, however, hb was defined as in RC11, that is, $\mathsf{hb}_{\mathsf{RC11}} = (\mathsf{po} \cup \mathsf{sw})^+$, then the state σ would be

[2] The condition $\mathit{irreflexive}(A;\ (B \cup C))$ is equivalent to $\mathit{irreflexive}(A;\ B) \wedge \mathit{irreflexive}(A;\ C)$, for any relations A, B, and C.

disallowed. This is of course problematic, because the behavior is allowed by the Ex86-compiled version of this program.

In §1, we informally justified why this behavior is allowed in Ex86 after compilation in terms of possible reorderings. Having introduced the key consistency condition of Ex86 (Condition External), we can now formally justify why this is the case. We take this opportunity to illustrate our idea of *mixed execution graphs*, a reasoning tool we introduce to conduct proofs of compilation correctness. It allows us to represent graphs from both source and compiled programs simultaneously:



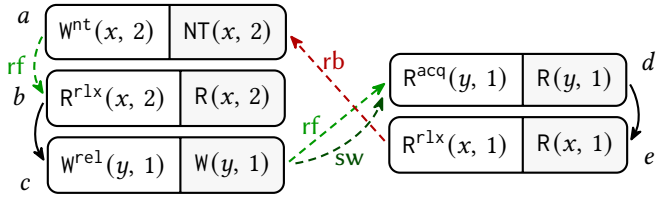
Nodes in this graph carry pairs of a $\text{RC11}^{\text{Ex86}}$ event, issued by the source program, and a Ex86 event, issued by the compiled program. Using this structure, we are able to make several observations:

(1) The behavior is allowed by Ex86 after compilation, because $(a, b) \notin \text{ppo}$, therefore the cycle (a, b, c, d) does not violate External.
(2) The behavior is allowed by $\text{RC11}^{\text{Ex86}}$. Two conditions could potentially be violated by the cycle (a, b, c, d): Coherence-I and Coherence-II. The cycle does not violate Coherence-I, because $(a, b) \notin \text{po}_{\text{RC11}}$; The cycle does not violate Coherence-II, because $(a, b) \notin \text{ppo}_{\text{asm}}$.
(3) The behavior breaks the irreflexivity of hb_{RC11}; eco, because $(a, d) \in \text{hb}_{\text{RC11}}$ and $(d, a) \in \text{rb} \subseteq \text{eco}$. Therefore, a naive extension of RC11 that keeps hb_{RC11} would be unsound.

3.2.4 Definition of eco. To see why rf_e is used in eco, let us consider the following example:

$$\begin{array}{l} \textbf{asm}\{[x] :=_{\text{nt}} 2\} \\ a := [x]^{\text{rlx}} \;//\; 2 \\ [y]^{\text{rel}} := 1 \end{array} \;\Big\|\; \begin{array}{l} b := [y]^{\text{acq}} \;//\; 1 \\ c := [x]^{\text{rlx}} \;//\; 1 \end{array}$$

This program is a slight variation of MP-NT, where we add a read instruction between the non-temporal store and the write to y. Again, we wish to study whether the behavior is allowed by Ex86 after compilation. If that is the case, then the behavior must be allowed by our model. As we shall see, the behavior is indeed exhibited by the compiled program and our model correctly allows it, thanks to the exclusion of rf_i edges from eco. The following mixed execution graph helps to sustain these claims:



This is the only execution graph that corresponds to the annotated behavior, because these rf edges are the only ones that comply with the results of the read operations. Here is the summary of the conclusions we can draw by studying this graph:

(1) The behavior is allowed by Ex86 after compilation, because the graph is Ex86-consistent. Indeed, both the edges (a, b) and (a, c) do not belong to ppo, therefore (a, c, d, e) does not violate External.

(2) The behavior is allowed by $\text{RC11}^{\text{Ex86}}$. Two conditions could potentially be violated by the cycle (a, c, d, e): Coherence-I and Coherence-II. The cycle does not violate Coherence-II, because $(a, c) \notin \text{ppo}_{\text{asm}}{}^{+}$. The cycle does not violate Coherence-I, because (e, a) is the longest eco edge starting from e, and because $(a, b) \notin \text{po}_{\text{RC11}}$, so extending the hb edge (b, e) with eco does not close the cycle.
(3) The behavior breaks the irreflexivity of hb; eco_{RC11}, even when the $\text{RC11}^{\text{Ex86}}$ definition of hb is used. Indeed, both the edges (b, c) and (d, e) belong to po_{RC11}, and $(a, b) \in \text{rf}_i \subseteq \text{eco}_{\text{RC11}}$, so (b, b) forms a reflexive edge in hb; eco_{RC11}. Therefore, a naive extension of RC11 that keeps eco_{RC11} would be unsound.

4 METATHEORY

In this section, we study properties of $\text{RC11}^{\text{Ex86}}$. In particular, we study the correctness of compilation, the correctness of compiler optimizations, and the *data-race-freedom* property: the property that, if a program p has races only on sc accesses, then p can exhibit only sequentially consistent behaviors. Data-race freedom is one of the main design goals of RC11, so it is important to show that $\text{RC11}^{\text{Ex86}}$ preserves this property.

The discussion is organized as follows. In §4.1, we define two compilation schemes to Ex86. In §4.2, before presenting a sketch of our proofs of compilation correctness in §4.3, we introduce the notion of mixed execution graphs, a key concept in these proofs. In §4.4, we discuss our results of compiler-optimization correctness. Finally, in §4.5, we present the formal statement of data-race freedom. The property that $\text{RC11}^{\text{Ex86}}$ is an extension of RC11 and Ex86 is included in the Appendix (Theorems C.14 and C.15).

4.1 Compilation Schemes – Definition and Correctness

Following the traditional approach in the weak-memory literature, we formalize the notion of compilation as a *compilation scheme*. Roughly speaking, a compilation scheme is a program transformation that modifies only memory instructions: the main structure of the program, including control flow and the distribution of threads, is kept, whereas memory instructions from the source language are mapped to zero, one, or multiple instructions from the target language. Therefore, this approach allows us to concentrate on how the transition from the model of the source language to the model of the target language affects the way in which the program interacts with memory. Intuitively, the compilation scheme is correct if the execution of the transformed program can update memory only to a subset of the final states reachable from the execution of the source program.

Definition 4.1 (Compilation Scheme from $\text{RC11}^{\text{Ex86}}$-lang to Ex86-lang).

$$
\begin{aligned}
(\!|\, [e]^{\mathsf{sc}} := e' \,|\!) &\triangleq [e] := e'; \texttt{mfence} & (\!|\, \texttt{fence}_{\mathsf{sc}} \,|\!) &\triangleq \texttt{mfence} \\
(\!|\, [e]^{\neq \mathsf{sc}} := e' \,|\!) &\triangleq [e] := e' & (\!|\, \texttt{fence}_{\neq \mathsf{sc}} \,|\!) &\triangleq \texttt{skip} \\
(\!|\, r := [e]^{md} \,|\!) &\triangleq r := [e] & (\!|\, r := \texttt{rmw}_{md}([e_1], e_2, e_3) \,|\!) &\triangleq r := \texttt{rmw}([e_1], e_2, e_3) \\
(\!|\, s; s' \,|\!) &\triangleq (\!|\, s \,|\!); (\!|\, s' \,|\!) & (\!|\, \texttt{while}\ e\ \{\ s\ \} \,|\!) &\triangleq \texttt{while}\ e\ \{\ (\!|\, s \,|\!)\ \} \\
(\!|\, \texttt{skip} \,|\!) &\triangleq \texttt{skip} & (\!|\, \texttt{if}\ e\ \{\ s\ \} \,|\!) &\triangleq \texttt{if}\ e\ \{\ (\!|\, s \,|\!)\ \} \\
(\!|\, \texttt{asm}\{s\} \,|\!) &\triangleq s
\end{aligned}
$$

Definition 4.2 (Alternative Compilation Scheme). Same as Def. 4.1 except for the following cases:

$$
\begin{aligned}
(\!|\, [e]^{\mathsf{sc}} := e' \,|\!)\text{-alt} &\triangleq \texttt{sfence}; [e] := e'; \texttt{mfence} & (\!|\, [e]^{\mathsf{rlx}} := e' \,|\!)\text{-alt} &\triangleq [e] :=_{\mathsf{nt}} e' \\
(\!|\, [e]^{\mathsf{rel}} := e' \,|\!)\text{-alt} &\triangleq \texttt{sfence}; [e] := e' & (\!|\, \texttt{fence}_{\mathsf{rel},\,\mathsf{acqrel}} \,|\!)\text{-alt} &\triangleq \texttt{sfence}
\end{aligned}
$$

Definition 4.1 follows largely the scheme from Lahav et al. [14]. Perhaps more striking is Definition 4.2, which provides an alternative scheme for Ex86, where relaxed writes can be compiled to

non-temporal stores. The price to pay is the addition of store fences to the compilation of `rel`/`sc` writes and `rel`/`acqrel` fences. The idea is to ensure that, after compilation, every sw edge starts with a store fence. In this way, non-temporal stores, even when emitted from the compilation of `rlx` writes, cannot invalidate release-acquire synchronization.

In a similar way to how we constructed the function $[\![\_]\!]_{\text{RC11}}$, which defines the semantics of RC11 programs, and to how we implicitly constructed $[\![\_]\!]_{\text{RC11}^{\text{Ex86}}}$, we can introduce the function $[\![\_]\!]_{\text{Ex86}}$ defining the semantics of Ex86-lang programs. The definition is included in the Appendix (Definition B.3). The statement of compilation correctness is then straightforward:

Theorem 4.3. *[Correctness of Definitions 4.1 and 4.2] For every $RC11^{Ex86}$-lang program p, the set of final states of $(\!|\, p \,|\!)$ defined by Ex86 is included in the set of final states of p defined by $RC11^{Ex86}$:*

$$\forall p.\ [\![\, (\!|\, p \,|\!) \,]\!]_{Ex86} \subseteq [\![\, p \,]\!]_{RC11^{Ex86}}$$

4.2 Mixed Execution Graphs

Our proofs of compilation correctness rely on the novel notion of *mixed execution graphs*, a type of execution graph whose nodes contain events from both the source-level and target-level models. Before presenting the proof sketch of our compilation correctness results, let us give a brief introduction to mixed execution graphs.

Informally speaking, a mixed execution graph is the superposition of two execution graphs: one called *source graph*, which is associated with a source program p; and one called *target graph*, which is associated with the compilation of p. The key feature of a mixed execution graph is that it captures the fact that source and target graphs share the same overall structure. Indeed, because a compilation scheme preserves the control flow of the source program and changes only how memory operations are mapped to operations in the target language, for every execution graph of the compiled program, one can always construct an execution graph of the source program that preserves much of the structure of the target graph, including its primitive relations po, rf, and mo. The only mismatches between these graphs come from how one memory operation from the source language might be mapped to zero, one, or multiple memory operations from the target language.

To account for these mismatches, nodes in a mixed graph, called *mixed nodes*, carry events from both source and target models. Events from the two models however cannot be arbitrarily assembled in a mixed node: the source-level events in a mixed node correspond to the events of a single source instruction and the target-level events correspond to the events emitted by the snippet of target-level language produced the mapping of this instruction. Therefore, the range of mixed nodes is fixed and determined by the underlying compilation scheme.

Mixed graphs form a very convenient tool for proving compilation-correctness results because they allow one to work with the execution graphs from both the source program and its compiled version at the same time, and because they allow one to forget about the compilation scheme which is ultimately encoded in the set of permissible mixed nodes. Moreover, it is possible to lift the consistency conditions from the models of source and target languages to this mixed-graph structure. Both models can thus be defined on the same structure, thereby allowing one to formally reason about statements of the kind "*one model is stronger than the other*". In fact, the main convenience of mixed execution graphs is precisely to allow one to formulate the compilation correctness result as a statement in this fashion: "*in a mixed execution graph with nodes taken from a well-chosen set, if the consistency conditions of the target model hold, then so do the consistency conditions of the source model*". The set of nodes has to be well chosen so as to correctly reflect the compilation scheme begin considered.

To give an illustration of mixed execution graphs, let us consider the example depicted in Figure 6. We refer the reader to the Appendix (appendix D.2) for a complete exposition of mixed execution

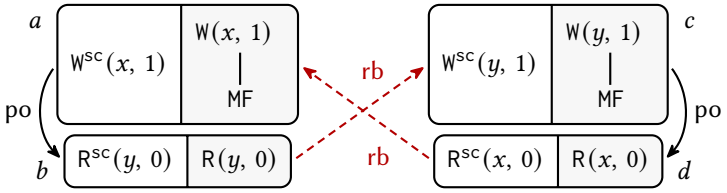
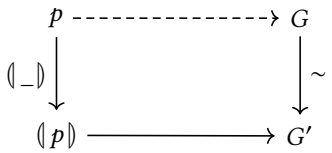


Fig. 6. Example of a mixed execution graph.

graphs and for a more thorough explanation of this example. The nodes are depicted as domino-shaped boxes where the first part contains $\mathsf{RC11}^{\mathsf{Ex86}}$ events and the second part contains Ex86 events. There are two types of nodes in this example: one captures how a sc write is compiled to a plain write followed by a memory fence; the other one captures how a sc read is compiled to a plain read. In this simple example, it is easy to see how a $\mathsf{RC11}^{\mathsf{Ex86}}$ graph G and a a Ex86 graph G' can be recovered from the mixed structure. We wish to argue that the behavior represented by the mixed graph, *store buffering* is disallowed in G because all access modes are sc. More specifically, we wish to argue that G is inconsistent. Consequently, if compilation is correct, then G' should also be inconsistent. Thanks to the mixed graph structure, we can carry out both proofs in the same graph: G is inconsistent because the cycle (a, b, c, d) contradicts SC and G' is inconsistent because the same cycle contradicts External.

4.3 Compilation Correctness - Proof Sketch

The overall structure of our proofs is depicted by the following diagram:



It illustrates the first step of a two-steps strategy to prove that $\llbracket \_ \rrbracket$ is correct. This first step consists of showing that, for every program p, for every execution graph G' associated with $\llbracket p \rrbracket$, there exists a graph G associated with p, such that G *is simulated by* G' (Definition D.8), noted $G \sim G'$, which means that G and G' can be merged into a mixed graph G_m. This first step is accomplished by induction over the construction of the graph G'. Intuitively, because the compiled program $\llbracket p \rrbracket$ preserves much of the structure of p, it is possible to replay the pool-reduction steps from $\llbracket p \rrbracket$ and yield a graph G that satisfies the desired properties. The second step is then to show that, if G_m is Ex86-consistent, then it is $\mathsf{RC11}^{\mathsf{Ex86}}$-consistent, for notions of Ex86-consistency and $\mathsf{RC11}^{\mathsf{Ex86}}$-consistency adapted to mixed graphs (Definitions D.5 and D.6). The consistency of a mixed graph holds iff each of its constituent graphs is consistent, a property we call *Transfer Principle* (Theorem D.7). It follows from this principle that the second step is equivalent to the proof that, if G' is Ex86-consistent, then G is $\mathsf{RC11}^{\mathsf{Ex86}}$-consistent. This is sufficient to conclude the proof.

4.4 Compiler Optimizations

To end this section, we consider the compiler optimizations from [14], and study under which conditions they are sound in $\mathsf{RC11}^{\mathsf{Ex86}}$. As previously stated, our model validates all thread-local optimizations. The only optimization that is only valid under additional conditions is sequentialization, which is a global transformation.

Following Lahav et al. [14], we formalize a compiler optimization as a *program transformation*; that is, a mapping that takes and produces programs in the source language, which, in our case, is the language RC11$^{\text{Ex86}}$-lang. When discussing a given transformation, we use the notation $p \rightsquigarrow p'$ to express that p' can be obtained by applying the transformation to p.

A program transformation is sound, if applying this transformation does not introduce new behaviors. Formally speaking, this means that, if $p \rightsquigarrow p'$ holds, then the set of behaviors of p' is a subset of the set of behaviors of p, that is, $[\![p']\!] \subseteq [\![p]\!]$.

To prove the soundness of a program transformation, we usually resort to its natural generalization to the level of execution graphs: a transformation that applies to events in an execution graph rather than to instructions. In the transformations considered here, this generalization is straightforward. As for programs, we use the notation $G \rightsquigarrow G'$ to express that G' can be obtained by applying the transformation to G. The property that allows us to shift our attention to the graph transformation when proving soundness of a program transformation, is that, if $p \rightsquigarrow p'$ and G' is an execution graph associated with p', then there exists an execution graph G associated with p such that $G \rightsquigarrow G'$. Under this property, to show the soundness of the program transformation, it suffices to show that, if G' is RC11$^{\text{Ex86}}$-consistent, then so is G; and, if G' is racy, then so is G.

4.4.1 Register Promotion. Register promotion replaces accesses to a memory location with accesses to a register, provided that this location is accessed by only one thread and that this location is not accessed via an inline-assembly read-modify-write. At the level of execution graphs, the transformation $G \rightsquigarrow G'$ removes all the accesses to a location x in G, provided that these accesses are related by $G.\text{po}$ and that their intersection with RMW^{tso} is empty. Avoiding RMW^{tso} is necessary, because RMWs act as barriers in x86. Intuitively, this transformation is correct because a consistency-violating cycle in G involving more than one thread must not contain accesses to x (because x is never shared between two threads), so such a cycle would still exist in G'.

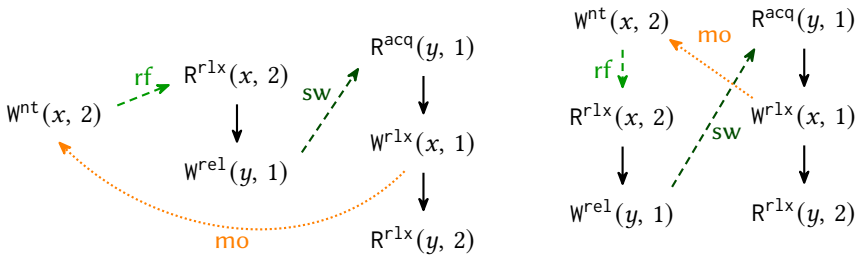
4.4.2 Strengthening. Strengthening replaces an access mode with a stronger one with respect to the ordering of access modes (Figure 5). Definitions in RC11$^{\text{Ex86}}$ are *monotonic*; that is, they restrict access modes using ranges of the kind "$\sqsupseteq$ *md*". [3] The correctness of this transformation is thus trivial, because, every edge of the original graph is preserved.

4.4.3 Deordering and Merging. Deordering transforms sequential composition into parallel composition: $s\,;\, s' \rightsquigarrow s \;||\; s'$. Merging transforms two consecutive instructions into one: $s\,;\, s' \rightsquigarrow s''$. Lahav et al. [14, Table 1 and Figure 11] defines the pairs of deorderable instructions and mergeable instructions permitted in RC11. Both transformations remain valid in RC11$^{\text{Ex86}}$ when restricted to the same deorderable and mergeable pairs of instructions. Intuitively, the correctness argument relies on the remark that these transformations have no effect on ppo_{asm}. Therefore, the additional Coherence-II condition of our extended model does not pose a risk to the correctness of these optimizations, because cycles in $\text{ppo}_{\text{asm}} \cup \text{eco}$ cannot be undone by deordering and merging.

4.4.4 Sequentialization. Sequentialization merges two threads into one by interleaving their instructions. The following example shows that sequentialization is unsound under RC11$^{\text{Ex86}}$:

$$\textbf{asm}\{[x] :=_{\text{nt}} 2\} \;\Big\|\; \begin{array}{l} a := [x]^{\text{rlx}} \;//\; \underline{2} \\ [y]^{\text{rel}} := 1 \end{array} \;\Big\|\; \begin{array}{l} b := [y]^{\text{acq}} \;//\; \underline{1} \\ [x]^{\text{rlx}} := 1 \\ c := [y]^{\text{rlx}} \;//\; \underline{2} \end{array} \quad \rightsquigarrow \quad \textbf{asm}\{[x] :=_{\text{nt}} 2\} \;\Big\|\; \begin{array}{l} b := [y]^{\text{acq}} \;//\; 1 \\ [x]^{\text{rlx}} := 1 \\ c := [y]^{\text{rlx}} \;//\; 2 \end{array} \;\Big\|\; \begin{array}{l} a := [x]^{\text{rlx}} \;//\; 2 \\ [y]^{\text{rel}} := 1 \end{array}$$

[3] Sets of the form S^{md}, for $md \in \{\text{sc}, \text{tso}\}$, can be rewritten as $S^{\sqsupseteq md}$, and sets of the form $S \setminus \text{W}^{\text{nt}}$ can be rewritten as $(S \setminus \text{W}) \cup (S \cap \text{W}^{\sqsupseteq \text{na}})$.



Sequentialization is unsound because, when merging two threads, an external rf edge might become internal. Because internal rf edges are not included in $\mathsf{po}_{\mathsf{RC11}}$, in $\mathsf{ppo}_{\mathsf{asm}}$, or in eco, exchanging a rf_e edge for a rf_i edge might undo cycles in $\mathsf{ppo}_{\mathsf{asm}} \cup \mathsf{eco}$, in hb; $\mathsf{eco}^?$, or in psc.

The omission of rf_i edges from $\mathsf{po}_{\mathsf{RC11}}$, $\mathsf{ppo}_{\mathsf{asm}}$, and eco, is necessary because non-temporal stores break release-acquire synchronization. Moreover, the omission of rf_i in the statement of Coherence-II is inherited from Ex86, which also omits rf_i edges in the statement of External. For this reason, sequentialization is unsound in plain Ex86 [12].

Because sequentialization is unsound in Ex86, its support is incompatible with Property P4. Dropping Property P4 is however undesirable, because its violation leads to one of the two following scenarios: (1) a model that is strictly weaker than Ex86; or (2) a model that is strictly stronger than, or incomparable to, Ex86. The first scenario leads to a lost of reasoning principles, whereas the second scenario invalidates the straightforward identity map as a sound compilation scheme for inline assembly. Therefore, instead of aiming to support sequentialization for the price of abandoning Property P4, we study refinements of sequentialization that are valid under $\mathsf{RC11}^{\mathsf{Ex86}}$ as is.

We call a rf_e edge of a $\mathsf{RC11}^{\mathsf{Ex86}}$-inconsistent graph G *problematic* if sequentialization transforms G into a $\mathsf{RC11}^{\mathsf{Ex86}}$-consistent graph. We note that a problematic edge must contain at least one inline-assembly event. Indeed, because $[\mathsf{E} \setminus \mathsf{W}^{\mathsf{nt}}]; \mathsf{rf}_i \subseteq \mathsf{po}_{\mathsf{RC11}}$, edges between standard RC11 events cannot undo cycles in hb; eco. Moreover, edges between standard RC11 events cannot undo cycles in $\mathsf{ppo}_{\mathsf{asm}} \cup \mathsf{eco}$, because, by the definition of $\mathsf{ppo}_{\mathsf{asm}}$, every edge $(a,\, b) \in G.\mathsf{rf}_e; [\mathsf{R}^{\neq \mathsf{tso}}]$ that is part of a cycle in $G.(\mathsf{ppo}_{\mathsf{asm}} \cup \mathsf{eco})$ must be followed by an edge $(b,\, c) \in \mathsf{po}; [\mathsf{RMW}^{\mathsf{tso}} \cup \mathsf{F}^{\sqsupseteq \mathsf{sf}}]$. Therefore, after sequentialization, it is easy to see that $(a,\, c)$ belongs to the $\mathsf{ppo}_{\mathsf{asm}}$ relation of the transformed graph, and that a cycle in $\mathsf{ppo}_{\mathsf{asm}} \cup \mathsf{eco}$ would still exist.

When we consider two threads, a sufficient purely syntactic condition to rule out the existence of such problematic rf edges in the eventual execution graphs of a program is the following: (1) if one thread includes plain RC11 reads then the addresses of all these accesses and the addresses of all locations modified by the other thread using inline assembly should be statically known and disjoint, and (2) if one thread includes inline-assembly reads then the addresses of all these accesses and the addresses of all locations modified by the other thread (using inline assembly or not) should be statically known and disjoint. We call this condition *No Interaction Through Inline Assembly* (NITIA). Notice that, thanks to the inclusions $[\mathsf{RMW}]; \mathsf{rf}_i \subseteq \mathsf{po}_{\mathsf{RC11}} \cap \mathsf{ppo}_{\mathsf{asm}}$ and $\mathsf{rf}_i; [\mathsf{RMW}] \subseteq \mathsf{po}_{\mathsf{RC11}} \cap \mathsf{ppo}_{\mathsf{asm}}$, read-modify-writes can be ignored when checking the NITIA condition. Refining the statement of sequentialization to require this condition to hold when merging two threads leads to a sound optimization. We prove this claim in the Appendix (Theorem D.16).

Another possible refinement of sequentialization is to add a sc fence between the threads to be merged. Inserting such a fence imposes the constraint that the instructions from one thread are ordered with respect to the instructions from the other thread. In comparison to the previous version of sequentialization, when merging two threads that satisfy NITIA, the resulting sequence of instructions can be any interleaving of the instructions from each of the threads. The proof that

this refinement of sequentialization is correct is included in the Appendix (Theorem D.17).

4.5 Data-Race Freedom

Informally stated, the data-race-freedom property posits that, if a program p has races only on sc accesses, then p can exhibit only sequentially consistent behaviors. This property enforces the reasoning principle that, to recover the relative simplicity of sequential consistency, it suffices to show the absence of races on non-sc accesses.

Because the notion of a race, as introduced in Definition 3.2, applies to execution graphs, not to programs, to formalize this statement, we must define what it means for a program *to have races only on* sc *accesses*, that is, to be *data-race free*:

Definition 4.4 (Data-Race Free). A program p *has races only on* sc *accesses*, or, is *data-race free*, if every SC-consistent execution graph G associated with p has races only on sc accesses:

$$p \text{ is data-race free} \iff \forall G, \mathsf{mo}, \mathsf{rf}, a, b. \begin{pmatrix} \mathit{toPool}(p) \;/\; \mathit{Init} \longrightarrow^* \_ \;/\; G \\ (G, \mathsf{rf}, \mathsf{mo}) \text{ is SC-consistent} \\ (a, b) \text{ forms a data race} \end{pmatrix} \implies a.\mathsf{md} = b.\mathsf{md} = \mathsf{sc}$$

The definition relies on the notion of SC-consistency, captured by a single condition: the acyclicity of $\mathsf{po} \cup \mathsf{rf} \cup \mathsf{mo} \cup \mathsf{rb}$. The restriction to SC-consistent graphs strengthens the reasoning principle enforced by data-race freedom. If, for example, the graphs were assumed to be only $\mathrm{RC11}^{\mathrm{Ex86}}$-consistent, then the resulting property would offer no benefit over reasoning about the program using $\mathrm{RC11}^{\mathrm{Ex86}}$ itself.

Finally, data-race freedom is formally stated as follows:

Theorem 4.5 (Data-Race Freedom). $\forall p.\ p \text{ is data-race free} \implies [\![p]\!]_{\mathrm{RC11}^{\mathrm{Ex86}}} = [\![p]\!]_{\mathrm{SC}}$

A detailed proof of this theorem can be found in the Appendix (§C.1).

5 RELATED WORK

To the extent of our knowledge, we are the first authors to consider the problem of extending C++'s memory model with support for inline assembly. In the following paragraphs, we discuss related work on topics that we covered in this paper.

Models of x86. Sewell et al. [27] introduce an operational model of x86 that, according to the documented tests, agrees with the behavior of actual x86 machines and is proven to be equivalent to the axiomatic formulation of Total Store Order (TSO) [29]. Such a model is devoid of the ambiguity that is often present in the documentation of multiprocessors written in informal prose. An interesting application of the model is to explain the correctness of an optimization that was the subject of a famous discussion in the Linux Kernel mailing list [16]. In this paper, we rely on Raad et al. [24]'s Ex86, an extension of x86 with support for (1) non-temporal stores, (2) store fences, and (3) reads and writes to the full range of Intel's *memory types* (*uncacheable*, *write-combined*, and *write-through*). More specifically, we rely on the axiomatic formulation of Ex86, which formulation is included in the Appendix (Definition B.1).

Models of C++. Batty et al. [4] introduce the first formal memory model of C++ as a formalization of the C++ standard [9] mechanized in Isabelle/HOL [19]. Lahav et al. [14] however identify several issues with this model and introduce RC11 (for *Repaired C11*) in an attempt to repair these flaws. Indeed, Lahav et al. [14] identify at least four problems with the original model of Batty et al. [4]: (1) the proposed compilation schemes [3, 26] to POWER is unsound; (2) the semantics of sc fences is too weak, the authors show that placing sc fences between every memory access is not

sufficient to enforce only sequentially consistent behaviors, and they argue that sc fences are not *cumulative*; (3) *out-of-thin-air* behaviors are allowed even though they cannot be observed in any actual hardware; (4) the model lacks *monotonicity* [30]. The RC11 model fixes these issues with the Axiom SC, which weakens the semantics of programs mixing sc and non-sc accesses so that the compilation schemes to POWER are sound and strengthens the semantics of sc fences; and the Axiom No-Thin-Air, which disallows out-of-thin-air behaviors. The latter axiom has the undesired effect of also disallowing *load buffering* behaviors, which can be observed in actual hardware. Kang et al. [12] present an operational model of C++ that prevents out-of-thin-air behaviors while allowing load buffering. Their model however does not account for sc accesses.

Multi-language semantics. Devising a model for C++ with inline assembly can be framed as a problem of combining the semantics of two different languages: C++ and the assembly language of the underlying hardware architecture. We identify some works that propose general solutions to the problem of specifying *multi-language semantics*. Sammler et al. [25] introduce DimSum, a generic framework to reason about programs written in different languages. Inspired by process calculi, one of the key ideas is to consider the semantics of a program as a labeled transition system where nodes represent the (global) state and transitions are labeled by events. The semantics of a program is written as a refinement statement that accounts for both *demonic non-determinism*, the usual flavor of non-determinism; and *angelic non-determinism* [7], motivated by scenarios where the representation of a value in one language matches the representation of multiple different values in another language. This framework is inadequate for our purposes because, as it stands, it is limited to sequential languages. Moreover, there is also a difference in the nature of our works: whereas Sammler et al. [25] concentrate on a general framework to define the semantics programs written in different languages, with special attention on how the memory representation differs in each of these languages, our focus is rather to underpin the exact (consistency) semantics of programs combining two specific languages, C++ and assembly. Goens et al. [10] study the question of devising memory models for *heterogeneous processors*, processors that mix CPUs and GPUs and allow them to share memory. Their contribution is the introduction of the notion of a *compound memory model*, a way to combine the different memory models from each of devices sharing memory. As the authors put it, "*a compound memory model is not a new memory model*", in the sense that threads from devices abiding by different memory models continue to adhere to these models. This is in contrast with our work, where (1) our extended model constitutes a new model and (2) single threads can mix accesses from two different models, RC11 and Ex86.

Compilation-correctness proofs. Lahav et al. [14] prove the correctness of compilation schemes from RC11 to several architectures (x86, POWER, and Armv7). Podkopaev et al. [20] introduces the idea of an *intermediate memory model* (IMM), a model to which high-level languages, such as C++, can be mapped and from which low-level code can be produced according to compilation schemes proven correct once and for all. The authors argue that IMM is useful for structuring proofs of correctness compilation, because, for example, in a scenario where one has to establish the correctness of compilation schemes of N languages to M architectures, instead of $N \times M$ results, one could instead prove correctness of the mappings from the N languages to IMM (assuming that these mappings exist). These N proofs would still be proofs of compilation correctness (from a given language to IMM); we argue that our idea of mixed execution graphs would be valuable in this compilation-correctness-proof effort. Kokologiannakis et al. [13] develop Kater, a tool that automates reasoning about the metatheory of memory models. The tool can decide the inclusion between two relations in an execution graph and it is possible, even though intricate, to formulate compilation-correctness statements in this fashion. The tool however is unfit

to our purposes because the notion of events comes as a built-in, thereby precluding its use with new types of events such as non-temporal stores and store fences.

6 CONCLUSION

In this paper, we have presented a formal model for C/C++ with inline x86 assembly as an extension of the RC11 formal consistency model for C/C++. One can similarly try to extend RC11 with inline assembly for other hardware platforms, such as Armv8. Doing so is expected to involve a few more challenges, since the Armv8 model makes use of syntactic dependencies between instructions, which do not have an analogue in the C/C++ setting and are not guaranteed to be preserved by compilers. Another possible extension of our work would be to model the persistency semantics of architectures over non-volatile memory. We think that both extensions are worth exploring and leave them for future work.

REFERENCES

[1] Jade Alglave, Richard Grisenthwaite, Artem Khyzha, Luc Maranget, and Nikos Nikoleris. 2024. Puss In Boots: on formalizing Arm's Virtual Memory System Architecture. *IEEE Micro* (July 2024), 1–9. https://doi.org/10.1109/MM.2024.3422668

[2] Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding Cats - Modelling, simulation, testing, and data-mining for weak memory. *ACM Transactions on Programming Languages and Systems* 36, 2 (2014). https://doi.org/10.1145/2627752

[3] Mark Batty, Kayvan Memarian, Scott Owens, Susmit Sarkar, and Peter Sewell. 2012. Clarifying and Compiling C/C++ Concurrency: From C++11 to POWER *(Principles of Programming Languages (POPL))*. 509—-520. https://doi.org/10.1145/2103656.2103717

[4] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ concurrency. In *Principles of Programming Languages (POPL)*. ACM Press, 55–66. https://www.cl.cam.ac.uk/~pes20/cpp/popl085ap-sewell.pdf

[5] Cppreference Community. 2019. Cppreference - Memory Order. https://en.cppreference.com/w/cpp/atomic/memory_order.

[6] The Linux Kernel Community. 2007. Linux Kernel-Based Virtual Machine. https://git.kernel.org/pub/scm/virt/kvm/kvm.git.

[7] Robert W. Floyd. 1967. Nondeterministic Algorithms. *Journal of the ACM* 14, 4 (Oct. 1967), 636—-644. https://doi.org/10.1145/321420.321422

[8] Michael J. Flynn. 1972. Some Computer Organizations and Their Effectiveness. *IEEE Trans. Computers* C-21 (Nov. 1972). https://ieeexplore.ieee.org/document/5009071

[9] International Organization for Standardization (ISO). 2011. *ISO International Standard ISO/IEC 14882:2011(E) – Programming Language C++*. International Organization for Standardization (ISO).

[10] Andrés Goens, Soham Chakraborty, Susmit Sarkar, Sukarn Agarwal, Nicolai Oswald, and Vijay Nagarajan. 2023. Compound Memory Models, Vol. 7. ACM Press, 153:1–153:24. https://doi.org/10.1145/3591267

[11] Intel. 2024. Intel 64 and IA-32 Architectures Software Developer's Manual (Combined Volumes). https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4.html. Order Number: 325462-083US.

[12] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A promising semantics for relaxed-memory concurrency. In *Principles of Programming Languages (POPL)*. 175–189. https://www.cs.tau.ac.il/~orilahav/papers/popl17.pdf

[13] Michalis Kokologiannakis, Ori Lahav, and Viktor Vafeiadis. 2023. Kater: Automating Weak Memory Model Metatheory and Consistency Checking. In *Principles of Programming Languages (POPL)*, Vol. 7. https://doi.org/10.1145/3571212

[14] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing sequential consistency in C/C++11. In *Programming Language Design and Implementation (PLDI)*. ACM Press, 618–632. https://plv.mpi-sws.org/scfix/paper.pdf

[15] Xavier Leroy. 2021. The CompCert C verified compiler. http://compcert.org/man.

[16] Linux Kernel Mailing List. 1999. spin_unlock optimization(i386). https://lists.archive.carbon60.com/linux/kernel/105412.

[17] Roy Margalit and Ori Lahav. 2021. Verifying Observational Robustness Against a C11-Style Memory Model. *Proceedings of the ACM on Programming Languages* 5, POPL (Jan. 2021). https://doi.org/10.1145/3434285

[18] Microsoft Learn. 2021. Advantages of Inline Assembly. https://learn.microsoft.com/en-us/cpp/assembler/inline/advantages-of-inline-assembly.

[19] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. 2002. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. LNCS, Vol. 2283. Springer.

[20] Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. 2019. Bridging the Gap between Programming Languages and Hardware Weak Memory Models. In *Principles of Programming Languages (POPL)*, Vol. 3. ACM Press, 69:1–69:31. https://doi.org/10.1145/3290382

[21] Jeff Preshing. 2012. Memory Ordering at Compile Time. https://preshing.com/20120625/memory-ordering-at-compile-time.

[22] GNU Project. 2005. GNU Compiler Collection. https://gcc.gnu.org/git/gcc.git.

[23] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2017. Simplifying ARM Concurrency: Multicopy-Atomic Axiomatic and Operational Models for ARMv8, Vol. 2. ACM Press, 19:1–19:29. https://doi.org/10.1145/3158107

[24] Azalea Raad, Luc Maranget, and Viktor Vafeiadis. 2022. Extending Intel-X86 Consistency and Persistency: Formalising the Semantics of Intel-X86 Memory Types and Non-Temporal Stores. *Proceedings of the ACM on Programming Languages* 6, POPL (Jan. 2022), 22:1–22:31. https://doi.org/10.1145/3498683

[25] Michael Sammler, Simon Spies, Youngju Song, Emanuele D'Osualdo, Robbert Krebbers, Deepak Garg, and Derek Dreyer. 2023. DimSum: A Decentralized Approach to Multi-Language Semantics and Verification, Vol. 7. 27:1–27:31. https://doi.org/10.1145/3571220

[26] Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams. 2012. Synchronising C/C++ and POWER *(Programming Language Design and Implementation (PLDI))*. 311—-322. https://doi.org/10.1145/2254064.2254102

[27] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. X86-TSO: A Rigorous and Usable Programmer's Model for X86 Multiprocessors. *Commun. ACM* 53, 7 (July 2010), 89—-97. https://doi.org/10.1145/1785414.1785443

[28] Ben Simner, Alasdair Armstrong, Jean Pichon-Pharabod, Christopher Pulte, Richard Grisenthwaite, and Peter Sewell. 2022. Relaxed Virtual Memory in Armv8-A. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 13240)*. Springer, 143–173. https://doi.org/10.1007/978-3-030-99336-8_6

[29] Pradeep S. Sindhu, Jean-Marc Frailong, and Michel Cekleov. 1992. *Formal Specification of Memory Models*. Springer, 25–41. https://doi.org/10.1007/978-1-4615-3604-8_2

[30] V. Vafeiadis, T. Balabonski, S. Chakraborty, R. Morisset, and F. Zappa Nardelli. 2015. Common Compiler Optimisations are Invalid in the C11 Memory Model and What We Can Do About It. In *Principles of Programming Languages (POPL)*. 209–220.

A CONSTRUCTION OF EXECUTION GRAPHS

Pool reduction $\boxed{P \,/\, G \longrightarrow P \,/\, G}$

ReadStep

$$\frac{P[i].\begin{cases}\texttt{reg\_st} &= \phi\\ \texttt{ev\_counter} &= j\\ \texttt{next\_cmd} &= r := [e]^{md}; s\end{cases} \qquad P' = P\left[i := P[i].\begin{cases}\texttt{reg\_st} &:= \phi[r := n]\\ \texttt{ev\_counter} &:= j+1\\ \texttt{next\_cmd} &:= s\end{cases}\right]}{\ell = [\![e]\!]_\phi \quad a = (i,\, j) \quad G' = G.\begin{cases}\texttt{E} &:= G.\texttt{E} \uplus \{a\}\\ \texttt{lab} &:= G.\texttt{lab}[a := \texttt{R}^{md}(\ell,\, n)]\end{cases}}{P \,/\, G \longrightarrow P' \,/\, G'}$$

WriteStep

$$\frac{P[i].\begin{cases}\texttt{reg\_st} &= \phi\\ \texttt{ev\_counter} &= j\\ \texttt{next\_cmd} &= [e]^{md} := e'; s\end{cases} \qquad P' = P\left[i := P[i].\begin{cases}\texttt{ev\_counter} &:= j+1\\ \texttt{next\_cmd} &:= s\end{cases}\right]}{\ell = [\![e]\!]_\phi \quad n = [\![e']\!]_\phi \quad a = (i,\, j) \quad G' = G.\begin{cases}\texttt{E} &:= G.\texttt{E} \uplus \{a\}\\ \texttt{lab} &:= G.\texttt{lab}[a := \texttt{W}^{md}(\ell,\, n)]\end{cases}}{P \,/\, G \longrightarrow P' \,/\, G'}$$

RMWSuccessStep

$$\frac{P[i].\begin{cases}\texttt{reg\_st} &= \phi\\ \texttt{ev\_counter} &= j\\ \texttt{next\_cmd} &= r := \texttt{rmw}_{md}([e_1],\, e_2,\, e_3); s\end{cases} \qquad P' = P\left[i := P[i].\begin{cases}\texttt{reg\_st} &:= \phi[r := n]\\ \texttt{ev\_counter} &:= j+1\\ \texttt{next\_cmd} &:= s\end{cases}\right]}{\ell = [\![e_1]\!]_\phi \quad n = [\![e_2]\!]_\phi \quad m = [\![e_3]\!]_\phi \quad a = (i,\, j) \quad G' = G.\begin{cases}\texttt{E} &:= G.\texttt{E} \uplus a\\ \texttt{lab} &:= G.\texttt{lab}[a := \texttt{RMW}^{md}(\ell,\, n,\, m)]\end{cases}}{P \,/\, G \longrightarrow P' \,/\, G'}$$

RMWFailStep

$$\frac{P[i].\begin{cases}\texttt{reg\_st} &= \phi\\ \texttt{ev\_counter} &= j\\ \texttt{next\_cmd} &= r := \texttt{rmw}_{md}([e_1],\, e_2,\, e_3); s\end{cases} \qquad P' = P\left[i := P[i].\begin{cases}\texttt{reg\_st} &:= \phi[r := m]\\ \texttt{ev\_counter} &:= j+1\\ \texttt{next\_cmd} &:= s\end{cases}\right]}{\ell = [\![e_1]\!]_\phi \quad n = [\![e_2]\!]_\phi \quad a = (i,\, j) \quad m \neq n \quad G' = G.\begin{cases}\texttt{E} &:= G.\texttt{E} \uplus a\\ \texttt{lab} &:= G.\texttt{lab}[a := \texttt{RMW}^{md}(\ell,\, m,\, \bot)]\end{cases}}{P \,/\, G \longrightarrow P' \,/\, G'}$$

FenceStep

$$\frac{P[i].\begin{cases}\texttt{ev\_counter} &= j\\ \texttt{next\_cmd} &= \texttt{fence}_{md}; s\end{cases} \qquad P' = P\left[i := P[i].\begin{cases}\texttt{ev\_counter} &:= j+1\\ \texttt{next\_cmd} &:= s\end{cases}\right]}{a = (i,\, j) \quad G' = G.\begin{cases}\texttt{E} &:= G.\texttt{E} \uplus \{a\}\\ \texttt{lab} &:= G.\texttt{lab}[a := \texttt{F}^{md}]\end{cases}}{P \,/\, G \longrightarrow P' \,/\, G'}$$

Fig. 7. Pool-reduction rules (Part 1 of 2).

$$\textsc{IfStep}\quad \frac{P[i].\begin{cases}\texttt{reg\_st} &= \phi\\ \texttt{next\_cmd} &= \texttt{if}\ e\ \{\ s\ \};\ s'\end{cases} \qquad n = [\![\, e \,]\!]_\phi \qquad P' = P\left[i := P[i].\left\{\texttt{next\_cmd} := \mathit{if}\ n \neq 0\ \mathit{then}\ s;\ s'\ \mathit{else}\ s'\right\}\right]}{P\ /\ G\ \longrightarrow\ P'\ /\ G}$$

$$\textsc{SkipStep}\quad \frac{P[i].\texttt{next\_cmd} = \texttt{skip};\ s \qquad P' = P[i].\texttt{next\_cmd} := s}{P\ /\ G\ \longrightarrow\ P'\ /\ G}$$

$$\textsc{WhileStep}\quad \frac{P[i].\texttt{next\_cmd} = \texttt{while}\ e\ \{\ s\ \};\ s' \qquad P' = P[i].\texttt{next\_cmd} := \texttt{if}\ e\ \{\ s;\ \texttt{while}\ e\ \{\ s\ \}\ \};\ s'}{P\ /\ G\ \longrightarrow\ P'\ /\ G}$$

$$\textsc{SeqStep}\quad \frac{P[i].\texttt{next\_cmd} = (s_1;\ s_2);\ s_3 \qquad P' = P[i].\texttt{next\_cmd} := s_1;\ (s_2;\ s_3)}{P\ /\ G\ \longrightarrow\ P'\ /\ G}$$

$$\textsc{TerminateStep}\quad \frac{P[i].\texttt{next\_cmd} = \texttt{skip} \qquad P' = \lambda j \in \mathit{dom}(P) \setminus \{i\}.\ P[j]}{P\ /\ G\ \longrightarrow\ P'\ /\ G}$$

Fig. 8. Pool-reduction rules (Part 2 of 2).

B MODELS

This section contains the definitions of Ex86 (§B.1), RC11 (§B.2), and RC11$^{\text{Ex86}}$ (§B.3). Each of these models includes the definition of a set of labels, the definition of its consistency conditions, and the syntax of a programming language whose semantics is given by the model. For every such language, the construction of the set of execution graphs associated with a program is essentially the same as the one presented in §A.

B.1 Ex86

$$\begin{aligned}\mathit{Expr} \ni e &::= n\ (\in \mathbb{N}) \mid r\ (\in \mathit{Reg}) \mid \ell\ (\in \mathit{Loc} \triangleq \mathbb{N}) \mid e + e \mid e - e \mid e * e\\ \mathit{Cmd} \ni s &::= r := [e] \mid [e] := e \mid r := \texttt{rmw}\ ([e], e, e) \mid \texttt{mfence}\\ &\quad \mid \texttt{if}\ e\ \{\ s\ \} \mid \texttt{while}\ e\ \{\ s\ \} \mid s;\ s \mid \texttt{skip}\\ &\quad \mid \texttt{sfence} \mid [e] :=_{\texttt{nt}} e\end{aligned}$$

Fig. 9. Syntax of Ex86-lang.

$$\mathit{Lab}_{\text{Ex86}} \ni a ::= \mathtt{W}(\ell,\ n) \mid \mathtt{R}(\ell,\ n) \mid \mathtt{RMW}(\ell,\ n,\ m^?) \mid \mathtt{MF} \mid \mathtt{SF} \mid \mathtt{NT}(\ell,\ n)$$

Fig. 10. Set of labels of Ex86.

Definition B.1 (Ex86-Consistency). An execution graph $(G, \mathsf{rf}, \mathsf{mo}, \mathsf{pf})$ is *Ex86-consistent* if the conditions

- $\mathit{irreflexive}(\mathsf{po};\ (\mathsf{rf}_i \cup \mathsf{mo}_i \cup \mathsf{rb}_i))$ (Internal)
- $\mathit{acyclic}(\mathsf{ob})$ (External)

hold, where the relations ob, rb, pb, and ppo are defined as follows:

$$\begin{aligned} \mathsf{ob} &\triangleq \mathsf{ppo} \cup \mathsf{rf}_e \cup \mathsf{mo}_e \cup \mathsf{rb}_e \\ \mathsf{rb} &\triangleq (\mathsf{rf}^{-1}; \mathsf{mo}) \setminus [\mathtt{E}] \end{aligned}$$

$$\begin{aligned} \mathsf{ppo} \triangleq\ & \mathsf{po}; [\mathtt{RMW} \cup \mathtt{MF} \cup \mathtt{SF}] \\ \cup\ & [\mathtt{R} \cup \mathtt{RMW} \cup \mathtt{MF}]; \mathsf{po} \\ \cup\ & [\mathtt{SF}]; \mathsf{po}; [\mathtt{E} \setminus \mathtt{R}] \\ \cup\ & [\mathtt{W}]; \mathsf{po}; [\mathtt{W}] \\ \cup\ & [\mathtt{W} \cup \mathtt{NT}]; \mathsf{po}|_{\mathsf{loc}}; [\mathtt{W} \cup \mathtt{NT}] \end{aligned}$$

Definition B.2 (Final State). The *final state* of an execution graph $(G, \mathsf{rf}, \mathsf{mo})$ is a partial function $\mathit{finalSt} : \mathit{Loc} \to \mathbb{N}$ that maps a location ℓ to the value of the mo-maximal (write) event on ℓ:

$$\mathit{finalSt}(G, \mathsf{mo})(\ell) = n \iff \exists a.\ \wedge \begin{cases} a \neq I(\_) \\ G.\mathtt{lab}(a) \in \mathtt{W}_\ell \cup \mathtt{RMW\text{-}s}_\ell \\ \mathtt{W}(\ell, n) = \mathit{max}\,(\mathsf{mo}_\ell) \end{cases}$$

Definition B.3 (Ex86-lang Semantics). The semantics of a Ex86-lang program p is defined as the set of final states to which an initial memory (where every location initially stores 0) can be updated:

$$\sigma \in [\![p]\!]_{\mathsf{Ex86}} \iff \exists G, \mathsf{rf}, \mathsf{mo}.\ \wedge \begin{cases} \mathit{toPool}(p)\ /\ \mathit{Init} \longrightarrow^*\ \emptyset\ /\ G \\ (G, \mathsf{rf}, \mathsf{mo}) \text{ is Ex86-consistent} \\ \sigma = \mathit{finalSt}(G) \end{cases}$$

B.2 RC11

We define a simplified version of the RC11 model introduced by Lahav et al. [14]. Our version omits *release sequences* (rs) as defined in RC11 from the *synchronizes-with* relation (sw). This simplification is in agreement with the recent literature [17] and with the current model of the C++ programming language [5].

$$\begin{aligned} \mathit{Expr} \ni e &::= n\ (\in \mathbb{N}) \mid r\ (\in \mathit{Reg}) \mid \ell\ (\in \mathit{Loc} \triangleq \mathbb{N}) \mid e + e \mid e - e \mid e * e \\ \mathit{Cmd} \ni s &::= r := [e]^{md} \mid [e]^{md} := e \mid r := \mathtt{rmw}_{md}\,([e], e, e) \\ &\quad \mid \mathtt{fence}_{md} \mid \mathtt{if}\ e\ \{\ s\ \} \mid \mathtt{while}\ e\ \{\ s\ \} \mid s; s \mid \mathtt{skip} \\ \mathit{Mode} \ni md &::= \mathtt{na} \mid \mathtt{rlx} \mid \mathtt{rel} \mid \mathtt{acq} \mid \mathtt{acqrel} \mid \mathtt{sc} \end{aligned}$$

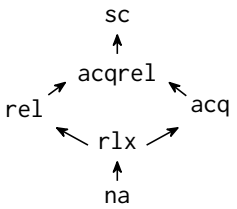


Fig. 11. Syntax of RC11-lang.

$$\mathit{Lab}_{\mathsf{RC11}} \ni a ::= \mathtt{W}^{md}(\ell, n) \mid \mathtt{R}^{md}(\ell, n) \mid \mathtt{F}^{md} \mid \mathtt{RMW}^{md}(\ell, n, m^?)$$

Fig. 12. Set of labels of RC11.

Definition B.4 (RC11-Consistency). An execution graph $(G, \mathsf{rf}, \mathsf{mo})$ is *RC11-consistent* if the conditions

- $\mathit{irreflexive}(\mathsf{hb}; \mathsf{eco}^?)$ (Coherence)
- $\mathit{acyclic}(\mathsf{psc})$ (SC)
- $\mathit{irreflexive}(\mathsf{rb}; \mathsf{mo})$ (Atomicity)
- $\mathit{acyclic}(\mathsf{po} \cup \mathsf{rf})$ (No-Thin-Air)

hold, where the relations hb, eco, rb, and psc are defined as follows:

$$\begin{array}{rcl} \mathsf{rb} & \triangleq & (\mathsf{rf}^{-1}; \mathsf{mo}) \setminus [\mathsf{E}] \\ \mathsf{hb} & \triangleq & (\mathsf{po} \cup \mathsf{sw})^{+} \\ \mathsf{eco} & \triangleq & (\mathsf{rf} \cup \mathsf{mo} \cup \mathsf{rb})^{+} \end{array} \qquad \mathsf{sw} \triangleq \begin{cases} [\mathsf{E}^{\sqsupseteq \mathtt{rel}}]; ([\mathsf{F}]; \mathsf{po})^{?}; [\mathsf{W}^{\sqsupseteq \mathtt{rlx}}]; \\ \quad \mathsf{rf}^{+}; \\ [\mathsf{R}^{\sqsupseteq \mathtt{rlx}}]; (\mathsf{po}; [\mathsf{F}])^{?}; [\mathsf{E}^{\sqsupseteq \mathtt{acq}}] \end{cases}$$

$$\mathsf{psc} \triangleq \mathsf{psc}_{\mathsf{base}} \cup \mathsf{psc}_{\mathsf{fence}} \qquad \begin{array}{rcl} \mathsf{scb} & \triangleq & \mathsf{po} \cup \mathsf{po}|_{\neq \mathsf{loc}}; \mathsf{hb}; \mathsf{po}|_{\neq \mathsf{loc}} \cup \mathsf{hb}|_{\mathsf{loc}} \cup \mathsf{mo} \cup \mathsf{rb} \\ \mathsf{psc}_{\mathsf{base}} & \triangleq & ([\mathsf{E}^{\mathtt{sc}}] \cup [\mathsf{F}^{\mathtt{sc}}]; \mathsf{hb}^{?}); \mathsf{scb}; ([\mathsf{E}^{\mathtt{sc}}] \cup \mathsf{hb}^{?}; [\mathsf{F}^{\mathtt{sc}}]) \\ \mathsf{psc}_{\mathsf{fence}} & \triangleq & [\mathsf{F}^{\mathtt{sc}}]; (\mathsf{hb} \cup \mathsf{hb}; \mathsf{eco}; \mathsf{hb}); [\mathsf{F}^{\mathtt{sc}}] \end{array}$$

The following definition of data race diverges from the one presented in the main text (Definition 3.3) by generalizing hb to an arbitrary relation R. In the main text, this generalization is not necessary, because only the notion of $\mathtt{race}(\mathsf{hb})$ is used. Here, it is necessary to avoid redundancy in §C.1.

Definition B.5 (Data Race). Let G be an execution graph, and R be a relation on the set of events $G.\mathsf{E}$. A pair of events (a, b) form a *data race with respect to* R, or a *R-race* for short, if the following conditions hold:

- $a \neq b$
- $a.\mathtt{loc} = b.\mathtt{loc}$
- $\{a.\mathtt{md}, b.\mathtt{md}\} \cap (\mathtt{W} \cup \mathtt{RMW\text{-}s}) \neq \emptyset$
- $(a, b) \notin R^{+}$ and $(b, a) \notin R^{+}$

The set of R-races of G is noted $G.\mathtt{race}(R)$, or simply $\mathtt{race}(R)$, when G can be easily inferred.

Definition B.6 (RC11-Behaviors).

$$\frac{\mathit{toPool}(p)\ /\ \mathit{Init} \longrightarrow^{*} \emptyset\ /\ G \qquad (G, \mathsf{rf}, \mathsf{mo})\ \mathit{is\ RC11\text{-}consistent}}{p \longrightarrow (G, \mathsf{rf}, \mathsf{mo})}$$

$$\frac{\mathit{toPool}(p)\ /\ \mathit{Init} \longrightarrow^{*} \_\ /\ G \qquad (G, \mathsf{rf}, \mathsf{mo})\ \mathit{is\ RC11\text{-}consistent} \qquad (a, b) \in \mathtt{race}(\mathsf{hb}) \wedge \mathtt{na} \in \{a.\mathtt{md}, b.\mathtt{md}\}}{p \longrightarrow \mathit{UB}}$$

Definition B.7 (RC11-lang Semantics). The semantics of a RC11-lang program p is defined as its set of final states:

$$\sigma \in [\![p]\!]_{\mathrm{RC11}} \iff p \longrightarrow \mathit{UB} \vee \exists G, \mathsf{rf}, \mathsf{mo}.\ p \longrightarrow (G, \mathsf{rf}, \mathsf{mo}) \wedge \sigma = \mathit{finalSt}(G)$$

B.3 RC11$^{\mathrm{Ex86}}$

$$\begin{array}{rcl} \mathit{Cmd} \ni s & ::= & \ldots \mid \mathbf{asm}\{[e] := e\} \mid \mathbf{asm}\{r := [e]\} \mid \mathbf{asm}\{r := \mathtt{rmw}\ ([e], e, e)\} \mid \mathbf{asm}\{\mathtt{mfence}\} \\ & & \mid \mathbf{asm}\{[e] :=_{\mathtt{nt}} e\} \mid \mathbf{asm}\{\mathtt{sfence}\} \\ \mathit{Mode} \ni \mathit{md} & ::= & \ldots \mid \mathtt{nt} \mid \mathtt{sf} \mid \mathtt{tso} \end{array}$$

Fig. 13. Syntax of RC11$^{\mathrm{Ex86}}$-lang.

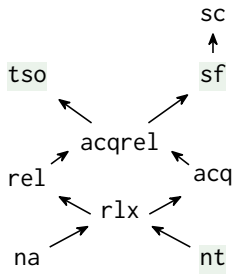


Fig. 14. Diagram of access modes.

Definition B.8 (RC11^{Ex86}-Consistency). An execution graph $(G, \mathsf{rf}, \mathsf{mo})$ is *RC11^{Ex86}-consistent* if the conditions

- $\mathit{irreflexive}(\mathsf{hb}; \mathsf{eco}^?)$ (Coherence-I)
- $\mathit{acyclic}(\mathsf{ppo}_{\mathsf{asm}} \cup \mathsf{eco})$ (Coherence-II)
- $\mathit{irreflexive}([\mathtt{W}^{\mathtt{nt}}]; \mathsf{po}; (\mathsf{rb} \cup \mathsf{mo}))$ (Coherence-III)
- $\mathit{irreflexive}(\mathsf{rb}; \mathsf{mo})$ (Atomicity)
- $\mathit{acyclic}(\mathsf{psc})$ (SC)
- $\mathit{acyclic}(\mathsf{po} \cup \mathsf{rf})$ (No-Thin-Air)

hold, where the relations hb, eco, $\mathsf{ppo}_{\mathsf{asm}}$, rb, and psc are defined as follows:

$$\mathsf{rb} \triangleq (\mathsf{rf}^{-1}; \mathsf{mo}) \setminus [\mathtt{E}] \qquad \mathsf{hb} \triangleq (\mathsf{po}_{\mathsf{RC11}} \cup \mathsf{sw})^+ \qquad \mathsf{eco} \triangleq (\mathsf{rf}_e \cup \mathsf{mo} \cup \mathsf{rb})^+$$

$$\mathsf{sw} \triangleq \begin{cases} [\mathtt{E}^{\sqsupseteq \mathtt{rel}}]; ([\mathtt{F}]; \mathsf{po})^?; [\mathtt{W}^{\sqsupseteq \mathtt{rlx}}]; \\ \quad \mathsf{rf}^+; \\ [\mathtt{R}^{\sqsupseteq \mathtt{rlx}}]; (\mathsf{po}; [\mathtt{F}])^?; [\mathtt{E}^{\sqsupseteq \mathtt{acq}}] \end{cases}$$

$$\mathsf{po}_{\mathsf{RC11}} \triangleq [\mathtt{E} \setminus \mathtt{W}^{\mathtt{nt}}]; \mathsf{po} \cup \mathsf{po}; [\mathtt{RMW}^{\mathtt{tso}} \cup \mathtt{F}^{\sqsupseteq \mathtt{sf}}]$$

$$\begin{aligned} \mathsf{ppo}_{\mathsf{asm}} \triangleq\ & \mathsf{po}; [\mathtt{RMW}^{\mathtt{tso}} \cup \mathtt{F}^{\sqsupseteq \mathtt{sf}}] \\ & \cup [\mathtt{R}^{\mathtt{tso}} \cup \mathtt{RMW}^{\mathtt{tso}} \cup \mathtt{F}^{\mathtt{sc}}]; \mathsf{po} \\ & \cup [\mathtt{F}^{\sqsupseteq \mathtt{sf}}]; \mathsf{po}; [\mathtt{E} \setminus \mathtt{R}] \\ & \cup [\mathtt{W}^{\mathtt{tso}}]; \mathsf{po}; [\mathtt{E} \setminus \mathtt{R} \setminus \mathtt{W}^{\mathtt{nt}}] \\ & \cup [\mathtt{E} \setminus \mathtt{R} \setminus \mathtt{W}^{\mathtt{nt}}]; \mathsf{po}; [\mathtt{W}^{\mathtt{tso}}] \end{aligned}$$

$$\begin{aligned} \mathsf{psc} &\triangleq \mathsf{psc}_{\mathsf{base}} \cup \mathsf{psc}_{\mathsf{fence}} \\ \mathsf{scb} &\triangleq \mathsf{po} \cup \mathsf{po}|_{\neq \mathsf{loc}}; \mathsf{hb}; \mathsf{po}|_{\neq \mathsf{loc}} \cup \mathsf{hb}|_{\mathsf{loc}} \cup \mathsf{mo} \cup \mathsf{rb} \\ \mathsf{psc}_{\mathsf{base}} &\triangleq ([\mathtt{E}^{\mathtt{sc}}] \cup [\mathtt{F}^{\mathtt{sc}}]; \mathsf{hb}^?); \mathsf{scb}; ([\mathtt{E}^{\mathtt{sc}}] \cup \mathsf{hb}^?; [\mathtt{F}^{\mathtt{sc}}]) \\ \mathsf{psc}_{\mathsf{fence}} &\triangleq [\mathtt{F}^{\mathtt{sc}}]; (\mathsf{hb} \cup \mathsf{hb}; \mathsf{eco}; \mathsf{hb}); [\mathtt{F}^{\mathtt{sc}}] \end{aligned}$$

Definition B.9 (RC11^{Ex86}- Behaviors). Analogous to Definition B.6.

Definition B.10 (RC11^{Ex86}-lang Semantics). Analogous to Definition B.7.

C METATHEORY

C.1 Data-Race Freedom

Simplifying Assumption. For simplicity, we ignore RMW accesses in this subsection. An idea to overcome this limitation is to replace our formulation of RC11^{Ex86}-consistency with one that (like the original formulation of RC11 [14]) models a RMW access as either a read or a pair of a write and a read related by a rmw relation.

Definition C.1 (SC-Consistency). An execution graph $(G, \mathsf{rf}, \mathsf{mo})$ is *SC-consistent* if the following condition holds:

- $\mathit{acyclic}(\mathsf{po} \cup \mathsf{rf} \cup \mathsf{mo} \cup \mathsf{rb})$ (SC-Coherence)

Definition C.2 (Racy). Let R be a relation on events. An execution graph G is *R-racy* if it contains a pair of events (a, b) that forms a R-race and for which either a or b is not $\mathtt{sc}$:

$$G \text{ is } R\text{-racy} \iff \exists\, a, b.\ (a, b) \in \mathtt{race}(R) \wedge (a.\mathtt{md} \neq \mathtt{sc} \vee b.\mathtt{md} \neq \mathtt{sc})$$

Definition C.3 (Data-Race Free). Let R be a relation on events. A program p is *data-race free with respect to R*, or simply *is R-race free*, if every SC-consistent execution graph G associated with p is not R-racy:

$$p \text{ is } R\text{-race free} \iff \forall\, G, \mathsf{mo}, \mathsf{rf}, a, b. \begin{pmatrix} \mathit{toPool}(p) \ /\ \mathit{Init} \longrightarrow^* \_ \ /\ G \\ (G, \mathsf{rf}, \mathsf{mo}) \text{ is SC-consistent} \\ (a, b) \in \mathtt{race}(R) \end{pmatrix} \implies a.\mathtt{md} = b.\mathtt{md} = \mathtt{sc}$$

We show that our extended model $\mathsf{RC11}^{\mathsf{Ex86}}$ enjoys the data-race-freedom property; that is, the semantics assigned by $\mathsf{RC11}^{\mathsf{Ex86}}$ to a hb-race free program coincides with SC:

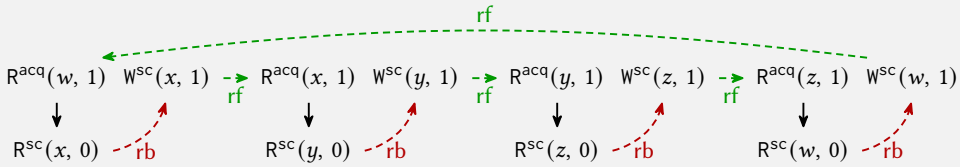
Theorem C.4 ($\mathsf{RC11}^{\mathsf{Ex86}}$- Data-Race Freedom).

$$\forall\, p.\ p \text{ is hb-race free} \implies [\![p]\!]_{RC11^{Ex86}} = [\![p]\!]_{SC}$$

Proof. From Corollary C.11, it follows that p is $(\mathsf{po} \cup \mathsf{rf}|_{\mathsf{sc}})$-race free. Let G be a $\mathsf{RC11}^{\mathsf{Ex86}}$-consistent graph associated with p. By Lemma C.13, the graph is not $(\mathsf{po} \cup \mathsf{rf}|_{\mathsf{sc}})$-racy. Then, by Lemma C.12, we conclude that G is SC-consistent. □

We now state and prove the lemmas on which the proof of Theorem C.4 relies.

Remark C.5. A $\mathsf{RC11}^{\mathsf{Ex86}}$-consistent (or RC11-consistent) that is not hb-racy is not necessarily SC-consistent:



Notation. The relation $R|_A$ denotes the restriction of R to a set A: $R|_A \triangleq [A]; R; [A]$. We use the abbreviation $R|_{\mathsf{sc}}$ for the restriction of R to E^{sc}: $R|_{\mathsf{sc}} \triangleq R|_{\mathsf{E}^{\mathsf{sc}}}$. The graph $G|_A$ is the restriction of G to A: $G|_A \triangleq G.\{\mathsf{E} := (G.\mathsf{E} \cap A)\}$.

Definition C.6 (R-closed). A set A is *closed with respect to a relation R*, or simply *R-closed*, if the inclusion $R; [A] \subseteq [A]; R$ holds.

Lemma C.7. *Let $(G, \mathsf{rf}, \mathsf{mo})$ be a $RC11^{Ex86}$-consistent (resp. SC-consistent) graph associated with a program p, and let $D \subseteq G.\mathsf{E}$ be a $(\mathsf{po} \cup \mathsf{rf})$-closed set. The graph $(G|_D, \mathsf{rf}|_D, \mathsf{mo}|_D)$ is $RC11^{Ex86}$-consistent (resp. SC-consistent). Moreover, the graph $G|_D$ is associated with p, and $G.\mathsf{hb}|_D$ (the restriction of $G.\mathsf{hb}$ to D) is included in $G|_D.\mathsf{hb}$ (the happens-before relation derived from $\mathsf{po}|_D$ and $\mathsf{rf}|_D$).*

Proof. Because D is $(\mathsf{po} \cup \mathsf{rf})$-closed, the following inclusion holds:

$$\begin{array}{rcl} G.\mathsf{sw}|_D & \subseteq & [D]; [\mathtt{E}^{\sqsupseteq \mathtt{rel}}]; ([\mathtt{F}]; \mathsf{po})^?; [\mathtt{W}^{\sqsupseteq \mathtt{rlx}}]; \mathsf{rf}^+; [\mathtt{R}^{\sqsupseteq \mathtt{rlx}}]; (\mathsf{po}; [\mathtt{F}])^?; [\mathtt{E}^{\sqsupseteq \mathtt{acq}}]; [D] \\ & \subseteq & [\mathtt{E}^{\sqsupseteq \mathtt{rel}}]; ([\mathtt{F}]; \mathsf{po}|_D)^?; [\mathtt{W}^{\sqsupseteq \mathtt{rlx}}]; \mathsf{rf}|_D^+; [\mathtt{R}^{\sqsupseteq \mathtt{rlx}}]; (\mathsf{po}|_D; [\mathtt{F}])^?; [\mathtt{E}^{\sqsupseteq \mathtt{acq}}] \\ & \subseteq & G|_D.\mathsf{sw} \end{array}$$

It is then easy to see that $G.\mathsf{hb}|_D \subseteq G|_D.\mathsf{hb}$. □

The restriction of rf and mo to D in the statement of Lemma C.7 is not necessary. However, these restrictions result in a stronger statement, because $\mathsf{RC11}^{\mathsf{Ex86}}$-consistency (resp. SC-consistency) is monotonic with respect to both rf and mo.

Lemma C.8. *The set $dom(R^*; [A])$ is R-closed, for any set A and relation R.*

Proof. Let D stand for $dom(R^*; [A])$, and let $(a, b) \in R; [D]$. It suffices to show that $a \in D$. There exists c such that and $(b, c) \in R^*; [A]$. Therefore $(a, c) \in R; R^*; [A] \subseteq R^*; [A]$, which implies that $a \in D$. □

Corollary C.9. *Let G be a graph, and D be a subset of $G.\mathtt{E}$. The set $dom((\mathsf{po} \cup \mathsf{rf})^*; [D])$ is $(\mathsf{po} \cup \mathsf{rf})$-closed.*

Lemma C.10. *Let p be a hb-race free program and let $(G, \mathsf{rf}, \mathsf{mo})$ be a SC-consistent graph associated with p. The relation rf is included in $(\mathsf{po} \cup \mathsf{rf}|_{\mathsf{sc}})^+$.*

Proof. We introduce the following notation:

$$\begin{array}{rcl} R & \triangleq & (\mathsf{po} \cup \mathsf{rf}|_{\mathsf{sc}})^+ \\ S & \triangleq & \mathsf{rf} \setminus R \end{array}$$

Suppose by contradiction that S is non-empty. Let (a, b) be a pair in S for which b is minimal with respect to $\mathsf{po} \cup \mathsf{rf}$. (Such a pair exists because $\mathsf{po} \cup \mathsf{rf}$ is acyclic in G.)

Claim 1. The pair (a, b) belongs to rf_e.

Proof. If $(a, b) \in \mathsf{rf}_i$, then $(a, b) \in \mathsf{po} \subseteq R$. □

Let D be the set $dom((\mathsf{po} \cup \mathsf{rf})^*; [\{a, b\}])$.

Claim 2. If c belongs to D, then $c \in dom((\mathsf{po} \cup \mathsf{rf}|_{\mathsf{sc}})^*; [\{a, b\}])$.

Proof. Suppose by contradiction that there exists $c \neq a, b$ such that $c \in dom((\mathsf{po} \cup \mathsf{rf})^*; [\{a, b\}])$ but $c \notin dom((\mathsf{po} \cup \mathsf{rf}|_{\mathsf{sc}})^*; [\{a, b\}])$. Then it must be the case that

$$c \in dom((\mathsf{po} \cup \mathsf{rf})^*; S; (\mathsf{po} \cup \mathsf{rf})^*; [\{a, b\}]).$$

However, this implies the existence of a pair in $S \setminus \{(a, b)\}$ that contradicts the minimality of b with respect to $\mathsf{po} \cup \mathsf{rf}$. □

Claim 3. The graph $(G|_D, \mathsf{rf}|_D, \mathsf{mo}|_D)$ is not $(G|_D.\mathsf{hb})$-racy, SC-consistent, and associated with p.

Proof. That the graph is SC-consistent and associated with p is a direct consequence from Lemma C.7. Let $c, d \in D$ be a pair of distinct events forming a $G|_D.\mathsf{hb}$-race. Suppose by contradiction that $min\,(a.\mathtt{md}, b.\mathtt{md}) \neq \mathtt{sc}$. Because G is not $G.\mathsf{hb}$-racy, it follows that $(c, d) \in G.\mathsf{hb} \cup G.\mathsf{hb}^{-1}$. It follows from Lemma C.7 that $(c, d) \in G|_D.\mathsf{hb} \cup G|_D.\mathsf{hb}^{-1}$. □

Claim 4. The events a and b are $\mathsf{po}|_D$-maximal.

Proof. Suppose by contradiction that there exists $c \in D$ such that $(a, c) \in \mathsf{po}$. Because $c \in D$, either $(c, a) \in (\mathsf{po} \cup \mathsf{rf})^*$ or $(c, b) \in (\mathsf{po} \cup \mathsf{rf})^*$. If $(c, a) \in (\mathsf{po} \cup \mathsf{rf})^*$, then (a, c) violates the acyclicity of $\mathsf{po} \cup \mathsf{rf}$. If $(c, b) \in (\mathsf{po} \cup \mathsf{rf})^*$, then, from Claim 2, it follows that $(c, b) \in (\mathsf{po} \cup \mathsf{rf}|_{\mathsf{sc}})^*$. But then $(a, b) \in \mathsf{po}$; $(\mathsf{po} \cup \mathsf{rf}|_{\mathsf{sc}})^* \subseteq R$, a contradiction to $(a, b) \in S$.

Suppose by contradiction that there exists $c \in D$ such that $(b, c) \in \mathsf{po}$. If $(c, a) \in (\mathsf{po} \cup \mathsf{rf})^*$, then (a, b, c) violates the acyclicity of $\mathsf{po} \cup \mathsf{rf}$. Analogously, if $(c, b) \in (\mathsf{po} \cup \mathsf{rf})^*$, then (b, c) violates the acyclicity of $\mathsf{po} \cup \mathsf{rf}$. □

Let $e \in D$ be the write event that immediately precedes a in mo, that is, $(e, a) \in \mathsf{mo}|_{\mathsf{imm}}$. We define the graph G', and the relations rf' and rb' as follows:

$$\begin{aligned} G' &\triangleq G|_D.\{\mathtt{lab} := G|_D.\mathtt{lab}[b \mapsto \mathsf{R}^{md_2}(x, v)]\} \text{ where } \begin{cases} G|_D.\mathtt{lab}(e) = \mathsf{W}^{md_1}(x, v) \\ G|_D.\mathtt{lab}(b) = \mathsf{R}^{md_2}(x, \_) \end{cases} \\ \mathsf{rf}' &\triangleq (\mathsf{rf}|_D \setminus \{(a, b)\}) \cup \{(e, b)\} \end{aligned}$$

Claim 5. The graph $(G', \mathsf{rf}', \mathsf{mo}|_D)$ is SC-consistent, and associated with p.

Proof. Suppose, by contradiction, that there is a cycle C in $\mathsf{po}|_D \cup \mathsf{rf}' \cup \mathsf{mo}|_D \cup \mathsf{rb}'$. Let rb' denote the derived reads-before relation $G'.\mathsf{rb}$. Because $\mathsf{rb}' \setminus \mathsf{rb}|_D = \{(b, a)\}$ and $\mathsf{rf}' \setminus \mathsf{rf}|_D = \{(e, b)\}$, the only edges that could possibly be in C but not in $G|_D$ are (b, a) and (e, b). If C includes the rf' edge (e, b), then it must be followed by a rb' edge, because b is a $\mathsf{po}|_D$-maximal read event. By definition of the reads-before relation, it follows that $\mathsf{rf}'; \mathsf{rb}'$ is included in $G'.\mathsf{mo}$, which coincides with $\mathsf{mo}|_D$. The sequence $[e]; \mathsf{rf}'; [b]; \mathsf{rb}'$ can thus be exchanged with $[e]; \mathsf{mo}|_D$. Moreover, if C includes the rb' edge (b, a), then it must be followed by a $\mathsf{mo}|_D$ edge because a is a $\mathsf{po}|_D$-maximal write event. The sequence $[b]; \mathsf{rb}'; [a]; \mathsf{mo}|_D$ can thus be exchanged with $[b]; \mathsf{rb}|_D$. Performing all these exchanges yields a cycle in $\mathsf{po}|_D \cup \mathsf{rf}|_D \cup \mathsf{mo}|_D \cup \mathsf{rb}|_D$, a contradiction to the SC-consistency of $G|_D$.

□

Claim 6. The graph $(G', \mathsf{rf}', \mathsf{mo}|_D)$ is $(G'.\mathsf{hb})$-racy.

Proof. The events a and b are distinct, they act on the same location, at least one of them is a write event. Moreover, $min\,(a.\mathtt{md}, b.\mathtt{md}) \neq \mathtt{sc}$, because otherwise $(a, b) \in R$. Finally, because both are $\mathsf{po}|_D$-maximal, and because $a \notin dom(\mathsf{rf}')$, it follows that $[\{a, b\}]; G'.\mathsf{hb} \subseteq [\{a, b\}]; (\mathsf{po}|_D \cup \mathsf{rf}')^+ = \emptyset$. Therefore, the pair (a, b) forms a $(G'.\mathsf{hb})$-race. □

From Claims 5 and 6, it follows that the graph $(G', \mathsf{rf}', \mathsf{mo}|_D)$ contradicts the assumption that p is hb-race free. □

Corollary C.11. *If a program p is hb-race free, then it is $(\mathsf{po} \cup \mathsf{rf}|_{\mathsf{sc}})$-race free.*

Lemma C.12. *Let G be a $RC11^{Ex86}$-consistent graph. If G is not $(\mathsf{po} \cup \mathsf{rf}|_{\mathsf{sc}})$-racy, then G is is SC-consistent.*

Proof. Because G is not $(\mathsf{po} \cup \mathsf{rf}|_{\mathsf{sc}})$-racy, the following inclusions hold:

$$\begin{aligned} \mathsf{rf} &\subseteq \mathsf{rf}|_{\mathsf{sc}} \cup (\mathsf{po} \cup \mathsf{rf}|_{\mathsf{sc}})^+ \\ \mathsf{mo} &\subseteq \mathsf{mo}|_{\mathsf{sc}} \cup (\mathsf{po} \cup \mathsf{rf}|_{\mathsf{sc}})^+ \\ \mathsf{rb} &\subseteq \mathsf{rb}|_{\mathsf{sc}} \cup (\mathsf{po} \cup \mathsf{rf}|_{\mathsf{sc}})^+ \end{aligned}$$

We can use these inclusions to show that the violation of SC-consistency yields the following chain of implications:

$$\begin{aligned}&\mathit{cyclic}(\mathsf{po} \cup \mathsf{rf} \cup \mathsf{mo} \cup \mathsf{rb})\\ &\implies \mathit{cyclic}(\mathsf{po} \cup \mathsf{rf}|_{\mathsf{sc}} \cup \mathsf{mo}|_{\mathsf{sc}} \cup \mathsf{rb}|_{\mathsf{sc}} \cup (\mathsf{po} \cup \mathsf{rf}|_{\mathsf{sc}})^+)\\ &\implies \mathit{cyclic}(\mathsf{po} \cup \mathsf{rf}_e|_{\mathsf{sc}} \cup \mathsf{mo}_e|_{\mathsf{sc}} \cup \mathsf{rb}_e|_{\mathsf{sc}})\end{aligned}$$

The internal edges of a cycle in $\mathsf{po} \cup \mathsf{rf}_e|_{\mathsf{sc}} \cup \mathsf{mo}_e|_{\mathsf{sc}} \cup \mathsf{rb}_e|_{\mathsf{sc}}$ are separated by external edges whose domain and codomain is included in E^{sc}. Consequently, the domain and codomain of the internal edges po must also be included in E^{sc}. It follows that the relation $\mathsf{po}|_{\mathsf{sc}} \cup \mathsf{rf}_e|_{\mathsf{sc}} \cup \mathsf{mo}_e|_{\mathsf{sc}} \cup \mathsf{rb}_e|_{\mathsf{sc}}$ is cyclic, a contradiction to Condition SC. □

Lemma C.13. *If a program p is $(\mathsf{po} \cup \mathsf{rf}|_{\mathsf{sc}})$-race free, then every RC11^{Ex86}-consistent graph associated with p is not $(\mathsf{po} \cup \mathsf{rf}|_{\mathsf{sc}})$-racy.*

Proof. Let $(G, \mathsf{rf}, \mathsf{mo})$ be a RC11$^{\mathsf{Ex86}}$-consistent graph associated with p. Suppose by contradiction that G is $(\mathsf{po} \cup \mathsf{rf}|_{\mathsf{sc}})$-racy. In other words, suppose that the set $S \triangleq \mathtt{race}(\mathsf{po} \cup \mathsf{rf}|_{\mathsf{sc}}) \setminus (\mathsf{E}^{\mathsf{sc}} \times \mathsf{E}^{\mathsf{sc}})$ is non-empty. Let (a, b) be a pair in S that is minimal with respect to $\mathsf{po} \cup \mathsf{rf}$, that is:

$$\forall c, d.\ (c, d) \in S \implies c, d \in \mathit{dom}((\mathsf{po} \cup \mathsf{rf})^*; [\{a, b\}]) \implies (c, d) \in \{(a, b), (b, a)\}.$$

Let D be the set $\mathit{dom}((\mathsf{po} \cup \mathsf{rf})^*; [\{a, b\}])$.

Claim 1. The events a and b belong to different threads.

Proof. If either $(a, b) \in \mathsf{po}$ or $(b, a) \in \mathsf{po}$, then (a, b) would not form a $(\mathsf{po} \cup \mathsf{rf}|_{\mathsf{sc}})$-race. □

Claim 2. $(\mathsf{po} \cup \mathsf{rf})^*; [\mathsf{E} \setminus \{a, b\}]; (\mathsf{po} \cup \mathsf{rf}); [\{a, b\}] \subseteq (\mathsf{po} \cup \mathsf{rf}|_{\mathsf{sc}})^+$

Proof. Suppose by contradiction that

$$(\mathsf{po} \cup \mathsf{rf})^*; [\mathsf{E} \setminus \{a, b\}]; (\mathsf{po} \cup \mathsf{rf}); [\{a, b\}] \setminus (\mathsf{po} \cup \mathsf{rf}|_{\mathsf{sc}})^+ \neq \emptyset.$$

Then, there must be an edge $(c, d) \in (\mathsf{po} \cup \mathsf{rf})^*$, such that

$$d \in \mathit{dom}([\mathsf{E} \setminus \{a, b\}]; (\mathsf{po} \cup \mathsf{rf}); [\{a, b\}]),$$

and $(c, d) \notin (\mathsf{po} \cup \mathsf{rf}|_{\mathsf{sc}})^+$.

We claim that $d \in \mathit{dom}((\mathsf{po} \cup \mathsf{rf}|_{\mathsf{sc}}); [\{a, b\}])$. If $d \in \mathit{dom}((\mathsf{po} \cup \mathsf{rf}_i); [\{a, b\}])$, then the assertion follows immediately. Moreover, if $d \in \mathit{dom}(\mathsf{rf}_e; [\{a, b\}])$, and if $d \notin \mathit{dom}((\mathsf{po} \cup \mathsf{rf}|_{\mathsf{sc}}); [\{a, b\}])$, then it must be the case that $d \notin \mathit{dom}((\mathsf{rf}_e \setminus \mathsf{rf}|_{\mathsf{sc}}); [\{a, b\}])$. However, in this case, either the pair (d, a) or the pair (d, b) would contradict the minimality of (a, b).

We now proceed by induction on the number of $(\mathsf{po} \cup \mathsf{rf})$ steps between c and d. In the base case, when $c = d$, the desired conclusion follows from the previous paragraph. In the inductive case, we have e such that $(c, e) \in (\mathsf{po} \cup \mathsf{rf})$ and $(e, d) \in (\mathsf{po} \cup \mathsf{rf}|_{\mathsf{sc}})^+$. We wish to prove that $(c, e) \in (\mathsf{po} \cup \mathsf{rf}|_{\mathsf{sc}})^+$. If not, then the pair (c, e) forms a $(\mathsf{po} \cup \mathsf{rf}|_{\mathsf{sc}})$-race. The minimality of (a, b) implies that $(c, e) \in \{(a, b), (b, a)\}$. However, because $e \in \mathit{dom}((\mathsf{po} \cup \mathsf{rf}|_{\mathsf{sc}})^+; [\{a, b\}])$, this contradicts either No-Thin-Air or the fact that $(a, b) \in S$. □

Claim 3. $[\{a, b\}]; (\mathsf{po} \cup \mathsf{rf})^+; (\mathsf{po} \cup \mathsf{rf}); [\{a, b\}] = \emptyset$.

Proof. The relation $R \triangleq [\{a, b\}]; (\mathsf{po} \cup \mathsf{rf})^+; (\mathsf{po} \cup \mathsf{rf}); [\{a, b\}]$ is included in

$$\{(a, a), (a, b), (b, a), (b, b)\}.$$

The inclusions $(a, a) \in R$ and $(b, b) \in R$ contradict No-Thin-Air. Therefore, if R is non-empty, then it must be the case that either $(a, b) \in R$ or $(b, a) \in R$. Suppose by contradiction, and without loss of generality, that $(a, b) \in R$. Then, there exists c such that $(a, c) \in (\mathsf{po} \cup \mathsf{rf})^+$ and $(c, b) \in$

$(\mathsf{po} \cup \mathsf{rf})$. If $c \in \{a, b\}$, then either (a, c) or (c, b) contradicts No-Thin-Air. If $c \notin \{a, b\}$, then, by Claim 2, it follows that $(a, b) \in (\mathsf{po} \cup \mathsf{rf}|_{\mathsf{sc}})^+$, a contradiction to the fact that $(a, b) \in S$. □

Claim 4. The events a and b are $\mathsf{po}|_D$-maximal.
If $(a, b) \in \mathsf{rf}$ (resp. $(b, a) \in \mathsf{rf}$), then b (resp. a) is $(\mathsf{po}|_D \cup \mathsf{rf}|_D)$-maximal.
If $(a, b) \notin \mathsf{rf} \cup \mathsf{rf}^{-1}$, then a and b are $(\mathsf{po}|_D \cup \mathsf{rf}|_D)$-maximal.

Proof. Suppose by contradiction, and without loss of generality, that a is not $\mathsf{po}|_D$-maximal, and let c be an event in D such that $(a, c) \in \mathsf{po}$. Then it must be the case that either $(c, a) \in (\mathsf{po} \cup \mathsf{rf})^*$ or $(c, b) \in (\mathsf{po} \cup \mathsf{rf})^*$. The first case contradicts No-Thin-Air. If $(c, b) \in (\mathsf{po} \cup \mathsf{rf})^*$, then $(c, b) \in (\mathsf{po} \cup \mathsf{rf})^+$, otherwise $c = b$ and $(a, b) \in \mathsf{po}$, a contradiction to Claim 1. The inclusion $(c, b) \in (\mathsf{po} \cup \mathsf{rf})^+$, however, contradicts Claim 3, because then $(a, b) \in [a]; \mathsf{po}; [c]; (\mathsf{po} \cup \mathsf{rf})^+; [b] \subseteq [\{a, b\}]; (\mathsf{po} \cup \mathsf{rf})^+; (\mathsf{po} \cup \mathsf{rf}); [\{a, b\}]$.

If $(a, b) \notin \mathsf{rf} \cup \mathsf{rf}^{-1}$, then we prove, without loss of generality, that a is $(\mathsf{po}|_D \cup \mathsf{rf}|_D)$-maximal. It suffices to show that $[a]; (\mathsf{po}|_D \cup \mathsf{rf}|_D)^+$ is empty. Suppose by contradiction that there exists $c \in D$ such that $(a, c) \in (\mathsf{po}|_D \cup \mathsf{rf}|_D)$ and $(c, \_) \in (\mathsf{po}|_D \cup \mathsf{rf}|_D)^*$. Because a is $\mathsf{po}|_D$-maximal, the edge (a, c) belongs to $\mathsf{rf}|_D$. Moreover, because $c \in D$ it is the case that either $(c, a) \in (\mathsf{po} \cup \mathsf{rf})^*$ or $(c, b) \in (\mathsf{po} \cup \mathsf{rf})^*$. The first case contradicts No-Thin-Air. In the second case, the edge (c, b) must also belong to $(\mathsf{po} \cup \mathsf{rf})^+$; otherwise, the events c and b would coincide, and therefore $(a, b) \in \mathsf{rf}$, a contradiction to the assumption that $(a, b) \notin \mathsf{rf} \cup \mathsf{rf}^{-1}$. The inclusion $(c, b) \in (\mathsf{po} \cup \mathsf{rf})^+$, however, contradicts Claim 3, because then $(a, b) \in [a]; \mathsf{rf}; [c]; (\mathsf{po} \cup \mathsf{rf})^+; [b] \subseteq [\{a, b\}]; (\mathsf{po} \cup \mathsf{rf})^+; (\mathsf{po} \cup \mathsf{rf}); [\{a, b\}]$.

Finally, we prove, without loss of generality, that, if $(a, b) \in \mathsf{rf}$, then b is $(\mathsf{po}|_D \cup \mathsf{rf}|_D)$-maximal. Because b is a $\mathsf{po}|_D$-maximal read event, the relation $[b]; (\mathsf{po}|_D \cup \mathsf{rf}|_D)$ is empty. Therefore, the relation $[b]; (\mathsf{po}|_D \cup \mathsf{rf}|_D)^+$ is empty, which conclusion finishes the proof. □

We now introduce the following sets:

$$\begin{aligned} D_{ab} &\triangleq D \setminus \{a, b\} \\ D_a &\triangleq D \setminus \{a\} \\ D_b &\triangleq D \setminus \{b\} \end{aligned}$$

Claim 5. The set D_{ab} is $(\mathsf{po} \cup \mathsf{rf})$-closed.
If $(a, b) \in \mathsf{rf}$ (resp. $(b, a) \in \mathsf{rf}$), then D_b (resp. D_a) is $(\mathsf{po} \cup \mathsf{rf})$-closed.
If $(a, b) \notin \mathsf{rf} \cup \mathsf{rf}^{-1}$, then the sets D_a and D_b are both $(\mathsf{po} \cup \mathsf{rf})$-closed.

Proof. To show that D_{ab} is $(\mathsf{po} \cup \mathsf{rf})$-closed, it suffices to prove that the inclusion $(\mathsf{po} \cup \mathsf{rf}); [D_{ab}] \subseteq [D_{ab}]; (\mathsf{po} \cup \mathsf{rf})$ holds. Suppose by contradiction that it does not, and let (c, d) be a pair such that $(c, d) \in (\mathsf{po} \cup \mathsf{rf}); [D_{ab}]$ and $(c, d) \notin [D_{ab}]; (\mathsf{po} \cup \mathsf{rf})$. Because D is $(\mathsf{po} \cup \mathsf{rf})$-closed (by definition), and because $D_{ab} \subseteq D$, it follows that $(c, d) \in [D]; (\mathsf{po} \cup \mathsf{rf})$. We thus conclude that $c \in \{a, b\}$, because $(c, d) \in ([D]; (\mathsf{po} \cup \mathsf{rf})) \setminus ([D_{ab}]; (\mathsf{po} \cup \mathsf{rf})) \subseteq [\{a, b\}]; (\mathsf{po} \cup \mathsf{rf})$. Because $d \in D_{ab} \subseteq D$, there exists $e \in \{a, b\}$ such that $(d, e) \in (\mathsf{po} \cup \mathsf{rf})^*$, by definition of D. In fact, it must be the case that $(d, e) \in (\mathsf{po} \cup \mathsf{rf})^+$, because $d \notin \{a, b\}$. We then reach a contradiction to Claim 3, because $(c, e) \in [\{a, b\}]; (\mathsf{po} \cup \mathsf{rf}); [d]; (\mathsf{po} \cup \mathsf{rf})^+; [\{a, b\}]; \subseteq [\{a, b\}]; (\mathsf{po} \cup \mathsf{rf})^+; (\mathsf{po} \cup \mathsf{rf}); [\{a, b\}]$. The remaining claims follow a similar proof. □

Claim 6. The graph $(G|_{D_{ab}}, \mathsf{rf}|_{D_{ab}}, \mathsf{mo}|_{D_{ab}})$ is not $(\mathsf{po}|_{D_{ab}} \cup \mathsf{rf}|_{D_{ab}}|_{\mathsf{sc}})$-racy.

Proof. Suppose by contradiction that $\mathtt{race}((\mathsf{po}|_{D_{ab}} \cup \mathsf{rf}|_{D_{ab}}|_{\mathsf{sc}})) \setminus (\mathsf{E}^{\mathsf{sc}} \times \mathsf{E}^{\mathsf{sc}})$ is non-empty, and let (c, d) be a pair in this set. We claim that $(c, d) \in S$. If not, then $(c, d) \in (\mathsf{po} \cup \mathsf{rf}|_{\mathsf{sc}})^+ \cup ((\mathsf{po} \cup \mathsf{rf}|_{\mathsf{sc}})^+)^{-1}$. Suppose without loss of generality that $(c, d) \in (\mathsf{po} \cup \mathsf{rf}|_{\mathsf{sc}})^+$. Because D_{ab} is $(\mathsf{po} \cup \mathsf{rf})$-closed, and because $c, d \in D_{ab}$, it follows that $(c, d) \in (\mathsf{po}|_{D_{ab}} \cup \mathsf{rf}|_{D_{ab}}|_{\mathsf{sc}})^+$, a

contradiction with the fact that (c, d) forms a $(\mathsf{po}|_{D_{ab}} \cup \mathsf{rf}|_{D_{ab}}|_{\mathsf{sc}})$-race. Therefore, it must be the case that $(c, d) \in S$. Because $c, d \in D$, it follows from the minimality of (a, b) that $(c, d) \in \{(a, b), (b, a)\}$, a contradiction with the fact that $c, d \in D_{ab}$. □

Claim 7. The graph $(G|_{D_a}, \mathsf{rf}|_{D_a}, \mathsf{mo}|_{D_a})$ is not $(\mathsf{po}|_{D_a} \cup \mathsf{rf}|_{D_a}|_{\mathsf{sc}})$-racy.
Analogously, the graph $(G|_{D_b}, \mathsf{rf}|_{D_b}, \mathsf{mo}|_{D_b})$ is not $(\mathsf{po}|_{D_b} \cup \mathsf{rf}|_{D_b}|_{\mathsf{sc}})$-racy.

Proof. Proof similar to Claim 6. □

Claim 8. The graph $(G|_{D_{ab}}, \mathsf{rf}|_{D_{ab}}, \mathsf{mo}|_{D_{ab}})$ is SC-consistent.

Proof. From Claim 5 and Lemma C.7, it follows that $G|_{D_{ab}}$ is $\mathrm{RC11}^{\mathrm{Ex86}}$-consistent. Finally, thanks to Claim 6 and Lemma C.12, it follows that $G|_{D_{ab}}$ is SC-consistent. □

Claim 9. If $(a, b) \in \mathsf{rf}$ (resp. $(b, a) \in \mathsf{rf}$), then $(G|_{D_b}, \mathsf{rf}|_{D_b}, \mathsf{mo}|_{D_b})$ (resp. $(G|_{D_a}, \mathsf{rf}|_{D_a}, \mathsf{mo}|_{D_a})$) is SC-consistent.

Proof. Proof similar to Claim 8. □

Claim 10. If $(a, b) \notin \mathsf{rf} \cup \mathsf{rf}^{-1}$, then $(G|_{D_a}, \mathsf{rf}|_{D_a}, \mathsf{mo}|_{D_a})$ and $(G|_{D_b}, \mathsf{rf}|_{D_b}, \mathsf{mo}|_{D_b})$ are SC-consistent.

Proof. Proof similar to Claim 8. □

Claim 11. The graph $(G|_D, \mathsf{rf}|_D, \mathsf{mo}|_D)$ is SC-consistent, and associated with p.

Proof. It follows from Lemma C.7 that $G|_D$ is $\mathrm{RC11}^{\mathrm{Ex86}}$-consistent, and associated with p. To prove that $G|_D$ is SC-consistent, suppose by contradiction that it is not, and let C be a cycle that violates SC-Coherence. Now, proceed by case disjunction on whether $(a, b) \in (\mathsf{rf} \cup \mathsf{mo} \cup \mathsf{rb}) \cup (\mathsf{rf} \cup \mathsf{mo} \cup \mathsf{rb})^{-1}$. If it does not, then a cycle that violates SC-Coherence must be included in D_{ab}. However, such a cycle would contradict Claim 6. Let us consider the case in which $(a, b) \in (\mathsf{rf} \cup \mathsf{mo} \cup \mathsf{rb}) \cup (\mathsf{rf} \cup \mathsf{mo} \cup \mathsf{rb})^{-1}$, and let us assume, without loss of generality, that $(a, b) \in (\mathsf{rf} \cup \mathsf{mo} \cup \mathsf{rb})$. We now consider the following cases:

- Case: $(a, b) \in \mathsf{rf}$.
  If $(a, b) \in \mathsf{rf}$, then, by Claim 9, the graph $G|_{D_b}$ is SC-consistent. Therefore, the cycle C must include b, otherwise it would violate the SC-consistency of $G|_{D_b}$. Because b is $\mathsf{po}|_D$-maximal (Claim 4), the only edge that can follow b in C is a rb edge. However, because $\mathsf{rf}; \mathsf{rb} \subseteq \mathsf{mo}$, the sequence $\mathsf{rf}; [b]; \mathsf{rb}$ can be replaced with a mo edge that avoids b, thereby yielding a cycle that violates the SC-consistency of $G|_{D_b}$.
- Case: $(a, b) \in (\mathsf{mo} \cup \mathsf{rb})$.
  If $(a, b) \in (\mathsf{mo} \cup \mathsf{rb})$, then, by Claim 10, both $G|_{D_a}$ and $G|_{D_b}$ are SC-consistent. Therefore, the cycle C must include a and b, otherwise it would violate the SC-consistency of either $G|_{D_a}$ or $G|_{D_b}$. Because b is $(\mathsf{po}|_D \cup \mathsf{rf}|_D)$-maximal (Claim 4), the only edge that can follow b in C is a mo edge. However, because $(\mathsf{mo} \cup \mathsf{rb}); \mathsf{mo} \subseteq (\mathsf{mo} \cup \mathsf{rb})$, the sequence $(\mathsf{mo} \cup \mathsf{rb}); [b]; \mathsf{mo}$ can be replaced with a $(\mathsf{mo} \cup \mathsf{rb})$ edge that avoids b, thereby yielding a cycle that violates the SC-consistency of $G|_{D_b}$.

□

The graph $G|_D$ contradicts the premise that p is $(\mathsf{po} \cup \mathsf{rf}|_{\mathsf{sc}})$-race free, because the graph $G|_D$ is (1) SC-consistent; (2) associated with p; and (3) contains a $(\mathsf{po} \cup \mathsf{rf}|_{\mathsf{sc}})$-race, the pair (a, b). □

C.2 Extension Property

Theorem C.14 (RC11^{Ex86}- Extension-I). *RC11-consistency conditions are equivalent to those of RC11^{Ex86} in every execution graph containing only RC11 events.*

Proof. In the absence of Ex86 events, the relation $\mathsf{po}_{\mathsf{RC11}}$ is equivalent to po, and the relation $\mathsf{ppo}_{\mathsf{asm}}$ is equivalent to $\mathsf{po}; [\mathsf{F}^{\mathsf{sc}}] \cup [\mathsf{F}^{\mathsf{sc}}]; \mathsf{po}$. Therefore, the only differences between the two models, with respect to the derived relations, is the definition of eco: in RC11, it is defined as $(\mathsf{rf} \cup \mathsf{mo} \cup \mathsf{rb})^+$, whereas, in RC11^{Ex86}, it is defined as $(\mathsf{rf}_e \cup \mathsf{mo} \cup \mathsf{rb})^+$. Let us use the names of these models as a prefix to distinguish to which version of eco (or of any other relation defined on top of eco) we refer.

To complete the proof it is thus sufficient to show that Coherence-II and Coherence-III are a consequence of RC11-consistency, and that, in the remaining RC11-consistency conditions, the relations RC11.eco and RC11^{Ex86}.eco can be used interchangeably. That is, it suffices to show that the following assertions hold:

(1) $\mathit{irreflexive}(\mathsf{hb}; \mathsf{RC11.eco}) \implies \mathit{irreflexive}(\mathsf{po}; \mathsf{rb})$
(2) $\mathit{acyclic}(\mathsf{RC11.psc}) \implies \mathit{acyclic}(\mathsf{ppo}_{\mathsf{asm}} \cup \mathsf{RC11}^{\mathsf{Ex86}}\mathsf{.eco})$
(3) $\mathit{irreflexive}(\mathsf{hb}; \mathsf{RC11.eco}^?) \iff \mathit{irreflexive}(\mathsf{hb}; \mathsf{RC11}^{\mathsf{Ex86}}\mathsf{.eco}^?)$
(4) $\mathit{acyclic}(\mathsf{RC11.psc}) \iff \mathit{acyclic}(\mathsf{RC11}^{\mathsf{Ex86}}\mathsf{.psc})$

Proof of Assertion (1). Immediate from $\mathsf{rb} \subseteq \mathsf{eco}$ and $\mathsf{po} \subseteq \mathsf{hb}$ (which holds of RC11-events-only execution graphs).

Proof of Assertion (2). We proceed by contradiction; that is, we show that, if $\mathsf{ppo}_{\mathsf{asm}} \cup \mathsf{RC11}^{\mathsf{Ex86}}\mathsf{.eco}$ is cyclic, then so is RC11.psc:

$$
\begin{aligned}
&\mathit{cyclic}(\mathsf{ppo}_{\mathsf{asm}} \cup \mathsf{RC11}^{\mathsf{Ex86}}\mathsf{.eco}) \\
&\implies \mathit{cyclic}(\mathsf{po}; [\mathsf{F}^{\mathsf{sc}}] \cup [\mathsf{F}^{\mathsf{sc}}]; \mathsf{po} \cup \mathsf{RC11.eco}) \\
&\implies \mathit{cyclic}(\mathsf{po}; [\mathsf{F}^{\mathsf{sc}}] \cup [\mathsf{F}^{\mathsf{sc}}]; \mathsf{po} \cup \mathsf{RC11.eco}) \\
&\implies \mathit{cyclic}(\mathsf{RC11}^{\mathsf{Ex86}}\mathsf{.eco}) \quad \textit{(Absurd)} \\
&\vee \quad \mathit{cyclic}([\mathsf{F}^{\mathsf{sc}}]; \mathsf{po}; \mathsf{RC11.eco}; \mathsf{po}; [\mathsf{F}^{\mathsf{sc}}]) \\
&\implies \mathit{cyclic}([\mathsf{F}^{\mathsf{sc}}]; \mathsf{hb}; \mathsf{RC11.eco}; \mathsf{hb}; [\mathsf{F}^{\mathsf{sc}}]) \\
&\implies \mathit{cyclic}([\mathsf{F}^{\mathsf{sc}}]; \mathsf{hb}; \mathsf{RC11.eco}; \mathsf{hb}; [\mathsf{F}^{\mathsf{sc}}]) \\
&\implies \mathit{cyclic}(\mathsf{RC11.psc})
\end{aligned}
$$

Proof of Assertion (3). Because RC11^{Ex86}.eco is included in RC11.eco, the left-to-right implication is trivial. The other direction follows by contradiction. The proof exploits the equality $\mathsf{eco} = \mathsf{rf} \cup (\mathsf{mo} \cup \mathsf{rb})^?; \mathsf{rf}^?$ and the inclusion $[\mathsf{E} \setminus \mathsf{W}^{\mathsf{nt}}]; \mathsf{rf}_i \subseteq \mathsf{hb}$:

$$
\begin{aligned}
&\neg\mathit{irreflexive}(\mathsf{hb}; \mathsf{RC11.eco}^?) \\
&\implies \neg\mathit{irreflexive}(\overbrace{\mathsf{hb}; \mathsf{rf}_i^?}^{\subseteq \mathsf{hb}}) \vee \neg\mathit{irreflexive}(\mathsf{hb}; \mathsf{rf}_e^?) \\
&\vee \quad \neg\mathit{irreflexive}(\mathsf{hb}; (\mathsf{mo} \cup \mathsf{rb})^?; \mathsf{rf}_e^?) \vee \underbrace{\neg\mathit{irreflexive}(\mathsf{hb}; (\mathsf{mo} \cup \mathsf{rb})^?; \mathsf{rf}_i^?)}_{\implies \neg\mathit{irreflexive}(\mathsf{rf}_i^?; \mathsf{hb}; (\mathsf{mo} \cup \mathsf{rb})^?)} \\
&\implies \neg\mathit{irreflexive}(\mathsf{hb}; \mathsf{rf}_e^?) \vee \neg\mathit{irreflexive}(\mathsf{hb}; (\mathsf{mo} \cup \mathsf{rb})^?; \mathsf{rf}_e^?) \\
&\implies \neg\mathit{irreflexive}(\mathsf{hb}; \mathsf{RC11}^{\mathsf{Ex86}}\mathsf{.eco}^?)
\end{aligned}
$$

Proof of Assertion (4). We show the following equality (which holds of RC11-events-only execution graphs):

$$[\mathsf{F}^{\mathsf{sc}}]; \mathsf{hb}; \mathsf{RC11.eco}; \mathsf{hb}; [\mathsf{F}^{\mathsf{sc}}] = [\mathsf{F}^{\mathsf{sc}}]; \mathsf{hb}; \mathsf{RC11}^{\mathsf{Ex86}}\mathsf{.eco}; \mathsf{hb}; [\mathsf{F}^{\mathsf{sc}}]$$

Because $\text{RC11}^{\text{Ex86}}.\text{eco} \subseteq \text{RC11.eco}$, it is easy to see that the relation on the right-hand side of the equality is included in the relation on the left-hand side. Let us now show the inclusion in the other direction:

$$[\mathsf{F}^{\mathsf{sc}}];\ \text{hb};\ \text{RC11.eco};\ \text{hb};\ [\mathsf{F}^{\mathsf{sc}}] \subseteq [\mathsf{F}^{\mathsf{sc}}];\ \text{hb};\ \text{RC11}^{\text{Ex86}}.\text{eco};\ \text{hb};\ [\mathsf{F}^{\mathsf{sc}}]$$

Exploiting the equality $\text{RC11.eco} = \text{rf} \cup (\text{mo} \cup \text{rb});\ \text{rf}^?$ and the inclusion $[\mathsf{E} \setminus \mathsf{W}^{\mathsf{nt}}];\ \text{rf}_i \subseteq \text{hb}$, it is then easy to see that every edge rf_i in RC11.eco can be merged into the hb edge that either precedes or succeeds RC11.eco.

□

Theorem C.15 ($\text{RC11}^{\text{Ex86}}$- Extension-II). *Ex86-consistency conditions are equivalent to those of $RC11^{Ex86}$ in every execution graph containing only Ex86 events.*

Proof. The proof is split into two parts. First we prove that Ex86-consistency implies $\text{RC11}^{\text{Ex86}}$-consistency, then we prove the converse:

(1) Ex86-consistency $\implies$ $\text{RC11}^{\text{Ex86}}$-consistency.
It is thus sufficient to prove that following conditions hold:

- Coherence-I.
The proof follows by contradiction. We prove that, in the absence of RC11 events, the violation of Coherence-I leads to the violation of Ex86-consistency or to the violation of Coherence-II (which we show to hold in the next item):

$$\begin{aligned}
&\neg \mathit{irreflexive}(\text{hb};\ \text{eco}^?)\\
&\implies \neg \mathit{irreflexive}(\text{hb}) \ \vee\ \neg \mathit{irreflexive}(\text{hb};\ \text{eco})
\end{aligned}$$

$$\begin{aligned}
&\neg \mathit{irreflexive}(\text{hb})\\
&\implies \mathit{cyclic}(\underbrace{[\mathsf{RMW}^{\mathsf{tso}} \cup \mathsf{R}^{\mathsf{tso}}];\ \text{po} \cup \text{rf}_e}_{\subseteq\ \text{Ex86.ppo}})\\
&\implies \mathit{cyclic}(\text{Ex86.ppo} \cup \text{rf}_e \cup \text{mo}_e \cup \text{rb}_e)
\end{aligned}$$

$$\begin{aligned}
&\neg \mathit{irreflexive}(\text{hb};\ \text{eco})\\
&\implies \neg \mathit{irreflexive}\left(\underbrace{\begin{pmatrix}[\mathsf{RMW}^{\mathsf{tso}} \cup \mathsf{R}^{\mathsf{tso}}];\ \text{po} \cup\\ [\mathsf{W}^{\mathsf{tso}}];\ \text{po};\ [\mathsf{E} \setminus \mathsf{R} \setminus \mathsf{W}^{\mathsf{nt}}] \cup\\ [\mathsf{W}^{\mathsf{nt}}];\ \text{po};\ [\mathsf{RMW}^{\mathsf{tso}} \cup \mathsf{F}^{\sqsupseteq \mathsf{sf}}];\ \text{po}^? \cup\\ [\mathsf{W}^{\mathsf{nt}}];\ \text{po}|_{\text{loc}}\end{pmatrix}}_{\subseteq\ [\mathit{codom}(\text{eco})];\ \text{po}_{\text{RC11}};\ ([\mathsf{F}^{\sqsupseteq \mathsf{sf}}];\ \text{po})^?;\ [\mathsf{W}^{\mathsf{tso}}]\ (\text{rf}_e;\ \text{po}^?)^+;\ \text{eco}}\right)\\
&\implies \neg \mathit{irreflexive}(\text{ppo}_{\text{asm}};\ (\text{rf}_e;\ (\underbrace{[\mathsf{R}^{\mathsf{tso}} \cup \mathsf{RMW}^{\mathsf{tso}}];\ \text{po}}_{\subseteq\ \text{ppo}_{\text{asm}}})^?)^+;\ \text{eco})\\
&\implies \mathit{cyclic}(\text{ppo}_{\text{asm}} \cup \text{eco})
\end{aligned}$$

- Coherence-II.
It suffices to exploit the inclusion $(\text{mo}_i \cup \text{rb}_i) \subseteq \text{po}|_{\text{loc}};\ [\mathsf{E} \setminus \mathsf{R}] \subseteq \text{Ex86.ppo}$ to show

that the violation of Coherence-II leads to the violation of External:

$$\begin{aligned} \mathit{cyclic}(\mathsf{ppo}_{\mathsf{asm}} \cup \mathsf{eco}) &\implies \mathit{cyclic}(\mathsf{ppo}_{\mathsf{asm}} \cup \mathsf{rf}_e \cup \mathsf{mo} \cup \mathsf{rb}) \\ &\implies \mathit{cyclic}(\underbrace{(\mathsf{ppo}_{\mathsf{asm}} \cup \mathsf{mo}_i \cup \mathsf{rb}_i)}_{\subseteq \mathsf{Ex86.ppo}} \cup \mathsf{rf}_e \cup \mathsf{mo}_e \cup \mathsf{rb}_e) \\ &\implies \mathit{cyclic}(\mathsf{Ex86.ppo} \cup \mathsf{rf}_e \cup \mathsf{mo}_e \cup \mathsf{rb}_e) \end{aligned}$$

- Coherence-III. (Immediate from Internal.)
- No-Thin-Air.
  We show that the negation of No-Thin-Air leads to a contradiction with External:

$$\begin{aligned} \mathit{cyclic}(\mathsf{po} \cup \mathsf{rf}) &\implies \mathit{cyclic}(\underbrace{(\mathsf{po} \cup \mathsf{rf}_i)}_{\subseteq \mathsf{po}} \cup \mathsf{rf}_e) \\ &\implies \mathit{cyclic}(\underbrace{([\mathsf{R}^{\mathsf{tso}} \cup \mathsf{RMW}^{\mathsf{tso}}]; \mathsf{po})}_{\subseteq \mathsf{Ex86.ppo}} \cup \mathsf{rf}_e) \\ &\implies \mathit{cyclic}(\mathsf{Ex86.ppo} \cup \mathsf{rf}_e \cup \mathsf{mo}_e \cup \mathsf{rb}_e) \end{aligned}$$

(2) $\mathsf{RC11}^{\mathsf{Ex86}}$-consistency $\implies$ Ex86-consistency.
Condition External is an immediate consequence of Coherence-II. The proof of Condition Internal is split into the three following subconditions:

- $\mathit{irreflexive}(\mathsf{po}; \mathsf{rf}_i)$. (Immediate from No-Thin-Air.)
- $\mathit{irreflexive}(\mathsf{po}; \mathsf{mo}_i)$.
  The violation of this condition leads to a contradiction with Coherence-II:

$$\begin{aligned} \neg\mathit{irreflexive}(\mathsf{po}; \mathsf{mo}_i) &\implies \neg\mathit{irreflexive}(\underbrace{(\mathsf{po}|_{\mathsf{loc}}; [\mathsf{E} \setminus \mathsf{R}])}_{\subseteq \mathsf{ppo}_{\mathsf{asm}}}; \mathsf{mo}_i) \\ &\implies \neg\mathit{irreflexive}(\mathsf{ppo}_{\mathsf{asm}}; \mathsf{eco}) \\ &\implies \mathit{cyclic}(\mathsf{ppo}_{\mathsf{asm}} \cup \mathsf{eco}) \end{aligned}$$

- $\mathit{irreflexive}(\mathsf{po}; \mathsf{rb}_i)$. (Immediate from Coherence-III.)

□

D COMPILATION

D.1 Compilation Schemes

We present two compilation schemes from $\mathsf{RC11}^{\mathsf{Ex86}}$-lang to Ex86-lang: one that naturally extends the scheme studied by Lahav et al. [14, Fig. 8], and a slightly more elaborated one that maps `rlx` writes to non-temporal stores, and adds store fences to the mapping of `rel`/`sc` writes and to the mapping of `rel`/`acqrel` fences. We prove that these schemes are correct (with respect to $\mathsf{RC11}^{\mathsf{Ex86}}$) in §D.3.

Definition D.1 (Compilation Scheme from $\mathsf{RC11}^{\mathsf{Ex86}}$-lang to Ex86-lang).

$$\begin{array}{lll} (\!|\, [e]^{\mathsf{sc}} := e' \,|\!) \triangleq [e] := e'; \mathtt{mfence} & (\!|\, \mathtt{fence}_{\mathsf{sc}} \,|\!) \triangleq \mathtt{mfence} & (\!|\, s; s' \,|\!) \triangleq (\!|\, s \,|\!); (\!|\, s' \,|\!) \\ (\!|\, [e]^{\neq \mathsf{sc}} := e' \,|\!) \triangleq [e] := e' & (\!|\, \mathtt{fence}_{\neq \mathsf{sc}} \,|\!) \triangleq \mathtt{skip} & (\!|\, \mathtt{skip} \,|\!) \triangleq \mathtt{skip} \\ (\!|\, r := [e]^{md} \,|\!) \triangleq r := [e] & (\!|\, \mathtt{if}\ e\ \{\ s\ \} \,|\!) \triangleq \mathtt{if}\ e\ \{\ (\!|\, s \,|\!)\ \} & (\!|\, \mathbf{asm}\{s\} \,|\!) \triangleq s \\ (\!|\, r := \mathsf{rmw}_{md}([e_1], e_2, e_3) \,|\!) \triangleq r := \mathsf{rmw}([e_1], e_2, e_3) & & (\!|\, \mathtt{while}\ e\ \{\ s\ \} \,|\!) \triangleq \mathtt{while}\ e\ \{\ (\!|\, s \,|\!)\ \} \end{array}$$

Definition D.2 (Alternative Compilation Scheme). Same as Def. D.1 except for the following cases:

$$\begin{array}{ll} (\!|\, [e]^{\mathsf{sc}} := e' \,|\!)\text{-alt} \triangleq \mathtt{sfence}; [e] := e'; \mathtt{mfence} & (\!|\, [e]^{\mathsf{rlx}} := e' \,|\!)\text{-alt} \triangleq [e] :=_{\mathsf{nt}} e' \\ (\!|\, [e]^{\mathsf{rel}} := e' \,|\!)\text{-alt} \triangleq \mathtt{sfence}; [e] := e' & (\!|\, \mathtt{fence}_{\mathsf{rel},\,\mathsf{acqrel}} \,|\!)\text{-alt} \triangleq \mathtt{sfence} \end{array}$$

D.2 Mixed Execution Graphs

Our proofs of compilation correctness (§D.3) rely on the novel notion of *mixed execution graphs*, a type of execution graph whose nodes contain events from both the source-level and target-level models. Before presenting our results of compilation correctness, we discuss the definition and some of the properties of mixed execution graphs.

Informally speaking, a mixed execution graph is the superposition of two execution graphs: one called *source graph*, which is associated with a source program p; and one called *target graph*, which is associated with the compilation of p. The key feature of a mixed execution graph is that it captures the fact that source and target graphs share the same overall structure. Indeed, because a compilation scheme preserves the control flow of the source program and changes only how memory operations are mapped to operations in the target language, for every execution graph of the compiled program, one can always construct an execution graph of the source program that preserves much of the structure of the target graph, including its primitive relations po, rf, and mo. The only mismatches between these graphs come from how one memory operation from the source language might be mapped to zero, one, or multiple memory operations from the target language.

To account for these mismatches, nodes in a mixed graph, called *mixed nodes*, carry events from both source and target models. Events from the two models however cannot be arbitrarily assembled in a mixed node: the source-level events in a mixed node correspond to the events of a single source instruction and the target-level events correspond to the events emitted by the snippet of target-level language produced by the mapping of this instruction. Therefore, the range of mixed nodes is fixed and determined by the underlying compilation scheme.

Mixed graphs form a very convenient tool for proving compilation-correctness results because they allow one to work with the execution graphs from both the source program and its compiled version at the same time, and because they allow one to forget about the compilation scheme which is ultimately encoded in the set of permissible mixed nodes. Moreover, it is possible to lift the consistency conditions from the models of source and target languages to this mixed-graph structure. Both models can thus be defined on the same structure, thereby allowing one to formally reason about statements of the kind "*one model is stronger than the other*". In fact, the main convenience of mixed execution graphs is precisely to allow one to formulate the compilation correctness result as a statement in this fashion: "*in a mixed execution graph with nodes taken from a well-chosen set, if the consistency conditions of the target model hold, then so do the consistency conditions of the source model*". The set of nodes has to be well chosen so as to correctly reflect the compilation scheme begin considered.

To give an illustration of mixed execution graphs, let us consider $\mathsf{RC11}^{\mathsf{Ex86}}$ as the source model, Ex86 as the target model, and $(\!|\_|\!)$ (Definition D.1) as the compilation scheme.

Figure 15 shows our choice for the set of permissible mixed nodes. The nodes are depicted as domino-shaped boxes where the left component stores $\mathsf{RC11}^{\mathsf{Ex86}}$ events and the right component stores Ex86 events. We use the symbol $\bot$ to denote an empty set of events (in addition to its meaning as the *none* element of an option type). It is easy to see how this definition mimics the compilation scheme from Definition D.1. Indeed, Node R-R reflects how read instructions are compiled to plain reads. Node W-WMF reflects how a `sc` write is compiled to a plain write followed by a memory fence. Moreover, node F-$\bot$ reflects how fences weaker than `sf` are erased by the compilation scheme. Nodes W-NT and F-SF reflects the compilation of inline-assembly instructions. Finally, nodes RMW-RMW-S and RMW-RMW-F reflect the compilation of read-modify-writes.

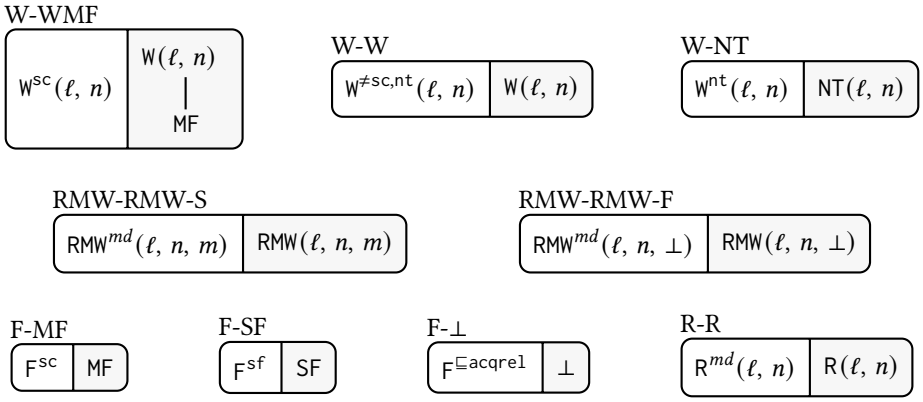
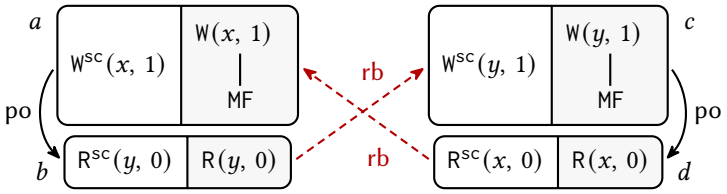


Fig. 15. Set of mixed nodes reflecting the compilation scheme from Definition D.1.

To see an example of a mixed execution graph constructed with these nodes, consider the following program:

$$p \triangleq \left(\begin{array}{l} [x]^{\mathsf{sc}} := 1; \\ r := [y]^{\mathsf{sc}} \end{array} \;\middle\|\; \begin{array}{l} [y]^{\mathsf{sc}} := 1; \\ r := [x]^{\mathsf{sc}} \end{array} \right)$$

This program implements the *store-buffering litmus test* (SB). SB is one of the simplest demonstrations of non-sequentially consistent behaviors: it would happen if both read instructions returned the value 0. The RC11$^{\mathsf{Ex86}}$-lang program p, however, exhibits only sequentially consistent behaviors because the access mode of all memory instructions is sc. The following mixed execution graph allows us to see simultaneously how RC11$^{\mathsf{Ex86}}$ rules out SB in p and how Ex86 rules out SB in the compilation of p, the program $(\!|\, p \,|\!)$:



To show that SB is ruled out (in both source and compiled programs), we must show that both RC11$^{\mathsf{Ex86}}$ and Ex86 graphs are inconsistent. Indeed, both graphs are inconsistent because of the cycle (a, b, c, d). In the Ex86 graph, this cycle violates Condition External. In the RC11$^{\mathsf{Ex86}}$ graph, this cycle violates Condition SC. Here is a summary of the technical arguments sustaining these claims:

Cycle in psc:

- $(a, b), (c, d) \in [\mathsf{E}^{\mathsf{sc}}]; \mathsf{po}; [\mathsf{E}^{\mathsf{sc}}] \subseteq \mathsf{psc}_{\mathsf{base}} \subseteq \mathsf{psc}$
- $(b, c), (d, a) \in [\mathsf{E}^{\mathsf{sc}}]; \mathsf{rb}; [\mathsf{E}^{\mathsf{sc}}] \subseteq \mathsf{psc}_{\mathsf{base}} \subseteq \mathsf{psc}$

Cycle in $\mathsf{ppo} \cup \mathsf{rf}_e \cup \mathsf{mo}_e \cup \mathsf{rb}_e$:

- $(a, b), (c, d) \in$ Ex86.ppo
- $(b, c), (d, a) \in \mathsf{rb}_e$

At first glance, the outlined arguments might seem informal because the relations so specified apply only to events of a specific model, not to events of the mixed graph. However, we show that these arguments can be made valid: in essence, it suffices to lift the relations from source and target models to the structure of mixed graphs. That is, when working with mixed graphs, we manipulate custom versions of these relations defined as relations on mixed nodes.

The following definition formally introduces mixed execution graphs and its custom version of the relations rf and mo:

Definition D.3 (Mixed Execution Graph). A *mixed execution graph* is a graph where every node, called a *mixed node*, is a pair of a set of $\text{RC11}^{\text{Ex86}}$ events and a set of Ex86 events. Figure 15 depicts the set of mixed nodes allowed in a mixed graph. The two basic relations are rf and mo:

(1) *Reads-from (rf).* The reads-from relation is a surjective and functional relation with domain and codomain specified as follows:

$$\text{W-W} \cup \text{W-WMF} \cup \text{W-NT} \cup \text{RMW-RMW-S} \xrightarrow{\text{rf}} \text{R-R} \cup \text{RMW-RMW-S} \cup \text{RMW-RMW-F}$$

(2) *Modification-order (mo).* The modification-order has domain and codomain specified as follows:

$$\text{W-W} \cup \text{W-WMF} \cup \text{W-NT} \cup \text{RMW-RMW-S} \xrightarrow{\text{mo}} \text{W-W} \cup \text{W-WMF} \cup \text{W-NT} \cup \text{RMW-RMW-S}$$

We introduce the following sets of mixed nodes:

$$\begin{aligned} \texttt{W} &\triangleq \text{W-W} \cup \text{W-WMF} \cup \text{W-NT} & \qquad \texttt{NT} &\triangleq \text{W-NT} \\ \texttt{RMW} &\triangleq \text{RMW-RMW-S} \cup \text{RMW-RMW-F} & \qquad \texttt{SF} &\triangleq \text{F-SF} \\ \texttt{R} &\triangleq \text{R-R} & \qquad \texttt{MF} &\triangleq \text{F-MF} \\ \texttt{F} &\triangleq \text{F-MF} \cup \text{F-SF} \cup \text{F-}\bot \end{aligned}$$

When applicable, we annotate sets of mixed nodes with superscripts of the form "$\sqsupseteq$ *md*" (and variations of it) to specify the range of access modes of the RC11 events in the left component of mixed nodes.

Naturally, reasoning at the level of mixed graphs and its corresponding version of the relations rf and mo leads to facts about mixed graphs and nodes; to extract a result about the source and target models, we provide a theorem that allows one to transfer results between these structures. For example, we prove that, if the consistency conditions of $\text{RC11}^{\text{Ex86}}$ hold of a mixed graph, then it also holds of the source graph.

Before we introduce this theorem, let us clarify the notions of *source graph* and *target graph*. These concepts are not yet well defined, because, given a mixed graph, we have not explained how they can be obtained. The missing piece of information is the notion of *source* and *target projections*: given a mixed graph G_m, its source projection $\lfloor G_m$ and target projection $G_m \rfloor$ correspond to the source and target graphs whose superposition is G_m.

Definition D.4 (Projections). Let G_m be a mixed execution graph. The *source* and *target projections* of G_m, noted $\lfloor G_m$ and $G_m \rfloor$, are $\text{RC11}^{\text{Ex86}}$ and Ex86 execution graphs. The nodes of $\lfloor G_m$ and $G_m \rfloor$ correspond to the first and second parts of G_m nodes. The edges of $\lfloor G_m$ and $G_m \rfloor$ are constructed through *projection rules*. A projection rule formalizes the correspondence between edges in G_m (appearing at the top of the rule) and the edges in $\lfloor G_m$ and $G_m \rfloor$ (appearing respectively at the bottom left and at the bottom right of the rule). Figure 16 shows a selection of the projection rules.

Definition D.5 (Mixed Execution Graph - Ex86-Consistency). A mixed execution graph (G_m, rf, mo) is *Ex86-consistent* if the conditions from Definition B.1 hold when the sets and relations to their corresponding mixed-graph versions as introduced in Definition D.3, and the ppo relation with the

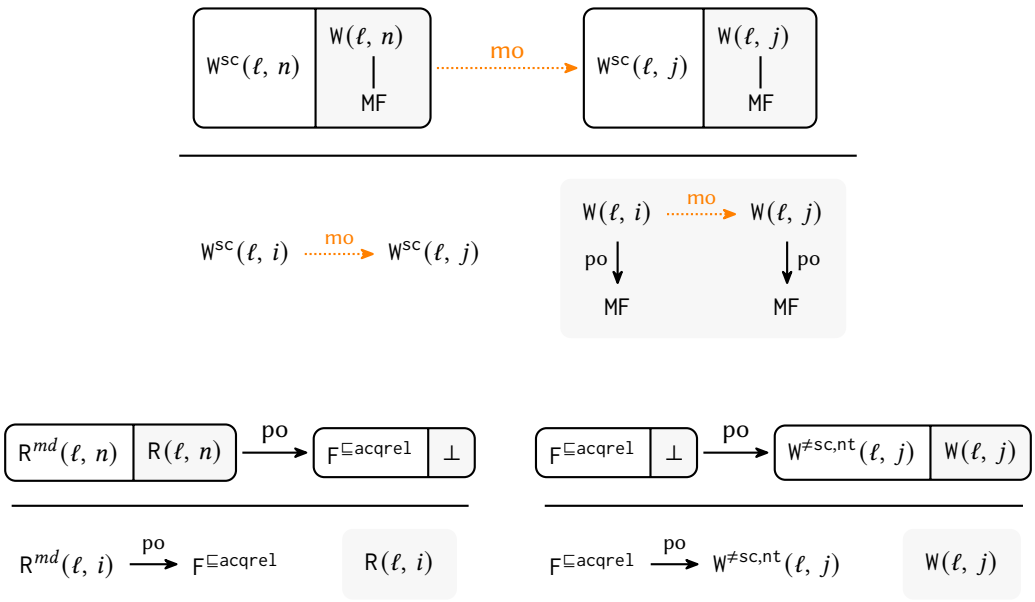


Fig. 16. Selection of projection rules.

following one:

$$
\begin{aligned}
\mathsf{ppo} \triangleq\ & [\mathsf{E} \setminus \mathsf{F}\text{-}\bot]; \mathsf{po}; [\mathsf{RMW} \cup \mathsf{MF} \cup \mathsf{SF}] \\
\cup\ & [\mathsf{R} \cup \mathsf{RMW} \cup \mathsf{MF}]; \mathsf{po}; [\mathsf{E} \setminus \mathsf{F}\text{-}\bot] \\
\cup\ & [\mathsf{E} \setminus \mathsf{F}\text{-}\bot]; \mathsf{po}^{?}; [\mathsf{W}\text{-}\mathsf{WMF}]; \mathsf{po}; [\mathsf{E} \setminus \mathsf{F}\text{-}\bot] \\
\cup\ & [\mathsf{SF}]; \mathsf{po}; [\mathsf{E} \setminus \mathsf{F}\text{-}\bot \setminus \mathsf{R}] \\
\cup\ & [\mathsf{W}^{\neq \mathsf{nt}}]; \mathsf{po}; [\mathsf{W}^{\neq \mathsf{nt}}] \\
\cup\ & [\mathsf{W}]; \mathsf{po}|_{\mathsf{loc}}; [\mathsf{W}]
\end{aligned}
$$

Definition D.6 (Mixed Execution Graph - RC11-Consistency). A mixed execution graph $(G_m, \mathsf{rf}, \mathsf{mo})$ is *RC11^{Ex86}-consistent* if the conditions from Definition B.8 hold when we replace the sets and relations to their corresponding mixed-graph versions as introduced in Definition D.3.

Theorem D.7 (Transfer Principle). *Let $(G_m, \mathsf{rf}, \mathsf{mo})$ be a mixed execution graph. The consistency conditions from Definition D.6 hold of G_m if, and only if, the RC11^{Ex86}-consistency conditions (Definition B.8) hold of $\lfloor G_m$. Analogously, the consistency conditions from Definition D.5 hold of G_m if, and only if, the Ex86-consistency conditions (Definition B.1) hold of $G_m \rfloor$.*

Proof. The first claim, that G_m is $\mathsf{RC11}^{\mathsf{Ex86}}$-consistent iff $\lfloor G_m$ is $\mathsf{RC11}^{\mathsf{Ex86}}$-consistent, is easy to see because every mixed node carries exactly one $\mathsf{RC11}^{\mathsf{Ex86}}$ event. Therefore, the projected nodes and relations can be related by a one-to-one correspondence. Since the consistency conditions from Definition D.6 are essentially the same as those from $\mathsf{RC11}^{\mathsf{Ex86}}$ (Definition B.8), this observation is sufficient to establish this claim.

The second claim, that G_m is Ex86-consistent iff $G_m \rfloor$ is Ex86-consistent, is slightly more intricate to prove than the previous one because of the nodes F-⊥ and W-WMF, which do not have a one-to-one correspondence with the projected Ex86 events; and because of the differences between the two versions of ppo from Definitions B.1 and D.5. That the Ex86-consistency of $G_m \rfloor$ implies that of G_m follows from the fact that the projection of every edge in $G_m.\mathsf{ppo}$ is an edge in $(G_m \rfloor).\mathsf{ppo}$.

Indeed, it is easy to see that the two problematic types of nodes, F-⊥ and W-WMF, are correctly handled by the definition of G_m.ppo: nodes of type F-⊥ are excluded from G_m.ppo, and nodes of type W-WMF always have a trailing po edge so that a memory fence is always between the two endpoints of the resulting projected edge. This concludes one direction of the logical equivalence. To prove the converse, that the Ex86-consistency of G_m implies that of $G_m\downarrow$, it suffices to show that every $(G_m\downarrow).\text{ppo}^+$ edge in a cycle that violates External is the projection of a $G_m.\text{ppo}^+$ edge. This condition can be easily checked; the only non-trivial case is when there is a memory-fence event between the endpoints of a $(G_m\downarrow).\text{ppo}^+$ edge, because this memory-fence could be the projection of either a F-MF node or a W-WMF node. In both cases, it is easy to see that the $(G_m\downarrow).\text{ppo}^+$ edge is the projection of an edge of type $G_m.([\mathsf{E} \setminus \text{F-}\bot]; \text{po}^?; [\mathtt{MF} \cup \text{W-WMF}]; \text{po}; [\mathsf{E} \setminus \text{F-}\bot])$, which is included in $G_m.\text{ppo}^+$. □

To conclude this discussion, we introduce the notion of *graph simulation*, an auxiliary concept for our upcoming compilation-correctness proofs (§D.3):

Definition D.8 (Graph Simulation). A $\mathsf{RC11}^{\mathsf{Ex86}}$ graph G is *simulated by* a Ex86 graph G', noted $G \sim G'$, if there exists a mixed execution graph G_m such that $\downarrow G_m = G$ and $G_m\downarrow = G'$.

D.3 Compilation Correctness

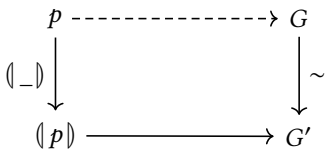
We state and prove correctness of the compilation schemes from Definitions D.1 and D.2. The statement of correctness is straightforward:

Theorem D.9 (Correctness of Definition D.1). *For every program p, the set of behaviors of $(\!|\, p \,|\!)$ defined by Ex86 is included in the set of behaviors of p defined by $RC11^{Ex86}$:*

$$\forall p.\ [\![(\!|\, p \,|\!)]\!]_{Ex86} \subseteq [\![p]\!]_{RC11^{Ex86}}$$

Theorem D.10 (Correctness of Definition D.2). *Statement analogous to Theorem D.9*

D.3.1 Proof Sketch. The overall structure of our proofs is depicted by the following diagram:



It illustrates the first step of a two-steps strategy to prove that $(\!|\_|\!)$ is correct. This first step consists of showing that, for every program p, for every execution graph G' associated with $(\!|\, p \,|\!)$, there exists a graph G associated with p, such that $G \sim G'$. The second step is then to show that, if the consistency conditions from the target model hold of a mixed execution graph G_m, then the consistency conditions from the source model also hold of G_m. Finally, by invoking Theorem D.7, it follows that, if G' is consistent (with respect to the target model), then G is consistent (with respect to the source model). In particular, this implies that every final state of $(\!|\, p \,|\!)$ is a final state of p, which statement corresponds precisely to the formulation of compilation correctness.

The first step is accomplished by induction over the construction of the graph G'. Intuitively, because the compiled program $(\!|\, p \,|\!)$ preserves much of the structure of p, it is possible to replay the pool-reduction steps (Figures 7 and 8) from $(\!|\, p \,|\!)$ and yield a graph G that satisfies the desired properties.

The second step is the crux of our proofs and it is where we concentrate our attention. Next, we discuss how to accomplish this step in the case of Ex86, first considering the standard compilation scheme (Definition D.1) and then the alternative one (Definition D.2).

D.3.2 Compilation to Ex86.

Lemma D.11 (RC11$^{\text{Ex86}}$-Weaker-Than-Ex86). *Let G_m be a mixed execution graph. If G_m is Ex86-consistent, then G_m is RC11$^{\text{Ex86}}$-consistent.*

Proof. Suppose that G_m is Ex86-consistent but not RC11$^{\text{Ex86}}$-consistent. Then at least one of RC11$^{\text{Ex86}}$ consistency conditions must not hold of G_m. We show that the violation of any of them leads to a contradiction:

(1) Coherence-I.
The violation of Coherence-I implies that (at least) one of the following assertions holds:

(a) Assertion: $\neg \mathit{irreflexive}(\mathsf{hb})$

$$\begin{aligned}\neg\mathit{irreflexive}(\mathsf{hb}) &\implies \mathit{cyclic}(\mathsf{po} \cup \mathsf{rf}_e)\\ &\implies \mathit{cyclic}(\underbrace{([\mathit{codom}(\mathsf{rf}_e)]; \mathsf{po})}_{\subseteq \mathsf{ppo}} \cup \mathsf{rf}_e)\\ &\implies \mathit{cyclic}(\mathsf{ppo} \cup \mathsf{rf}_e \cup \mathsf{mo}_e \cup \mathsf{rb}_e)\end{aligned}$$

(b) Assertion: $\neg \mathit{irreflexive}(\mathsf{hb}; \mathsf{rf}_e)$

$$\begin{aligned}\neg\mathit{irreflexive}(\mathsf{hb}; \mathsf{rf}_e) &\implies \mathit{cyclic}(\mathsf{po} \cup \mathsf{rf}_e)\\ &\implies \mathit{cyclic}(\mathsf{ppo} \cup \mathsf{rf}_e \cup \mathsf{mo}_e \cup \mathsf{rb}_e)\end{aligned}$$

(c) Assertion: $\neg \mathit{irreflexive}(\mathsf{hb}; (\mathsf{mo} \cup \mathsf{rb}))$

$$\begin{aligned}\neg\mathit{irreflexive}(\mathsf{hb}; (\mathsf{mo}_i \cup \mathsf{rb}_i)) &\implies \neg\mathit{irreflexive}((\mathsf{po} \cup \mathsf{rf}_e)^+; (\mathsf{mo}_i \cup \mathsf{rb}_i))\\ &\implies \neg\mathit{irreflexive}((\mathsf{po} \cup \mathsf{rf}_e)^+; \mathsf{po})\\ &\implies \mathit{cyclic}(\mathsf{po} \cup \mathsf{rf}_e)\\ &\implies \mathit{cyclic}(\mathsf{ppo} \cup \mathsf{rf}_e \cup \mathsf{mo}_e \cup \mathsf{rb}_e)\end{aligned}$$

$$\begin{aligned}&\neg\mathit{irreflexive}(\mathsf{hb}; (\mathsf{mo}_e \cup \mathsf{rb}_e))\\ &\implies \neg\mathit{irreflexive}((\mathsf{po}_{\text{RC11}} \cup \mathsf{sw})^+; (\mathsf{mo}_e \cup \mathsf{rb}_e)) \implies\\ &\neg\mathit{irreflexive}\left(\begin{array}{l} \subseteq \left(\begin{array}{l}[\mathsf{W}^{\neq\mathsf{nt}}]; \mathsf{po}; [\mathsf{W}^{\neq\mathsf{nt}}] \cup\\ [\mathsf{W}^{\mathsf{nt}}]; \mathsf{po}|_{\mathsf{loc}}; [\mathsf{W}] \cup\\ [\mathsf{W}^{\mathsf{nt}}]; \mathsf{po}; [\mathsf{RMW}^{\mathsf{tso}} \cup \mathsf{F}^{\sqsupseteq\mathsf{sf}}]; \mathsf{po}\end{array}\right) \subseteq \mathsf{ppo}\\ \overbrace{\begin{array}{l}[\mathit{codom}(\mathsf{mo} \cup \mathsf{rb})]; \mathsf{po}_{\text{RC11}}; [\mathsf{E}^{\sqsupseteq\mathsf{rel}}]; ([\mathsf{F}]; \mathsf{po})^?; [\mathsf{W}^{\sqsupseteq\mathsf{rlx}}];\\ (\mathsf{rf}_e; \mathsf{ppo}^?)^+; (\mathsf{mo}_e \cup \mathsf{rb}_e)\end{array}}\end{array}\right)\\ &\implies \neg\mathit{irreflexive}(\mathsf{ppo}; (\mathsf{rf}_e; \mathsf{ppo}^?)^+; (\mathsf{mo}_e \cup \mathsf{rb}_e))\\ &\implies \mathit{cyclic}(\mathsf{ppo} \cup \mathsf{rf}_e \cup \mathsf{mo}_e \cup \mathsf{rb}_e)\end{aligned}$$

(d) Assertion: $\neg \mathit{irreflexive}(\mathsf{hb}; (\mathsf{mo} \cup \mathsf{rb}); \mathsf{rf}_e)$

$$\begin{aligned}\neg\mathit{irreflexive}(\mathsf{hb}; (\mathsf{mo}_i \cup \mathsf{rb}_i); \mathsf{rf}_e) &\implies \neg\mathit{irreflexive}(\mathsf{hb}; \mathsf{po}_{\text{RC11}}; \mathsf{rf}_e)\\ &\implies \neg\mathit{irreflexive}(\mathsf{hb}; \mathsf{rf}_e)\end{aligned}$$

$$\begin{aligned}&\neg\mathit{irreflexive}(\mathsf{hb}; (\mathsf{mo}_e \cup \mathsf{rb}_e); \mathsf{rf}_e)\\ &\implies \neg\mathit{irreflexive}((\mathsf{po} \cup \mathsf{rf}_e)^*; (\mathsf{mo}_e \cup \mathsf{rb}_e); \mathsf{rf}_e)\\ &\implies \neg\mathit{irreflexive}([\mathit{codom}(\mathsf{rf})]; \mathsf{po}; (\mathsf{rf}_e; \mathsf{po}^?)^*; (\mathsf{mo}_e \cup \mathsf{rb}_e); \mathsf{rf}_e)\\ &\implies \neg\mathit{irreflexive}(\mathsf{ppo}; (\mathsf{rf}_e; \mathsf{ppo}^?)^*; (\mathsf{mo}_e \cup \mathsf{rb}_e); \mathsf{rf}_e)\\ &\implies \mathit{cyclic}(\mathsf{ppo} \cup \mathsf{rf}_e \cup \mathsf{mo}_e \cup \mathsf{rb}_e)\end{aligned}$$

(2) Coherence-II.

$$\begin{aligned} \mathit{cyclic}(\mathrm{ppo}_{\mathrm{asm}} \cup \mathrm{eco}) &\implies \mathit{cyclic}(\mathrm{ppo}_{\mathrm{asm}} \cup \mathrm{rf}_e \cup \mathrm{mo} \cup \mathrm{rb}) \\ &\implies \mathit{cyclic}((\mathrm{ppo}_{\mathrm{asm}} \cup \mathrm{mo}_i \cup \mathrm{rb}_i) \cup \mathrm{rf}_e \cup \mathrm{mo}_e \cup \mathrm{rb}_e) \\ &\implies \mathit{cyclic}(\mathrm{ppo} \cup \mathrm{rf}_e \cup \mathrm{mo}_e \cup \mathrm{rb}_e) \end{aligned}$$

(3) Coherence-III. (Immediate by Internal.)

(4) Atomicity.

$$\begin{aligned} \neg\mathit{irreflexive}(\mathrm{rb}; \mathrm{mo}) &\implies \neg\mathit{irreflexive}(\mathrm{rb}_i; \mathrm{mo}_i) \vee \neg\mathit{irreflexive}(\mathrm{rb}_e; \mathrm{mo}_e) \\ &\implies \neg\mathit{irreflexive}(\mathrm{po}) \vee \mathit{cyclic}(\mathrm{ppo} \cup \mathrm{rf}_e \cup \mathrm{mo}_e \cup \mathrm{rb}_e) \end{aligned}$$

(5) SC.

We prove that $\mathrm{psc} \subseteq \mathrm{ob}^+$, therefore $\mathit{cyclic}(\mathrm{psc}) \implies \mathit{cyclic}(\mathrm{ob})$.

$$\mathrm{psc} \triangleq \mathrm{psc}_{\mathrm{fence}} \cup \mathrm{psc}_{\mathrm{base}}$$

$$\begin{aligned} \mathrm{psc}_{\mathrm{fence}} &\triangleq [\mathsf{F}^{\mathsf{sc}}]; (\mathrm{hb} \cup \mathrm{hb}; \mathrm{eco}; \mathrm{hb}); [\mathsf{F}^{\mathsf{sc}}] \\ &= [\mathsf{F}^{\mathsf{sc}}]; \mathrm{hb}; [\mathsf{F}^{\mathsf{sc}}] \cup [\mathsf{F}^{\mathsf{sc}}]; \mathrm{hb}; \mathrm{eco}; \mathrm{hb}; [\mathsf{F}^{\mathsf{sc}}] \end{aligned}$$

$$[\mathsf{F}^{\mathsf{sc}}]; \mathrm{hb}; [\mathsf{F}^{\mathsf{sc}}] \subseteq \underbrace{[\mathsf{F}^{\mathsf{sc}}]; \mathrm{po}}_{\subseteq \mathrm{ppo}}; (\mathrm{rf}_e \cup \mathrm{ppo})^* \subseteq (\mathrm{ppo} \cup \mathrm{rf}_e)^+ \subseteq \mathrm{ob}^+$$

$$\begin{aligned} &[\mathsf{F}^{\mathsf{sc}}]; \mathrm{hb}; \mathrm{eco}; \mathrm{hb}; [\mathsf{F}^{\mathsf{sc}}] \\ &\quad \overbrace{[\mathsf{F}^{\mathsf{sc}}]; \mathrm{po}; (\mathrm{rf}_e; \mathrm{ppo}^?)^*}^{\subseteq \mathrm{ppo}}; \overbrace{\mathrm{eco}}^{\subseteq \mathrm{ob}^+}; \\ &\quad \underbrace{[\mathit{codom}(\mathrm{eco})]; \mathrm{po}_{\mathrm{RC11}}{}^?; ([\mathsf{E}^{\sqsupseteq \mathsf{rel}}]; ([\mathsf{F}]; \mathrm{po})^?; [\mathsf{W}^{\sqsupseteq \mathsf{rlx}}]; (\mathrm{rf}_e; \mathrm{ppo}^?)^*)^?; [\mathsf{F}^{\mathsf{sc}}]}_{\subseteq} \\ &\subseteq \left(\begin{array}{l} [\mathsf{R} \cup \mathsf{RMW}]; \mathrm{po}; (\mathrm{rf}_e; \mathrm{ppo}^?)^* \cup \\ [\mathsf{W}^{\neq \mathsf{nt}}]; \mathrm{po}; [\mathsf{W}^{\neq \mathsf{nt}} \cup \mathsf{F}^{\mathsf{sc}}]; (\mathrm{rf}_e; \mathrm{ppo}^?)^* \cup \\ [\mathsf{W}^{\mathsf{nt}}]; \mathrm{po}|_{\mathrm{loc}}; [\mathsf{E} \setminus \mathsf{R}]; (\mathrm{rf}_e; \mathrm{ppo}^?)^* \cup \\ [\mathsf{W}^{\mathsf{nt}}]; \mathrm{po}; [\mathsf{RMW}^{\mathsf{tso}} \cup \mathsf{F}^{\sqsupseteq \mathsf{sf}}]; \mathrm{po}; [\mathsf{E} \setminus \mathsf{R}]; (\mathrm{rf}_e; \mathrm{ppo}^?)^* \end{array} \right)^? \\ &\subseteq \mathrm{ppo}; (\mathrm{rf}_e; \mathrm{ppo}^?)^*; \mathrm{ob}^+; \mathrm{ppo}^?; (\mathrm{rf}_e; \mathrm{ppo}^?)^* \subseteq \mathrm{ob}^+ \end{aligned}$$

$$\begin{aligned} \mathrm{psc}_{\mathrm{base}} \triangleq\ & ([\mathsf{E}^{\mathsf{sc}}] \cup [\mathsf{F}^{\mathsf{sc}}]; \mathrm{hb}^?); \\ & (\mathrm{po} \cup \mathrm{po}|_{\neq \mathrm{loc}}; \mathrm{hb}; \mathrm{po}|_{\neq \mathrm{loc}} \cup \mathrm{hb}|_{\mathrm{loc}} \cup \mathrm{mo} \cup \mathrm{rb}); \\ & (\mathrm{hb}^?; [\mathsf{F}^{\mathsf{sc}}] \cup [\mathsf{E}^{\mathsf{sc}}]) \end{aligned}$$

$$\begin{aligned} &\subseteq [\mathsf{E}^{\mathsf{sc}}]; (\mathrm{po} \cup \mathrm{rf}_e)^+ && (\subseteq [\mathsf{E}^{\mathsf{sc}}]; \mathrm{po}; (\mathrm{rf}_e; \mathrm{ppo}^?)^* \subseteq \mathrm{ob}^+) \\ &\cup [\mathsf{F}^{\mathsf{sc}}]; \mathrm{hb}; \mathrm{eco}; \mathrm{hb}; [\mathsf{F}^{\mathsf{sc}}] && (\subseteq \mathrm{ob}^+) \\ &\cup [\mathsf{F}^{\mathsf{sc}}]; \mathrm{hb}; (\mathrm{mo} \cup \mathrm{rb}) && (\subseteq [\mathsf{F}^{\mathsf{sc}}]; \mathrm{po}; (\mathrm{rf}_e; \mathrm{ppo}^?)^*; \mathrm{eco} \subseteq \mathrm{ob}^+) \\ &\cup (\mathrm{mo} \cup \mathrm{rb}); \mathrm{hb}; [\mathsf{F}^{\mathsf{sc}}] && (\text{See proof of } [\mathsf{F}^{\mathsf{sc}}]; \mathrm{hb}; \mathrm{eco}; \mathrm{hb}; [\mathsf{F}^{\mathsf{sc}}] \subseteq \mathrm{ob}^+) \\ &\cup (\mathrm{mo} \cup \mathrm{rb}) && (\subseteq (\mathrm{mo}_i \cup \mathrm{rb}_i) \cup \mathrm{mo}_e \cup \mathrm{rb}_e \subseteq \mathrm{ob}) \end{aligned}$$

(6) No-Thin-Air.

$$\begin{aligned} \mathit{cyclic}(\mathrm{po} \cup \mathrm{rf}_e) &\implies \mathit{cyclic}(\underbrace{([\mathit{codom}(\mathrm{rf})]; \mathrm{po})}_{\subseteq \mathrm{ppo}} \cup \mathrm{rf}_e) \\ &\implies \mathit{cyclic}(\mathrm{ppo} \cup \mathrm{rf}_e \cup \mathrm{mo}_e \cup \mathrm{rb}_e) \end{aligned}$$

□

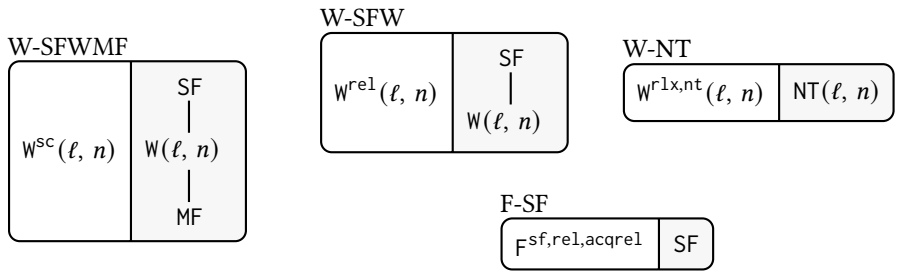


Fig. 17. Selected set of mixed nodes reflecting the alternative compilation scheme from Definition D.2.

D.3.3 Alternative Compilation to Ex86. To apply our methodology of mixed execution graphs to show the correctness of the alternative compilation scheme (Definition D.2), we need to complete the three following preliminary steps:

(1) Define the set of mixed nodes that reflect the alternative compilation scheme.
(2) State RC11$^{\text{Ex86}}$-consistency and Ex86-consistency of mixed execution graphs containing this new set of nodes.
(3) Prove the transfer principle for these new consistency definitions.

Figure 17 depicts a selection of the updated mixed nodes reflecting the alternative compilation scheme. The complete set of nodes is the same as Figure 15 with the following exceptions: (1) nodes of type W-WMF are replaced with W-SFWMF, (2) nodes of type F-⊥ with access mode `rel/acqrel` are replaced with F-SF, and (3) nodes of type W-W with access modes `rlx` and `rel` are respectively replaced with W-NT and W-SFW.

To make the distinction between mixed graphs composed of nodes as defined in Figure 15 and mixed graphs composed of nodes as defined in Figure 17 clear, we call the later *alternative mixed execution graphs*. To state RC11$^{\text{Ex86}}$-consistency for alternative mixed graphs, it suffices to update the notation introduced in Definition D.5:

$$\begin{aligned} \mathsf{W} &\triangleq \text{W-W} \cup \text{W-SFWMF} \cup \text{W-NT} \\ \mathsf{RMW} &\triangleq \text{RMW-RMW-S} \cup \text{RMW-RMW-F} \\ \mathsf{R} &\triangleq \text{R-R} \\ \mathsf{F} &\triangleq \text{F-MF} \cup \text{F-SF} \cup \text{F-}\bot \end{aligned} \qquad \begin{aligned} \mathsf{NT} &\triangleq \text{W-NT} \\ \mathsf{SF} &\triangleq \text{F-SF} \\ \mathsf{MF} &\triangleq \text{F-MF} \end{aligned}$$

The statement of RC11$^{\text{Ex86}}$-consistency thus corresponds to Definition B.8 when we replace the sets and relations to their corresponding mixed-graph versions just introduced. The statement of Ex86-consistency however needs more attention, so we state it in a separate definition:

Definition D.12 (Alternative Mixed Execution Graph - Ex86-Consistency). An alternative mixed execution graph $(G_m, \text{rf}, \text{mo})$ is *Ex86-consistent* if the conditions from Definition B.1 hold when we replace rf, mo, and the sets of nodes with the ones just introduced and the ppo relation with the following one:

$$\begin{aligned} \text{ppo} \triangleq\ & [\mathsf{E} \setminus \text{F-}\bot]; \text{po}; [\mathsf{RMW} \cup \mathsf{MF} \cup \mathsf{SF} \cup \text{W-SFWMF}] \\ \cup\ & [\mathsf{R} \cup \mathsf{RMW} \cup \mathsf{MF} \cup \text{W-SFWMF}]; \text{po}; [\mathsf{E} \setminus \text{F-}\bot] \\ \cup\ & [\mathsf{E} \setminus \text{F-}\bot]; \text{po}; [\text{W-SFW}]; \text{po}^{?}; [\mathsf{E} \setminus \text{F-}\bot] \\ \cup\ & [\mathsf{SF}]; \text{po}; [\mathsf{E} \setminus \text{F-}\bot \setminus \mathsf{R}] \\ \cup\ & [\mathsf{W}^{\neq \mathtt{nt}}]; \text{po}; [\mathsf{W}^{\neq \mathtt{nt}}] \\ \cup\ & [\mathsf{W}]; \text{po}|_{\text{loc}}; [\mathsf{W}] \end{aligned}$$

Finally, we state and prove the corresponding transfer principle for alternative mixed graphs:

Theorem D.13 (Transfer Principle). *Let* $(G_m, \mathsf{rf}, \mathsf{mo})$ *be an alternative mixed execution graph. The* $RC11^{Ex86}$*-consistency conditions hold of* G_m *iff they hold of* $\lfloor G_m$*. Analogously, the Ex86-consistency conditions hold of* G_m *iff they hold of* $G_m\rfloor$*.*

Proof. As in the proof of Theorem D.7, that G_m is $\mathrm{RC11}^{\mathrm{Ex86}}$-consistent iff $\lfloor G_m$ is $\mathrm{RC11}^{\mathrm{Ex86}}$-consistent, is straightforward, because there is a one-to-one correspondence between G_m and $\lfloor G_m$ and because $\mathrm{RC11}^{\mathrm{Ex86}}$-consistency conditions are equivalently defined for both graphs.

We now prove that G_m is Ex86-consistent iff $G_m\rfloor$ is Ex86-consistent. By studying the definition of ppo from Definition D.12, it is easy to see that every $G_m.\mathsf{ppo}$ edge is projected to a $(G_m\rfloor).\mathsf{ppo}^+$ edge. To give an example, edges of type

$$G_m.([\mathsf{E} \setminus \mathsf{F\text{-}}\bot]; \mathsf{po}; [\mathsf{W\text{-}SFW}]; \mathsf{po}^?; [\mathsf{E} \setminus \mathsf{F\text{-}}\bot \setminus \mathsf{R}])$$

are projected to edges of type

$$(G_m\rfloor).(\underbrace{\mathsf{po}; [\mathsf{SF}]}_{\subseteq \mathsf{ppo}}; \underbrace{[\mathsf{SF}]; \mathsf{po}; [\mathsf{W}]; \mathsf{po}^?; [\mathsf{E} \setminus \mathsf{R}]}_{\subseteq \mathsf{ppo}}),$$

who belong to $(G_m\rfloor).\mathsf{ppo}^+$. Therefore, if the Ex86-consistency conditions hold of $G_m\rfloor$ they must hold of G_m.

To show the converse, it suffices to check that every edge of type

$$R \triangleq (G_m\rfloor).([\mathsf{E} \setminus \mathsf{MF} \setminus \mathsf{SF}]; \mathsf{ppo}^+; [\mathsf{E} \setminus \mathsf{MF} \setminus \mathsf{SF}])$$

is the projection of an edge of type $G_m.\mathsf{ppo}^+$. (It is sound to restrict our attention to edges that do not start or end in a fence, because only this type of edge can be used to form cycles that violate External.) Let (a, b) be an edge in R. The proof that (a, b) is the projection of an edge of type $G_m.\mathsf{ppo}^+$ goes by disjunction of cases on whether there is a fence or a read-modify-write event between a and b.

In the negative case, the $(G_m\rfloor).\mathsf{ppo}$ edges between a and b are of type either

$$(G_m\rfloor).([\mathsf{Ex86.W}]; \mathsf{po}; [\mathsf{Ex86.W}]) \quad \text{or} \quad (G_m\rfloor).([\mathsf{Ex86.}(\mathsf{W} \cup \mathsf{NT})]; \mathsf{po}|_{\mathsf{loc}}; [\mathsf{Ex86.}(\mathsf{W} \cup \mathsf{NT})]).$$

It is easy to see that, in this case, events in $dom((G_m\rfloor).([\mathsf{W} \cup \mathsf{NT}]; \mathsf{ppo}; [b]))$ come from the projection of mixed nodes of type either W-W,or W-NT; and that $(G_m\rfloor).\mathsf{ppo}$ edges between these nodes correspond to either

$$G_m.([\mathsf{W}^{\neq \mathsf{nt}}]; \mathsf{po}; [\mathsf{W}^{\neq \mathsf{nt}}]) \quad \text{or} \quad G_m.([\mathsf{W}]; \mathsf{po}|_{\mathsf{loc}}; [\mathsf{W}]),$$

both of which are included in $G_m.\mathsf{ppo}$.

In the affirmative case, there must be at least two $(G_m\rfloor).\mathsf{ppo}$ edges between a and b; then the proof follows by induction on the number n of the remaining $(G_m\rfloor).\mathsf{ppo}$ edges. In the inductive case, we can assume that there is only one fence or read-modify-write event between a and b and that this event is the target of the immediate $(G_m\rfloor).\mathsf{ppo}$ coming out from a, because, otherwise, the edge (a, b) would fit into a smaller number of $(G_m\rfloor).\mathsf{ppo}$ edges and the inductive hypothesis would be applicable. □

Lemma D.14 ($\mathrm{RC11}^{\mathrm{Ex86}}$-Weaker-Than-Ex86-Alt). *Let* G_m *be an alternative mixed execution graph (that is, a mixed graph formed of nodes as specified in Figure 17). If* G_m *is Ex86-consistent, then* G_m *is* $RC11^{Ex86}$*-consistent.*

Proof. The proof is analogous to the proof of Lemma D.11. The main difference is how we show that the po prefix of a hb edge is included in ppo. To give an illustration, we include here the proof that, in the alternative mixed graph G_m, the violation of the condition *irreflexive*(hb; $(\mathsf{mo}_e \cup \mathsf{rb}_e)$)

(ensured by Coherence-I) implies the violation of External. The idea is to exploit the fact that, thanks to the alternative compilation scheme (which is encoded in the structure of G_m), a sw edge always starts with a store fence or a stronger barrier:

$$
\begin{aligned}
&\neg \mathit{irreflexive}(\mathsf{hb};\ (\mathsf{mo}_e \cup \mathsf{rb}_e)) \\
&\implies \neg \mathit{irreflexive}((\mathsf{po}_{\mathsf{RC11}} \cup \mathsf{sw})^+;\ (\mathsf{mo}_e \cup \mathsf{rb}_e)) \implies \\
&\neg \mathit{irreflexive}\left(\begin{array}{l}
\subseteq \left(\begin{array}{l}
[\mathsf{W}^{\mathsf{nt}}];\ \mathsf{po}|_{\mathsf{loc}};\ [\mathsf{W}] \cup \\
[\mathsf{W}];\ \mathsf{po};\ \left[\begin{array}{l}\mathsf{MF} \cup \mathsf{SF} \cup \mathsf{RMW} \cup \\ \text{W-SFW} \cup \\ \text{W-SFWMF}\end{array}\right];\ \mathsf{po}^?;\ [\mathsf{W}]
\end{array}\right)^? \subseteq \mathsf{ppo}^? \\
\overbrace{[\mathit{codom}(\mathsf{mo} \cup \mathsf{rb})];\ \mathsf{po}_{\mathsf{RC11}}{}^?;\ [\mathsf{E}^{\sqsupseteq \mathsf{rel}}];\ (\mathsf{[F]};\ \mathsf{po})^?;\ [\mathsf{W}^{\sqsupseteq \mathsf{rlx}}]}; \\
(\mathsf{rf}_e;\ \mathsf{ppo}^?)^+;\ (\mathsf{mo}_e \cup \mathsf{rb}_e)
\end{array}\right) \\
&\implies \neg \mathit{irreflexive}(\mathsf{ppo}^?;\ (\mathsf{rf}_e;\ \mathsf{ppo}^?)^+;\ (\mathsf{mo}_e \cup \mathsf{rb}_e)) \\
&\implies \mathit{cyclic}(\mathsf{ppo} \cup \mathsf{rf}_e \cup \mathsf{mo}_e \cup \mathsf{rb}_e)
\end{aligned}
$$

□

D.4 Compiler Optimizations

We consider common compiler optimizations and study under which conditions they are sound in the context of $\mathsf{RC11}^{\mathsf{Ex86}}$. Following Lahav et al. [14], we formalize a compiler optimization as a *program transformation*; that is, a mapping that takes and produces programs in the source language, which, in our case, is the language $\mathsf{RC11}^{\mathsf{Ex86}}$-lang. When discussing a given transformation, we use the notation $p \rightsquigarrow p'$ to express that p' can be obtained by applying the transformation to p.

Intuitively, a program transformation is sound under $\mathsf{RC11}^{\mathsf{Ex86}}$ if applying this transformation does not introduce new behaviors. Formally speaking, this means that, if $p \rightsquigarrow p'$ holds, then the set of behaviors of p' is a subset of the set of behaviors of p, that is, $[\![p']\!] \subseteq [\![p]\!]$.

To prove the soundness of a program transformation, we usually resort to its natural generalization to the level of execution graphs: a transformation that applies to events in an execution graph rather than to instructions. In all the transformations considered here, this generalization is straightforward. For example, the *strengthening transformation*, which replaces the mode of a RC11 instruction with a stronger mode (according to the ordering of access modes), can be easily generalized to a graph transformation that replaces the mode of the event emitted by this instruction. As for programs, we use the notation $G \rightsquigarrow G'$ to express that G' can be obtained by applying the transformation to G. The key property, which holds of all transformations here considered, and which allows us to shift our attention to the graph transformation when proving soundness of a program transformation, is that, if $p \rightsquigarrow p'$ and G' is an execution graph associated with p', then there exists an execution graph G associated with p such that $G \rightsquigarrow G'$. Under this property, to show the soundness of the program transformation, it suffices to show that, if G' is $\mathsf{RC11}^{\mathsf{Ex86}}$-consistent, then so is G; and, if G' is racy, then so is G.

D.4.1 Register Promotion. Register promotion replaces accesses to a memory location with accesses to a register, provided that this location is accessed by only one thread and that this location is not accessed via an inline-assembly read-modify-write. At the level of execution graphs, the transformation $G \rightsquigarrow G'$ removes all the accesses to a location x in G, provided that these accesses

are related by G.po and that their intersection with $\mathtt{RMW}^{\mathtt{tso}}$ is empty. Avoiding $\mathtt{RMW}^{\mathtt{tso}}$ is necessary:

$$\begin{array}{l}\textbf{asm}\,\{[x] :=_{\mathtt{nt}} 1\}\\ \textbf{asm}\,\{a := \mathtt{rmw}\,([z],\,0,\,1)\}\;//\;\underline{0}\\ [y]^{\mathtt{rel}} := 1\end{array} \;\Big\|\; \begin{array}{l} b := [y]^{\mathtt{acq}}\;//\;\underline{1}\\ c := [x]^{\mathtt{rlx}}\;//\;\underline{1}\end{array} \quad \rightsquigarrow \quad \begin{array}{l}\textbf{asm}\,\{[x] :=_{\mathtt{nt}} 1\}\\ d := 0\\ a := d\\ d := 1\\ [y]^{\mathtt{rel}} := 1\end{array} \;\Big\|\; \begin{array}{l} b := [y]^{\mathtt{acq}}\;//\;1\\ c := [x]^{\mathtt{rlx}}\;//\;1\end{array}$$

Theorem D.15 (Register Promotion). *The transformation that promotes accesses to a register a location used by only one thread and not via inline-assembly read-modify-writes is sound: for every p and p', if $p \rightsquigarrow p'$, then $[\![p']\!] \subseteq [\![p]\!]$.*

Proof. Let z be the location that is promoted to a register, let j be the identifier of the thread to which this location belongs, and let G' be a execution graph associated with p'. Let G be a graph associated with p, obtained by extending $G'.\mathtt{E}$ with the missing z accesses and by extending $G'.\mathsf{mo}, \mathsf{rf}$ in such a way that $G.\mathsf{mo}_z, \mathsf{rb}_z, \mathsf{rf}_z \subseteq G.\mathrm{po}$ (this is possible because the register instructions in p', to which the accesses to z were promoted, are executed in order). We show that, if G' is $\mathsf{RC11}^{\mathsf{Ex86}}$-consistent, then so is G, and that, if G' is racy, then so is G.

The proof relies on the fact that $[G'.\mathtt{E}]; G.(\mathrm{po}_{\mathsf{RC11}}{}^{+}); [G'.\mathtt{E}]$ is included in $G'.(\mathrm{po}_{\mathsf{RC11}}{}^{+})$. Indeed, let (a, b) be a pair in $G'.\mathtt{E}$, such that $(a, b) \in G.(\mathrm{po}_{\mathsf{RC11}}{}^{+})$. We show that $(a, b) \in G'.(\mathrm{po}_{\mathsf{RC11}}{}^{+})$. If $a \notin \mathtt{W}^{\mathtt{nt}}$, then this is clearly the case. Now, suppose that $a \in \mathtt{W}^{\mathtt{nt}}$. Consequently, for (a, b) to be included in $G.(\mathrm{po}_{\mathsf{RC11}}{}^{+})$, there must be an event $c \in \mathtt{RMW}^{\mathtt{tso}} \cup \mathtt{F}^{\sqsupseteq \mathtt{sf}}$, such that $(a, c), (c, b) \in G.\mathrm{po}_{\mathsf{RC11}}$. Such an event is not an access to z, because z is not accessed through inline-assembly read-modify-writes and because a fence is not an access to z. This event is thus included in $G'.\mathtt{E}$. It is then easy to see that $(a, b) \in G'.(\mathrm{po}_{\mathsf{RC11}}{}^{+})$.

From this fact, it follows that $[G'.\mathtt{E}]; G.\mathsf{hb}; [G'.\mathtt{E}]$ is included in $G'.\mathsf{hb}$.

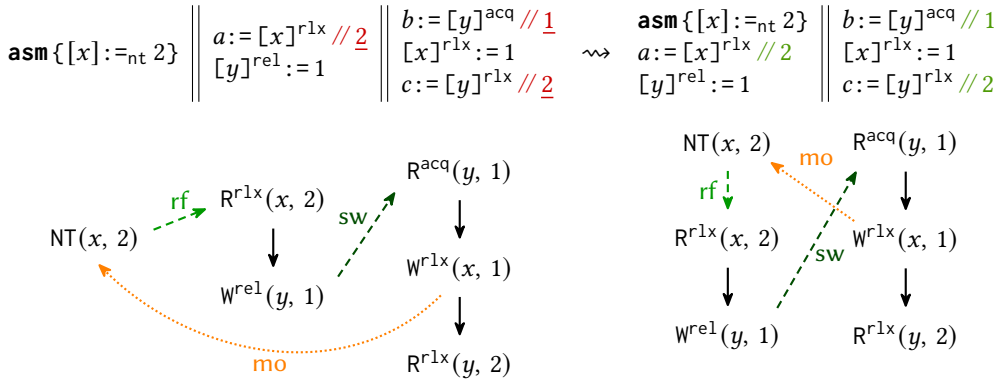
We now prove that, if G' is racy, then so is G. If G' is racy, then there are $a, b \in G'.\mathtt{E}$, such that $(a, b) \in \mathtt{race}(G'.\mathsf{hb})$. To show that $(a, b) \in \mathtt{race}(G.\mathsf{hb})$, it suffices to show that $(a, b) \notin G.\mathsf{hb} \cup G.\mathsf{hb}^{-1}$. Suppose, by contradiction, and without loss of generality, that $(a, b) \in G.\mathsf{hb}$. Then, from the inclusion $[G'.\mathtt{E}]; G.\mathsf{hb}; [G'.\mathtt{E}] \subseteq G'.\mathsf{hb}$, it follows that $(a, b) \in G'.\mathsf{hb}$, a contradiction to $(a, b) \in \mathtt{race}(G'.\mathsf{hb})$.

Now, suppose that G' is $\mathsf{RC11}^{\mathsf{Ex86}}$-consistent. We prove that so is G. Suppose, by contradiction, this is not the case, and let C be a cycle violating $\mathsf{RC11}^{\mathsf{Ex86}}$-consistency in G. This cycle must contain at least one access to z, because, if every event in C is included in $G'.\mathtt{E}$, then C is also a violation to $\mathsf{RC11}^{\mathsf{Ex86}}$ in G'. The cycle is either contained in thread j or it spans over more than one thread. If the cycle spans over more than one thread, then every access to z in C must be surrounded by two accesses to other locations that are distinct from z (but not necessarily between themselves), because z is not shared among threads. The accesses to z in C can thus be avoided, thereby yielding a consistency-violating cycle in G', a contradiction to its consistency. If the cycle is contained in thread j, then there must be two accesses $a, b \in C$, such that $(a, b) \in G.\mathrm{po}$ and $(b, a) \in G.(\mathsf{rf} \cup \mathsf{mo} \cup \mathsf{rb})$. These accesses cannot be to z, because $G.\mathsf{mo}_z, \mathsf{rb}_z, \mathsf{rf}_z \subseteq G.\mathrm{po}$. Therefore, this consistency-violating pair of events also belongs to G', a contradiction to its consistency. □

D.4.2 Strengthening. Strengthening replaces an access mode with a stronger one with respect to the ordering of access modes (Figure 14). Definitions in $\mathsf{RC11}^{\mathsf{Ex86}}$ are *monotonic*; that is, they restrict access modes using ranges of the kind "$\sqsupseteq md$". [4] The correctness of this transformation is thus trivial, because, every edge of the original graph is preserved.

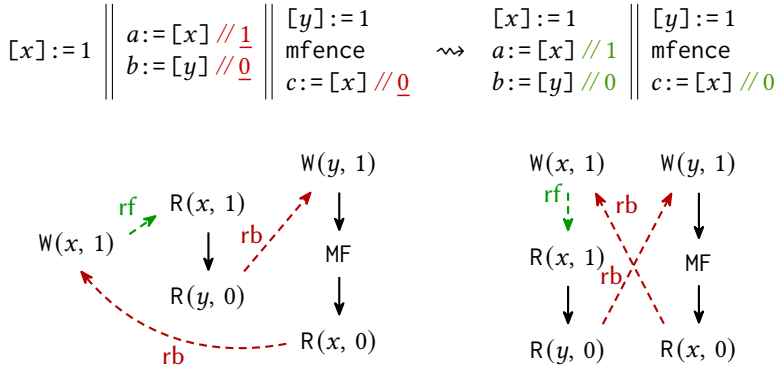
[4] Sets of the form S^{md}, for $md \in \{\mathtt{sc}, \mathtt{tso}\}$, can be rewritten as $S^{\sqsupseteq md}$, and sets of the form $S \setminus \mathtt{W}^{\mathtt{nt}}$ can be rewritten as $(S \setminus \mathtt{W}) \cup (S \cap \mathtt{W}^{\sqsupseteq \mathtt{na}})$.

D.4.3 Sequentialization. Sequentialization merges two threads into one by interleaving their instructions. The following example shows that sequentialization is unsound under $\mathrm{RC11}^{\mathrm{Ex86}}$:



Sequentialization is unsound because, when merging two threads, an external rf edge might become internal. Because internal rf edges are not included in $\mathrm{po}_{\mathrm{RC11}}$, in $\mathrm{ppo}_{\mathrm{asm}}$, or in eco, exchanging a rf_e edge for a rf_i edge might undo cycles in $\mathrm{ppo}_{\mathrm{asm}} \cup \mathrm{eco}$, in hb; $\mathrm{eco}^?$, or in psc.

The omission of rf_i edges from $\mathrm{po}_{\mathrm{RC11}}$, $\mathrm{ppo}_{\mathrm{asm}}$, and eco, is necessary because non-temporal stores break release-acquire synchronization. Moreover, the omission of rf_i in the statement of Coherence-II is inherited from Ex86 which also omits rf_i edges in the statement of External. For this reason, sequentialization is also unsound in plain Ex86:



Because sequentialization is unsound in Ex86, any extension of Ex86 model for C/C++ with inline Ex86-assembly (Theorem C.15) does not support sequentialization. In light of these remarks, we see two possible approaches to conciliate this tension: we can either abandon the property of being an extension of Ex86 or abandon sequentialization. We argue in favor of the first option, to abandon sequentialization in its full generality; and we argue against the second option, which would mean modifying the model so as to validate sequentialization and thereby abandon the property of being an extension of Ex86. We argue against this second option because it would lead either to a more relaxed model in which some rf_e edges are not included in eco, or to a stronger model in which some rf_i edges are included in eco. Both scenarios are unsatisfactory. In the first scenario, one would lose reasoning principles; whereas, in the second scenario, the straightforward identity map would not be a sound compilation scheme for inline assembly. We thus focus on refinements of sequentialization that are valid under $\mathrm{RC11}^{\mathrm{Ex86}}$ as is.

We call a rf_e edge of a $\mathsf{RC11}^{\mathsf{Ex86}}$-inconsistent graph G *problematic* if sequentialization transforms G into a $\mathsf{RC11}^{\mathsf{Ex86}}$-consistent graph. We note that a problematic edge must contain at least one inline-assembly event. Indeed, because $[\mathsf{E} \setminus \mathsf{W}^{\mathsf{nt}}]; \mathsf{rf}_i \subseteq \mathsf{po}_{\mathsf{RC11}}$, edges between standard RC11 events cannot undo cycles in hb; eco. Moreover, edges between standard RC11 events cannot undo cycles in $\mathsf{ppo}_{\mathsf{asm}} \cup \mathsf{eco}$, because, by the definition of $\mathsf{ppo}_{\mathsf{asm}}$, every edge $(a, b) \in G.\mathsf{rf}_e$; $[\mathsf{R}^{\neq \mathsf{tso}}]$ that is part of a cycle in $G.(\mathsf{ppo}_{\mathsf{asm}} \cup \mathsf{eco})$ must be followed by an edge $(b, c) \in \mathsf{po}$; $[\mathsf{RMW}^{\mathsf{tso}} \cup \mathsf{F}^{\sqsupseteq \mathsf{sf}}]$. Therefore, after sequentialization, it is easy to see that (a, c) belongs to the $\mathsf{ppo}_{\mathsf{asm}}$ relation of the transformed graph, and that a cycle in $\mathsf{ppo}_{\mathsf{asm}} \cup \mathsf{eco}$ would still exist.

When we consider two threads, a sufficient purely syntactic condition to rule out the existence of such problematic rf edges in the eventual execution graphs of a program is the following: (1) if one of the threads includes plain RC11 reads then the addresses of all these accesses and the addresses of all locations modified by the other thread using inline assembly should be statically known and disjoint, and (2) if one of the threads includes inline-assembly reads then the addresses of all these accesses and the addresses of all locations modified by the other thread (using inline assembly or not) should be statically known and disjoint. We call this condition *No Interaction Through Inline Assembly* (NITIA). Notice that, thanks to the inclusions $[\mathsf{RMW}]; \mathsf{rf}_i \subseteq \mathsf{po}_{\mathsf{RC11}} \cap \mathsf{ppo}_{\mathsf{asm}}$ and $\mathsf{rf}_i; [\mathsf{RMW}] \subseteq \mathsf{po}_{\mathsf{RC11}} \cap \mathsf{ppo}_{\mathsf{asm}}$, read-modify-writes can be ignored when checking the NITIA condition. Refining the statement of sequentialization to require this condition to hold when merging two threads leads to a sound optimization:

Theorem D.16 (NITIA-Sequentialization). *The transformation that merges two threads that satisfy NITIA is sound: for every p and p', if $p \rightsquigarrow p'$, then $[\![p']\!] \subseteq [\![p]\!]$.*

Proof. Let G' be an execution graph associated with p', and let G be an execution graph associated with p such that $G \rightsquigarrow G'$. We show that, (1) if G' is consistent, then so is G; and that, (2) if G' is racy, then so is G.

The proof of (1) follows by contradiction: suppose that G is $\mathsf{RC11}^{\mathsf{Ex86}}$-inconsistent, but G' is not. Then there exists a problematic edge between the two threads being merged. We saw that such an edge must contain at least one inline-assembly event. This contradicts the NITIA condition, which prevents the existence of rf_e edges between such accesses.

To show (2), it suffices to notice that G.hb is included in G'.hb. Indeed, this inclusion holds because the definition of hb does not distinguish the internal from the external components of rf and because G.po is included in G'.po. □

Another possible refinement of sequentialization is to add a sc fence between the threads to be merged. Inserting such a fence imposes the constraint that the instructions from one of the threads are executed only after the instructions from the other thread have been completed. In comparison to the previous version of sequentialization, when merging two threads that satisfy NITIA, the resulting sequence of instructions can be any interleaving of the instructions from each of the threads.

Theorem D.17 (Fence-Sequentialization). *The transformation that merges two threads by inserting a* sc *fence between them is sound: for every p and p', if $p \rightsquigarrow p'$, then $[\![p']\!] \subseteq [\![p]\!]$.*

Proof. It suffices to notice that an external edge $(a, b) \in G.\mathsf{rf}_e$ becomes part of both $G'.\mathsf{po}_{\mathsf{RC11}}{}^{+}$ and $G'.\mathsf{ppo}_{\mathsf{asm}}{}^{+}$, because a sc fence is inserted between a and b: $(a, b) \in G'.(\mathsf{po}; [\mathsf{F}^{\mathsf{sc}}]; \mathsf{po}) \subseteq G'.(\mathsf{po}_{\mathsf{RC11}}{}^{+} \cap \mathsf{ppo}_{\mathsf{asm}}{}^{+})$. □

D.4.4 Deordering. Deordering transforms sequential composition into parallel composition:

$$s; s' \rightsquigarrow s \parallel s'$$

	$\mathsf{R}_y^{md_2}$	$\mathsf{W}_y^{md_2}$	$\mathsf{RMW}_y^{md_2}$	F^{md_2}
$\mathsf{R}_x^{md_1}$	$md_1 \sqsubseteq \mathtt{rlx}$	$md_1, md_2 \sqsubseteq \mathtt{rlx} \wedge \mathsf{na} \in \{md_1, md_2\}$	$md_1 = \mathsf{na} \wedge md_2 \sqsubseteq \mathsf{acq}$	$md_1 \neq \mathtt{rlx} \wedge md_2 = \mathsf{acq}$
$\mathsf{W}_x^{md_1}$	$\{md_1, md_2\} \neq \{\mathsf{sc}\}$	$md_2 \sqsubseteq \mathtt{rlx}$	$md_2 \sqsubseteq \mathsf{acq}$	$md_2 = \mathsf{acq}$
$\mathsf{RMW}_x^{md_1}$	$md_1 \sqsubseteq \mathtt{rel}$	$md_1 \sqsubseteq \mathtt{rel} \wedge md_2 = \mathsf{na}$	—	$md_1 \sqsupseteq \mathsf{acq} \wedge md_2 = \mathsf{acq}$
F^{md_1}	$md_1 = \mathtt{rel}$	$md_1 = \mathtt{rel} \wedge md_2 \neq \mathtt{rlx}$	$md_1 = \mathtt{rel} \wedge md_2 \sqsupseteq \mathtt{rel}$	$md_1 = \mathtt{rel} \wedge md_2 = \mathsf{acq}$

Fig. 18. Deorderable pairs. Events belong to RC11. The variables x and y denote distinct locations.

$$\mathsf{W}^{md}; \mathsf{W}^{md} \rightsquigarrow \mathsf{W}^{md} \qquad \mathsf{W}^{\mathsf{sc}}; \mathsf{R}^{\mathsf{sc}} \rightsquigarrow \mathsf{W}^{\mathsf{sc}} \qquad \mathsf{RMW}^{md}; \mathsf{RMW}^{md} \rightsquigarrow \mathsf{RMW}^{md}$$
$$\mathsf{R}^{md}; \mathsf{R}^{md} \rightsquigarrow \mathsf{R}^{md} \qquad \mathsf{W}^{md}; \mathsf{R}^{\mathsf{acq}} \rightsquigarrow \mathsf{W}^{md} \qquad \mathsf{F}^{md}; \mathsf{F}^{md} \rightsquigarrow \mathsf{F}^{md}$$

$$\mathsf{W}^{md_1}; \mathsf{RMW}^{md_2} \rightsquigarrow \mathsf{W}^{md_1} \qquad md_1 = max\,(md \mid \mathtt{rlx} \sqsubseteq md \sqsubseteq md_2)$$
$$\mathsf{RMW}^{md_1}; \mathsf{R}^{md_2} \rightsquigarrow \mathsf{RMW}^{md_2} \qquad md_2 = max\,(md \mid \mathtt{rlx} \sqsubseteq md \sqsubseteq md_1)$$

Fig. 19. Mergeable pairs. Events belong to RC11. Events in a pair access the same location.

Because, for simplicity, $\mathsf{RC11}^{\mathsf{Ex86}}$-lang does not support the dynamic allocation of threads, we can only express this optimization as a transformation that converts a single two-instructions thread into two one-instruction threads. At the level of execution graphs, however, we are able to express a more general transformation that removes a po edge between two given events a and b but keeps every other po edge from and to these events:

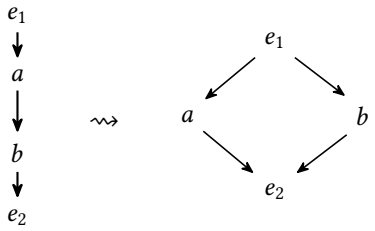


Figure 18 shows the pairs of RC11 events a and b for which this transformation is correct. The Figure corresponds to Table 1 from Lahav et al. [14]. Because events are restricted to plain RC11, and because every $\mathsf{ppo}_{\mathsf{asm}}$ edge contains at least one asm event, this transformation has no effect on $\mathsf{ppo}_{\mathsf{asm}}$. Moreover, no external edge is removed by deordering. Therefore, the restriction of rf to rf_e is not problematic and cycles in $\mathsf{ppo}_{\mathsf{asm}} \cup \mathsf{eco}$ cannot be undone by deordering. In sum, verifying the correctness of deordering is limited to studying the effects of this transformation to the $\mathsf{po}_{\mathsf{RC11}}$ component of hb and to the cycles that violate the standard RC11 conditions; it can thus follow the same arguments as those conveyed by Lahav et al. [14].

D.4.5 Merging. Merging transforms two consecutive instructions into one. Figure 19 depicts the pairs of events that can be merged. These pairs impose constraints on the access modes of the events, but, when performed after strengthening, this transformation can be applied to any pair of instructions whose accesses are at least the ones so specified. The remarks from the previous discussion (about the correctness of deordering) also apply here: roughly speaking, because inline-assembly events are left intact, the correctness of merging follows essentially the same arguments as those presented by Lahav et al. [14].