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 on 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.

CCS Concepts: • **Theory of computation** → **Semantics and reasoning**; **Concurrency**; • **Software and its engineering** → *Formal language definitions*; • **Computer systems organization** → Architectures.

Additional Key Words and Phrases: Concurrency, Weak Memory Models, Semantics of Programming Languages

ACM Reference Format:
Paulo Emílio de Vilhena, Ori Lahav, Viktor Vafeiadis, and Azalea Raad. 2024. Extending the C/C++ Memory Model with Inline Assembly. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 309 (October 2024), 28 pages. https://doi.org/10.1145/3689749

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 prologue and epilogue code of *naked* functions [Microsoft Learn 2021], and (3) to keep the ordering of instructions at compile time [Preshing 2012]. 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) [Linux Kernel Community 2007] and the GNU Compiler Collection (GCC) [GNU Project 1987], each 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 [Batty et al. 2011; Lahav et al. 2017], and despite the

Authors' Contact Information: 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.

ACM 2475-1421/2024/10-ART309
https://doi.org/10.1145/3689749

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* [Flynn 1972] (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* [Intel 2024, Vol. 2B, §4]), whose effect cannot be expressed in the source language. Such instructions are commonly used in libraries for parallel and persistent programming, efficient moving of data, and 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 [Simner et al. 2022; Alglave et al. 2024], and flushing the *translation lookaside buffer* [Intel 2024, Vol. 2A, §3].

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. [2022] 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* [Intel 2024, 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 [Raad et al. 2022], 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` [GNU Project 1987] or `clang` [Clang Project 2007], 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):

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

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]

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

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*, that is, 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.

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:

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

(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 [Batty et al. 2011; Lahav et al. 2017]: 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 [Lahav et al. 2017]: 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 [Leroy 2021, 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 [Vafeiadis et al. 2015; Lahav et al. 2017]. 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 we 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):

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

	P0	P1	P2	P3	P4	P5
Hardware	★	⯪	☆	☆	★	★
Branching	★	⯪	☆	★	★	★
TSO-as-RA	☆	★	★	★	☆	☆
Projection	★	★	★	★	★	☆
Goens et al. [2023]	☆	★	★	★	★	☆
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.

This behavior is allowed by RC11 and observed when the program is run on the POWER [Alglave et al. 2014] 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 [Pulte et al. 2017], 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 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:

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

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. [2023] 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 comes 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 before the read to y (as the latter reads from the former).

The happens-before relation hb, defined as $(\mathsf{po} \cup \mathsf{sw})^+$, 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 (for example, 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 $(\mathsf{po}_{\mathsf{RC11}} \cup \mathsf{sw})^+$, where the relation $\mathsf{po}_{\mathsf{RC11}}$ 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 (that is, 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 (for example, 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 programs 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:

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

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:

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

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

Counterexample IRIW-TSO-2 shows that it is too strong to stipulate the absence of Ex86-consistency-violating cycles that 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 a (ppo ∪ eco)-cycle (that is, in a Ex86-consistency-violating cycle) contain at least one x86 instruction or a sc fence. This is because neither x86 instructions nor sc fences can be optimized 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 [Lahav et al. 2017] 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 the original presentation by Lahav et al. [2017]. We also rely on Podkopaev et al. [2019] for the precise 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

Syntax of expressions, commands, and access modes

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

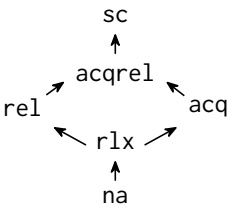


Fig. 2. Syntax of RC11-lang.

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 selection, among this resulting set of execution graphs, of the *consistent* execution graphs. 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. 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 read-modify-writes.

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

We formalize an *event* either as an *initialization event* $I(\ell)$, representing the initialization of ℓ with the default value 0, or as 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

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

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

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

Fig. 3. Definition of pool reduction.

represented as $\mathsf{R}^{md}(\ell, n)$; a write label is represented as $\mathsf{W}^{md}(\ell, n)$; a fence is represented as F^{md}; and a read-modify-write label is represented as $\mathsf{RMW}^{md}(\ell, n, m^?)$, where $m^?$ denotes either a number or the marker $\perp$ representing the case of a failed read-modify-write operation. We are often lax about the distinction between events and labels; we 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 partition RMW into its subset of successful read-modify-writes RMW-s and its subset of 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 of the set of thread pools, *Pool*:

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

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 $\mathit{toPool}(p) \,/\, \mathit{Init} \longrightarrow^* \emptyset \,/\, G$ expresses that G is an execution graph associated with p. The graph *Init* in this statement denotes the *initial execution graph*, a graph where $\mathit{Init}.\mathsf{E}$ is the set of initialization events $I(\ell)$ for every location ℓ, and where $\mathit{Init}.\mathtt{lab}$ maps $I(\ell)$ to $\mathsf{W}^{\mathsf{na}}(\ell, 0)$. The pool $\emptyset$ denotes a thread pool whose domain is empty. The pool $\mathit{toPool}(p)$ denotes a thread pool in its initial state:

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

Figure 3 includes some illustrative cases of the pool reduction relation. The complete definition can be found in the Appendix [de Vilhena et al. 2024, §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 to 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 removed 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 an execution graph G and its extension $(G, \mathsf{rf}, \mathsf{mo})$.

Notation. The metavariables a, b, c, d, and e range over events. An event, as we recall, is formalized as either an initialization event, $I(\ell)$, or as a pair 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 the second projections of a in the case where 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 of events. 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 $(a, b) \in \mathsf{rf}$. 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* (sw), *extended coherence order* (eco), *reads-before* (rb), *partial-SC* (psc), and *SC-before* (scb) are defined as follows:

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

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

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

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

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

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

These consistency conditions are equivalent to the ones formulated by Margalit and Lahav [2021], who diverge from Lahav et al. [2017] 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 documentation of the C++ programming language [Cppreference Community 2019]. 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 *data race* and of *undefined behavior* (*UB*):

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) $\{a, b\} \cap (\mathtt{W} \cup \mathtt{RMW\text{-}s}) \neq \emptyset$, and (4) $(a, b) \notin \mathsf{hb} \cup \mathsf{hb}^{-1}$.

Definition 3.3 (RC11-Behaviors).

$$\left(\mathit{toPool}(p) \;/\; \mathit{Init} \longrightarrow^{*} \emptyset \;/\; G \;\wedge\; (G, \mathsf{rf}, \mathsf{mo})\ \mathit{is\ RC11\text{-}consistent}\right) \vdash p \longrightarrow (G, \mathsf{rf}, \mathsf{mo})$$

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

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

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

Figure 4 shows the syntactical increments of RC11$^{\text{Ex86}}$-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*.

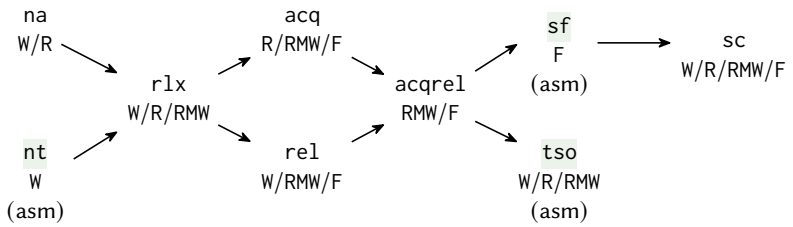


Fig. 5. Diagram of access modes of RC11$^{\text{Ex86}}$.

To give an intuitive operational account of these instructions, we can rely on the formal operational model of Ex86 [Raad et al. 2022]. In this operational model, every thread contains a local buffer 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 buffer, if the buffer 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 other words, in our proposed model, programmers have no good reason to use **`asm`**`{mfence}`, as they can equivalently use $\mathtt{fence}_{\mathtt{sc}}$; we include **`asm`**`{mfence}` only for comprehensiveness.)

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

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

hold, where the relations hb, eco, $\mathrm{po}_{\mathrm{RC11}}$, and $\mathrm{ppo}_{\mathrm{asm}}$ are defined as follows:

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

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

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

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 $\mathtt{tso} \sqsubset \mathtt{sc}$ or perhaps even $\mathtt{na} \sqsubset \mathtt{nt}$. Given that non-temporal stores break release-acquire synchronization, as we shall explain, it is not difficult to understand the absence of the ordering $\mathtt{na} \sqsubset \mathtt{nt}$. Perhaps more striking is the absence of the ordering $\mathtt{tso} \sqsubset \mathtt{sc}$. 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, a tso read-modify-write, or a write to the same location. 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 $\mathsf{RC11}^{\mathsf{Ex86}}$ 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 $\mathsf{RC11}^{\mathsf{Ex86}}$, 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 $(a, a) \in [\mathsf{W}^{\mathsf{nt}}]; \mathsf{po}; (\mathsf{rb} \cup \mathsf{mo})$ 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:

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

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:

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

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

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

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 (for example, 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 $\mathtt{po} \cup \mathtt{rf}$. In our setting, the compiler is not expected to reorder inline assembly, and our compilation schemes are only to Ex86, which enforces this acyclicity condition.

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 $\mathtt{tso} \sqsubset \mathtt{sc}$ however is intriguing, because sequential consistency is stronger than *total store*

order [Sindhu et al. 1992]. 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 [Lahav et al. 2017]) shows that the semantics assigned to sc accesses by RC11 can be weaker than the semantics assigned to tso accesses by our model:

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

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

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 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. [2022], includes two consistency conditions: an *internal* condition, which applies to cycles confined within single threads; and an *external* condition, which posits the absence of certain cycles spanning over multiple threads.

The internal condition in Ex86 posits the irreflexivity of po; $(\mathsf{rf}_i \cup \mathsf{mo}_i \cup \mathsf{rb}_i)$. This condition 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 this definition here, putting it side-by-side with Coherence-II:

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

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

(Ex86 - External)

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

We keep the notation used by Raad et al. [2022] 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 regular writes (W).

[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.

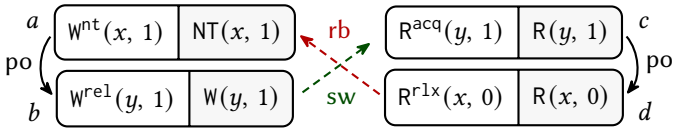
The side-by-side comparison reinforces the claim that Coherence-II integrates Ex86-consistency into $\text{RC11}^{\text{Ex86}}$ under the condition that pairs of po-separated events in a violating cycle include at least one inline-assembly event. Indeed, most cases of ppo_{asm} edges either start or 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 differences between the statements of the acyclicity conditions: Coherence-II uses eco, which includes the internal edges mo_i and rb_i; whereas External uses $\text{rf}_e \cup \text{mo}_e \cup \text{rb}_e$, thereby including only external edges. The inclusion of $[\mathtt{W} \cup \mathtt{NT}]; \text{po}|_{\text{loc}}; [\mathtt{W} \cup \mathtt{NT}]$ edges in the definition of ppo compensates for the absence of mo_i, whereas the inclusion of $[\mathtt{R}]; \text{po}$ edges compensates for the absence of rb_i. This explanation also justifies why, in the statement of Coherence-II, we can omit the "per-location" case in the definition of ppo_{asm}, and reuse eco. The attentive reader might notice that the internal edges in eco 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 (1) no optimization applies to pairs of a read and a write to the same location, so rb_i edges cannot be undone; and (2) mo_i edges between plain RC11 accesses in a $\text{ppo}_{\text{asm}} \cup \text{eco}$ cycle can always be merged into an edge of type mo_e, rb_e, or ppo_{asm}.

3.2.3 Definition of hb. To see why hb is defined using po_{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 (that is, 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 po_{RC11} in the definition of hb. If, however, hb was defined as in RC11, that is, $\text{hb}_{\text{RC11}} = (\text{po} \cup \text{sw})^+$, then the state σ would be 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:



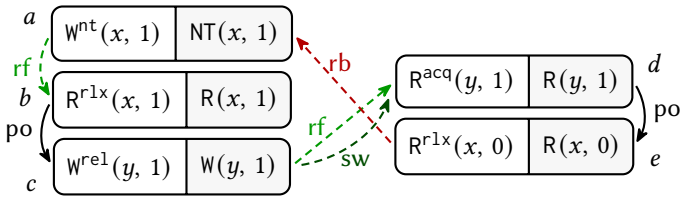
Nodes in this graph carry pairs of a $\text{RC11}^{\text{Ex86}}$ event, issued by the source program, and a Ex86 event, issued by the compiled program. Using this structure, we are able to make several observations:

(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:

$$\begin{array}{l} \mathbf{asm}\,\{[x] :=_{\mathsf{nt}} 1\} \\ a := [x]^{\mathsf{rlx}}\ //\ 1 \\ [y]^{\mathsf{rel}} := 1 \end{array} \;\Big\|\; \begin{array}{l} b := [y]^{\mathsf{acq}}\ //\ 1 \\ c := [x]^{\mathsf{rlx}}\ //\ 0 \end{array}$$

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

4 Metatheory

In this section, we study properties of $\mathsf{RC11}^{\mathsf{Ex86}}$. 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 $\mathsf{RC11}^{\mathsf{Ex86}}$ preserves this property.

The discussion is organized as follows. In §4.1, we define two compilation schemes to Ex86. In §4.2, we introduce the notion of mixed execution graphs, a key concept in our proofs of compilation correctness, whose sketch we present in §4.3. 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 $\mathsf{RC11}^{\mathsf{Ex86}}$ is an extension of RC11 and Ex86 is in the Appendix [de Vilhena et al. 2024, Theorems C.13 and C.14].

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 RC11^{Ex86}-lang to Ex86-lang).

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

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

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

Definition 4.1 follows largely the scheme from Lahav et al. [2017]. 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 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 $[\![\ \_\ ]\!]_{\mathrm{RC11}}$, which defines the semantics of RC11 programs, and to how we implicitly constructed $[\![\ \_\ ]\!]_{\mathrm{RC11}^{\mathrm{Ex86}}}$, we can introduce the function $[\![\ \_\ ]\!]_{\mathrm{Ex86}}$ defining the semantics of Ex86-lang programs. The definition is in the Appendix [de Vilhena et al. 2024, 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

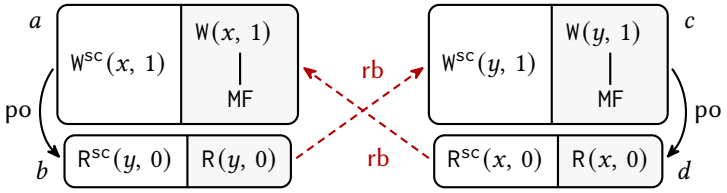


Fig. 6. Example of a mixed execution graph.

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.

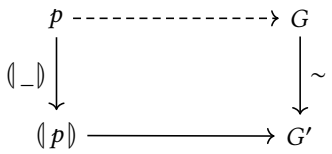
To account for these mismatches, nodes in a mixed graph, called *mixed nodes*, carry events from both source and target models. Events from the two models however cannot be arbitrarily assembled in a mixed node: the source-level events in a mixed node correspond to the events of a single source instruction 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 the example depicted in Figure 6. We refer the reader to the Appendix [de Vilhena et al. 2024, §D.2] for a complete exposition of mixed execution 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 is disallowed in G because all access modes are sc. In other words, we wish to argue that G is inconsistent. 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 $\langle\!| \_ |\!\rangle$ is correct.

This first step consists of showing that, for every program p, for every execution graph G' associated with $\langle\!| p |\!\rangle$, there exists a graph G associated with p, such that G *is simulated by* G' [de Vilhena et al. 2024, 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 $\langle\!| p |\!\rangle$ preserves much of the structure of p, it is possible to replay the pool-reduction steps from $\langle\!| p |\!\rangle$ 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 [de Vilhena et al. 2024, 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* [de Vilhena et al. 2024, 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

We now study the compiler optimizations discussed by Lahav et al. [2017]. We wish to determine 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. [2017], we formalize a compiler optimization as a *program transformation*: 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.

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. 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 the following: if $p \rightsquigarrow p'$, and if 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 (1) that, if G' is $\mathsf{RC11}^{\mathsf{Ex86}}$-consistent, then so is G; and (2) that, 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 $G.\mathsf{po}$ and that their intersection with $\mathsf{RMW}^{\mathsf{tso}}$ is empty. Avoiding $\mathsf{RMW}^{\mathsf{tso}}$ is necessary,

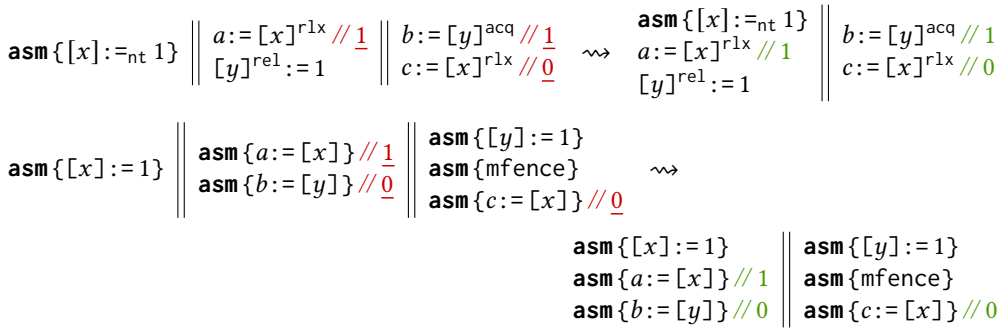


Fig. 7. Counterexamples showing the unsoundness of sequentialization in RC11$^{\text{Ex86}}$.

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*: only upward-closed ranges of the form "$\sqsupseteq md$" occur. [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: $s\,;\,s' \rightsquigarrow s \parallel s'$. Merging transforms two consecutive instructions into one: $s\,;\,s' \rightsquigarrow s''$. Lahav et al. [2017, 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. Figure 7 depicts two counterexamples showing the unsoundness of sequentialization in RC11$^{\text{Ex86}}$.

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

The omission of rf_i edges from po_{RC11}, ppo_{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 [Kang et al. 2017].

Because sequentialization is unsound in Ex86, its support is incompatible with Property P4. If we ignore Property P4, then there are two approaches to add support for sequentialization: (1) to relax the model so as to allow the behavior of the programs on the left-hand side of Figure 7, or (2) to make the model stronger than, or incomparable to, RC11$^{\text{Ex86}}$ so as to disallow the behavior of the programs on the right-hand side of Figure 7. The first approach leads to a lost of reasoning

[3] 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 $(S \setminus \mathtt{W}) \cup (S \cap \mathtt{W}^{\sqsupseteq \mathtt{na}})$.

principles, whereas the second approach invalidates the straightforward identity map as a sound compilation scheme for inline assembly (Property P1). Therefore, instead of aiming to support sequentialization for the price of abandoning Property P4, we investigate conditions under which sequentialization is sound in $\text{RC11}^{\text{Ex86}}$ as is.

We call a rf_e edge of a $\text{RC11}^{\text{Ex86}}$-inconsistent graph G *problematic* if sequentialization transforms G into a $\text{RC11}^{\text{Ex86}}$-consistent graph G'. We note that a problematic edge must contain at least one inline-assembly event. Indeed, because $[\text{E} \setminus \text{W}^{\text{nt}}]$; rf_i is included in po_{RC11}, transforming a $G.\text{rf}_e$ edge between plain RC11 events into $G'.\text{rf}_i$ makes this edge part of $G'.\text{hb}$, so it cannot undo a cycle in hb; eco. Such a transformation cannot undo a cycle in $\text{ppo}_{\text{asm}} \cup \text{eco}$ either, because, by definition of ppo_{asm}, every edge $(a, b) \in G.\text{rf}_e$; $[\text{R}^{\neq\text{tso}}]$ that is part of a cycle in $\text{ppo}_{\text{asm}} \cup \text{eco}$ must be followed by an edge $(b, c) \in \text{po}$; $[\text{RMW}^{\text{tso}} \cup \text{F}^{\sqsupseteq\text{sf}}]$, therefore $(a, c) \in G'.\text{ppo}_{\text{asm}}$.

When we consider two threads, a sufficient purely syntactic condition to rule out the existence of such problematic rf edges 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 $[\text{RMW}]$; $\text{rf}_i \subseteq \text{po}_{\text{RC11}} \cap \text{ppo}_{\text{asm}}$ and rf_i; $[\text{RMW}] \subseteq \text{po}_{\text{RC11}} \cap \text{ppo}_{\text{asm}}$, 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 [de Vilhena et al. 2024, 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 retrospect, with the NITIA-refinement of sequentialization, threads can be arbitrarily interleaved. We prove soundness of this second version of sequentialization in the Appendix [de Vilhena et al. 2024, 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:

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

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

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

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

A detailed proof of this theorem can be found in the Appendix [de Vilhena et al. 2024, §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. [2010] 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) [Sindhu et al. 1992]. 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 [1999]. In this paper, we rely on Raad et al. [2022]'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 [de Vilhena et al. 2024, Definition B.1].

Models of C++. Batty et al. [2011] introduce the first formal memory model of C++ as a formalization of the C++ standard [ISO 2011] mechanized in Isabelle/HOL [Nipkow et al. 2002]. Lahav et al. [2017] however identify several issues with this model. They introduce RC11 (for *Repaired C11*) in an attempt to repair these flaws. Indeed, Lahav et al. [2017] identify at least four problems with the original model of Batty et al. [2011]: (1) the proposed compilation schemes [Batty et al. 2012; Sarkar et al. 2012] 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* [Vafeiadis et al. 2015]. 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 which strengthens the semantics of `sc` fences; and with 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.

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. [2023] 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* [Floyd 1967], which is motivated by situations 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. [2023] 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. [2023] 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. [2017] prove the correctness of compilation schemes from RC11 to several architectures (x86, POWER, and Armv7). Podkopaev et al. [2019] introduce 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 the situation where one has to establish the correctness of compilation schemes of a language to N architectures, instead of proving N results, one could instead prove correctness of a mapping from this language to IMM (assuming that the mappings exists). Such a proof would still be a proof of compilation correctness (from the 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. [2023] 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. At the start of our project, the tool was 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. The tool has since then been extended with support for introducing user-defined sets of events. However, at the time of writing, this feature lacks a comprehensive documentation and the tool lacks a specification of the facts that it takes as assumptions.

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.

Acknowledgments

Paulo Emílio de Vilhena is supported by the UKRI Future Leaders Fellowship MR/V024299/1. Ori Lahav is supported by the European Research Council under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 851811) and by the Israel Science Foundation (grant No. 814/22). Viktor Vafeiadis is supported by the European Research Council under the European Union's Horizon 2020 research and by innovation programme (grant agreement No. 101003349). Azalea Raad is supported by the UKRI Future Leaders Fellowship MR/V024299/1, by the EPSRC grant EP/X037029/1, and by VeTSS.

References

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

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

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

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

Clang Project. 2007. Clang: a C language family frontend for LLVM. https://clang.llvm.org/

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

Paulo Emílio de Vilhena, Ori Lahav, Viktor Vafeiadis, and Azalea Raad. 2024. Extending the C/C++ Memory Model with Inline Assembly – Technical Appendix. https://doi.org/10.5281/zenodo.13625916

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

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

GNU Project. 1987. GNU Compiler Collection. https://gcc.gnu.org/git/gcc.git

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

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.

ISO. 2011. *ISO International Standard ISO/IEC 14882:2011(E) – Programming Language C++*. International Organization for Standardization (ISO). https://www.iso.org/standard/50372.html

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

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

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

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

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

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

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

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

Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. 2002. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Lecture Notes in Computer Science, Vol. 2283. Springer. https://doi.org/10.1007/3-540-45949-9

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

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

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

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

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

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

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

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

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

Viktor Vafeiadis, Thibault Balabonski, Soham Chakraborty, Robin Morisset, and Francesco 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)*. ACM Press, 209–220. https://dl.acm.org/doi/10.1145/2676726.2676995

Received 2024-04-05; accepted 2024-08-18