Towards Certified Compositional Compilation for Concurrent Programs

Anonymous Author(s)

Abstract

Certified compositional compilation is important for establishing end-to-end guarantees for certified systems consisting of separately compiled modules. In this paper, we propose a language-independent framework consisting of the key semantics components and verification steps that bridge the gap between the compilers for sequential programs and for (race-free) concurrent ones, so that the efforts on certified sequential compilation can be reused for concurrent programs. One of the key contributions of the framework is a novel footprint-preserving compositional simulation as the compilation correctness criterion. The framework also provides a new mechanism to introduce confined benign races in the target code, which allows us to support efficient implementation of synchronization primitives. With our framework, we have verified the correctness of CompCert (including all the translation passes from Clight to x86, and two optimization passes) for compositional compilation of race-free concurrent programs.

1 Introduction

Compositional compilation is important for real-world systems, which usually consist of multiple program modules that need to be compiled independently. Correct compilation then needs to guarantee that the target modules can work together and preserve the semantics of the source program as a whole. It requires not only that individual modules be compiled correctly, but also that the expected interaction between modules be preserved at the target.

CompCert [9], the most well-known certified realistic compiler, establishes the semantics preservation property for compilation of *sequential* Clight programs, but with no explicit support of separate compilation. To support general separate compilation, Stewart et al. [20] develop Compositional CompCert, which allows the modules to call each other's external functions. Like CompCert, Compositional CompCert only supports sequential programs too.

Stewart et al. [20] do argue that Compositional CompCert may be applied for data-race-free (DRF) concurrent programs, since they behave the same in the standard interleaving semantics as in certain non-preemptive semantics where switches between threads occur only at certain designated program points. The sequential compilation is sound as long as the switch points are viewed as external function calls so that optimizations do not go beyond them.

Although the argument is plausible, there are still significant challenges to actually implement it. We need proper formulation of the non-preemptive semantics and the notion of DRF. Then we need to indeed prove that DRF programs have the same behaviors (including termination) in the interleaving semantics as in the non-preemptive semantics, and verify that the compilation preserves DRF. More importantly, the formulation and the proofs should be done in a compositional and language-independent way, to allow compositional compilation where the modules can be implemented in different languages. Reusing the proofs of CompCert is challenging too, as memory allocation for multiple threads may require a different memory model from that of CompCert, and the non-deterministic semantics also makes it difficult to reuse the downward simulation in CompCert.

In addition, the requirement of DRF could be sometimes overly restrictive. Although Boehm [3] points out there are essentially *no* "benign" races in source programs, and languages like C/C++ essentially give no semantics to racy programs, it is still meaningful to allow benign races in machine language code for better performance (e.g., the spin locks in Linux). Also machine languages commonly allow relaxed behaviors. Can we support realistic target code with potential benign races in relaxed machine models?

There has been work on verified compilation for concurrent programs [13, 18], but with only limited support of compositionality. We explain the major challenges in detail in Sec. 2, and discuss more related work in Sec. 8.

In this paper, we propose a language-independent framework consisting of the key semantics components and verification steps that bridge the gap between compilation for sequential programs and for DRF concurrent programs. We apply our framework to verify the correctness of CompCert for compositional compilation of DRF programs to x86 assembly. We also extend the framework to allow a limited form of benign races in x86-TSO (which we call *confined benign races*), so that optimized implementation of synchronization primitives like spin-locks in Linux can be supported. Our work is based on previous work on certified compilation, but makes the following new contributions:

- We design a compositional *footprint-preserving simulation* as the correctness formulation of separate compilation for sequential modules (Sec. 4). It considers module interactions at both external function calls and synchronization points, thus is compositional with respect to both module

PL'18, January 01–03, 2018, New York, NY, USA
2018.

linking and non-preemptive concurrency. It also requires the footprints (i.e., the set of memory locations accessed during the execution) of the source and target modules to be related. This way we effectively reduce the proof of the compiler's preservation of DRF, a whole program property, to the proof of *local* footprint preservation.

- We work with an *abstract* programming language (Sec. 3), which is not tied to any specific synchronization constructs such as locks but uses abstract labels to model how such constructs interact with other modules. It also abstracts away the concrete primitives that access memory. We introduce the notion of *well-defined languages* to enforce a set of constraints over the state transitions and the related footprints, which actually give an extensional interpretation of footprints. These constraints are satisfied by various real languages such as Clight and x86 assembly. With the abstract language, we study the equivalence between preemptive and non-preemptive semantics (Sec. 3.3), the equivalence between DRF and NPDRF (the notion of race-freedom defined in the non-preemptive setting, shown in Sec. 5), and the properties of our new simulation. As a result, the lemmas in our proof framework are re-usable when instantiating to real languages.
- We prove a strengthened DRF-guarantee theorem for the x86-TSO model (Sec. 7.3). It allows an x86-TSO program to call a (x86-TSO) module with benign races. If one can replace the racy module with a more abstract version so that the resulting program is DRF in the sequentially consistent (SC) semantics, then the original racy x86-TSO program would behave the same with the more abstract program in the SC semantics. This way we allow the target programs to have confined benign races in the relaxed x86-TSO model. We use this approach to support efficient x86-TSO implementations of locks.
- Putting all these together, our framework (see Fig. 2 and Fig. 3) is the first to build certified compositional compilation for concurrent programs *from sequential compilation*. It highlights the importance of DRF preservation for correct compilation. It also shows a possible way to adapt the existing work of CompCert and Compositional CompCert to interleaving concurrency.
- As an instantiation of our language-independent compilation verification framework, we instantiate the source and target languages as Clight and x86 assembly (Sec. 7). We also provide an efficient x86-TSO implementation of locks as a synchronization library. We have successfully proved that CompCert-3.0.1 [21] (including all the translation passes and two optimization passes) satisfy our compilation correctness criterion. In the verification of the CompCert compilation passes, we reuse a considerable amount of the original CompCert proofs, with minor adjustment for footprint-preservation. The proofs for each pass take less than one person week on average.

2 Informal Development

Below we first give an overview of the main ideas in CompCert [9, 10] and Compositional CompCert [20] as starting points for our work. Then we explain the challenges and our ideas in reusing them to build certified compositional compilation for concurrent programs.

2.1 CompCert

Figure 1(a) shows the key proof structure of CompCert. The compilation $Comp$ is correct, if for every source program S, the compiled code C preserves the semantics of S. That is, $\text{Correct}(Comp) \stackrel{\text{def}}{=} \forall S, C.\ Comp(S) = C \implies S \approx C$. The semantics preservation $S \approx C$ requires S and C have the same sets of observable event traces:

$$S \approx C \text{ iff } \forall \mathcal{B}.\ Etr(S, \mathcal{B}) \iff Etr(C, \mathcal{B})\ .$$

Here we write $Etr(S, \mathcal{B})$ to mean that an execution of S produces the observable event trace $\mathcal{B}$, and likewise for C.

To verify $S \approx C$, CompCert relies on the determinism of the target language (written as $\text{Det}(C)$ in Fig. 1(a)) and proves only the downward direction $S \sqsubseteq C$, i.e., S refines C:

$$S \sqsubseteq C \text{ iff } \forall \mathcal{B}.\ Etr(S, \mathcal{B}) \implies Etr(C, \mathcal{B})\ .$$

The determinism $\text{Det}(C)$ ensures that C admits only one event trace, so we can derive the upward refinement $S \sqsupseteq C$ from $S \sqsubseteq C$. The latter is then proved by constructing a (downward) simulation relation $S \precsim C$, depicted in Fig. 1(c). However, the simulation $\precsim$ relates whole programs only and it does not take into account the interactions with other modules which may update the shared resource. So it is not compositional and does *not* support separate compilation.

2.2 Compositional CompCert

Compositional CompCert supports separate compilation by re-defining the simulation relation for modules. Figure 1(b) shows its proof structure. Suppose we make separate compilation $Comp_1, \ldots, Comp_n$. Each $Comp_i$ transforms a source module S_i to a target module C_i. The overall compilation is correct if, when linked together, the target modules $C_1 \circ \ldots \circ C_n$ preserve the semantics of the source modules $S_1 \circ \ldots \circ S_n$ (here we write $\circ$ as the module-linking operator). For example, the following program consists of two modules. The function `f` in module `S1` calls the external function `g`. The external module `S2` may access the variable `b` in `S1`.

```
// Module S1
extern void g(int *x);
int f(){
  int a = 0, b = 0;
  g(&b);
  return a + b; }
```

```
// Module S2
int g(int *x){
  *x = 3;
}
```

(2.1)

Suppose the two modules `S1` and `S2` are independently compiled to the target modules `C1` and `C2`. The correctness of the overall compilation requires $(\texttt{S1} \circ \texttt{S2}) \approx (\texttt{C1} \circ \texttt{C2})$.

With the determinism of the target modules, we only need to prove the downward refinement $(\texttt{S1} \circ \texttt{S2}) \sqsubseteq (\texttt{C1} \circ \texttt{C2})$, which is reduced to proving $(\texttt{S1} \circ \texttt{S2}) \precsim (\texttt{C1} \circ \texttt{C2})$, just as

in CompCert. Ideally we hope to know (S1∘S2) $\precsim$ (C1∘C2) from S1 $\precsim$ C1 and S2 $\precsim$ C2, and ensure the latter two by correctness of $Comp_1, \ldots, Comp_n$. However, the CompCert simulation $\precsim$ is not compositional.

To achieve compositionality, Compositional CompCert defines the simulation relation $\precsim'$ shown in Fig 1(d). It is parameterized with the interactions between modules, formulated as the rely/guarantee conditions R and G [7]. The rely condition R of a module specifies the general callee behaviors happen at the *external function calls* of the current module (shown as the thick arrows in Fig. 1(d)). The guarantee G specifies the possible transitions made by the module itself (the thin arrows). The simulation is compositional as long as the R and G of linked modules are compatible.

Compositional CompCert proves that the CompCert compilation passes satisfy the new simulation $\precsim'$. The intuition is that the compiler optimizations do not go beyond external calls (unless only local variables get involved). That is, for the above example (2.1), the compiler cannot do optimizations based on the (wrong) assumption that b is 0 when f returns.

Since the R steps happen only at the external calls, it cannot be applied to concurrent programs, where module/thread interactions may occur at any program point. However, if we consider race-free concurrent programs only, where threads are properly synchronized, we may consider the interleaving at certain synchronization points only. It has been a folklore theorem that DRF programs in interleaving semantics behave the same as in non-preemptive semantics. For instance, the following program (2.2) uses a lock to synchronize the accesses of the shared variables, and it is race-free. Its behaviors are the same as those when the threads yield controls at `lock()` and `unlock()` only. That is, we can view lines 1–2 and lines 4–5 in either thread as sequential code that will not be interfered by the other. The interactions occur only at the boundaries of the critical regions.

```
1  r1 = 1;          ||  r2 = 2;
2  r1 = r1 + 1;     ||  r2 = r2 + 1;
3  lock();          ||  lock();
4    x = 1;         ||    x = 2;
5    y = x + 1;     ||    y = x + 1;
6  unlock();        ||  unlock();
```
(2.2)

Intuitively, we can use Compositional CompCert to compile the program, where the code segment between two consecutive switch points is compiled as sequential code. By viewing the switch points as special external function calls, the simulation $\precsim'$ can be applied to relate the non-preemptive executions of the source and target modules.

2.3 Challenges and our approaches

Although the idea of applying Compositional CompCert to compile DRF programs is plausible, we have to address several key challenges to really implement it.

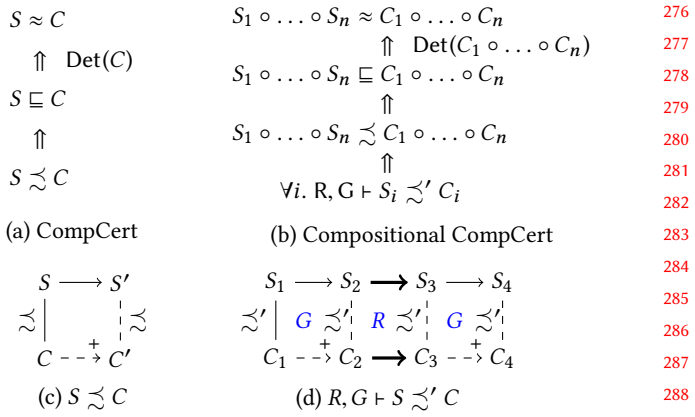


Figure 1. Proof structures of certified compilation

Q: How to give language-independent formulation of DRF and non-preemptive semantics? Compositional CompCert introduces interaction semantics, which describes module interactions without referring to the concrete languages used to implement the modules. This allows composition of modules implemented in different languages. We would like to follow the semantics framework, but how do we define DRF and non-preemptive semantics if we do not even know the concrete synchronization constructs and the commands that access memory?

A: We extend the module-local semantics in Compositional CompCert so that each local step of a module reports its footprints, i.e. the memory locations it accesses. Instead of relying on the concrete memory-access commands to define what valid footprints are, we introduce the notion of *well-defined languages* (in Sec. 3) to specify the requirements over the state transitions and the related footprints. For instance, we require the behavior of each step is affected by the read set only, and each step does *not* touch the memory outside of the write set. When we instantiate the framework with real languages, we prove they satisfy these requirements.

Besides, we also allow module-local steps to generate messages EntAtom and ExtAtom to indicate the boundary of the atomic operations inside the module. The concrete commands that generate these messages are unspecified, which can be different in different modules.

Q: What memory model to use in the proofs? The choice of memory models could greatly affect the complexity of proofs. For instance, using the same memory model as CompCert allows us to reuse CompCert proofs, but it also causes many problems. The CompCert memory model records the next available block number `nextblock` for memory allocation. Using the model under a concurrent setting may ask all threads to share one `nextblock`. Then allocation in one thread would affect the subsequent allocations in other threads. This breaks the property that re-ordering non-conflicting operations from different threads would *not* affect the final states, which is a key lemma we rely on to prove

the equivalence between preemptive and non-preemptive semantics for DRF programs. In addition, sharing the `nextblock` by all threads also means we have to keep track of the ownership of each allocated block when we reason about footprints.

A: We decide to use a different memory model. We reserve separate address spaces F for memory allocation in different threads (see Sec. 3). Therefore allocation of one thread would not affect behaviors of others. This greatly simplifies the semantics and the proofs, but also makes it almost impossible to reuse CompCert proofs (see Sec. 7.2). We address this problem by establishing some semantics equivalence between our memory model and the CompCert memory model (shown in Sec. 7.2).

Q: How to compositionally prove DRF-preservation? The simulation $\precsim'$ in Compositional CompCert cannot ensure DRF-preservation. As we have explained, DRF is a whole-program property, and so is DRF-preservation. To support separate compilation, we need to reduce DRF-preservation to some thread-local property.

A: We propose a new compositional simulation $\preccurlyeq$ (see Sec. 4). Based on the simulation in Fig. 1(d), we additionally require *footprint consistency* saying that the target C should have the same or smaller footprints than the source S during related transitions. For instance, when compiling lines 4–5 of the left thread in (2.2), the target is only allowed to read x and write to x and y. Note that we check footprint consistency at switch points only. This way we allow compiler optimizations as long as they do not go beyond the switch points. For lines 4–5 of the left thread in (2.2), the target could be `y=2;x=1` where the writes to x and y are swapped.

Q: Can we flip refinement/simulation in spite of non-determinism? As we explained, the last steps of CompCert and Compositional CompCert in Fig. 1 derive semantics equivalence $\approx$ (or the upward refinement $\sqsupseteq$) from the downward refinement $\sqsubseteq$ using determinism of target programs. Actually the simulations $\precsim$ and $\precsim'$ can also be flipped if the target programs are deterministic. It is unclear if the refinement or simulation can still be flipped in concurrent settings where programs have non-deterministic behaviors. The problem is that the target program can be more fine-grained and have more non-deterministic interleavings than the source.

A: With non-preemptive semantics, the non-determinism is focused on certain switch points. Then, in our simulation, we require the source and target to switch at the same time and to the same thread. As a result, although the switching step is still non-deterministic, there exists a one-to-one correspondence between the switching steps of the source and of the target. Thus such non-determinism will not affect the flip of our simulation.

Q: How to support benign races and relaxed machine models? It is difficult to write useful DRF programs within *sequential* languages (e.g., CompCert Clight) because they do not provide synchronization primitives (e.g., locks). Nevertheless, we can implement synchronization primitives as external modules so that the threads written in the sequential languages can call functions (e.g., lock-acquire and lock-release) in the external modules. The implementations π_o of the synchronization primitives may be written directly in assembly languages and very efficient which may lead to benign races when used by multiple threads simultaneously (e.g., see Fig. 10 for such an efficient implementation of spin locks). We may view that there is a separate compiler transforming the abstract primitives γ_o to their implementations π_o. But how do we prove the compilation correctness in the case when the target code may contain races?

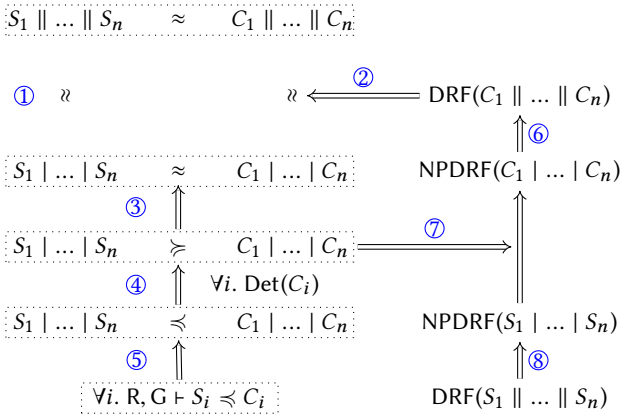


Figure 2. Our basic framework

In addition, our previous discussions all assumed that the source and target programs have sequentially consistent (SC) behaviors. But real-world target machines commonly use relaxed semantics. Although most relaxed models have DRF guarantees saying that DRF programs have SC behaviors in the relaxed semantics, there are no such guarantees for racy programs. It is unclear whether the compilation is still correct when the target code (which may have benign races due to the use of efficient implementations of synchronization primitives) uses relaxed semantics.

A: Although the target assembly program using π_o may contain races, we do know that the assembly program using γ_o is DRF if the source program using γ_o is DRF and the compilation is DRF-preserving. We propose a compositional simulation $\pi_o \preccurlyeq^{\text{o}} \gamma_o$, which ensures a strengthened DRF-guarantee theorem for the x86-TSO model. It says, the x86-TSO program using π_o behaves the same as the program using γ_o in the SC semantics if the latter is DRF.

2.4 The framework and key semantics components

Figure 2 shows the semantics components and proof steps in our basic framework for DRF and SC target code. The goal of our compilation correctness proof is to show the semantics preservation (i.e., $S_1 \| \ldots \| S_n \approx C_1 \| \ldots \| C_n$ at the top of the figure). This follows the correctness of separate compilation of each module, formulated as $\text{R}, \text{G} \vdash S_i \preccurlyeq C_i$ (the bottom

$$\mathbb{P} \overset{\text{def}}{=} \textbf{let } \{\gamma_1, \ldots, \gamma_m, \gamma_o\} \textbf{ in } \mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n$$
$$P^{\mathtt{sc}} \overset{\text{def}}{=} \textbf{let } \{\pi_1^{\mathtt{sc}}, \ldots, \pi_m^{\mathtt{sc}}, \gamma_o\} \textbf{ in } \mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n$$
$$P^{\mathtt{rmm}} \overset{\text{def}}{=} \textbf{let } \{\pi_1^{\mathtt{rmm}}, \ldots, \pi_m^{\mathtt{rmm}}, \pi_o^{\mathtt{rmm}}\} \textbf{ in } \mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n$$

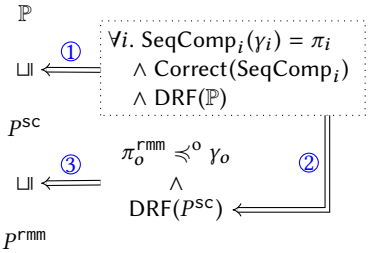


Figure 3. The extended framework

left), which is our new footprint-preserving module-local simulation. We do the proofs in the following steps. Double arrows in the figure are logical implications.

First, we restrict the compilation to source preemptive code that is race-free (i.e., DRF($S_1 \parallel \ldots \parallel S_n$) at the right bottom of the figure). Then from the equivalence between preemptive and non-preemptive semantics, we derive ①, the equivalence between $S_1 \parallel \ldots \parallel S_n$ and $S_1 \mid \ldots \mid S_n$, the latter representing non-preemptive execution of the threads. Similarly, *if we have* DRF($C_1 \parallel \ldots \parallel C_n$) (at the top right), we can derive ②. With ① and ②, we can derive the semantics preservation $\approx$ between preemptive programs from $\approx$ between their non-preemptive counterparts.

Second, DRF of the target programs is obtained through the path ⑥,⑦ and ⑧. We define a notion of DRF for non-preemptive programs (called NPDRF), making it equivalent to DRF, from which we derive ⑥ and ⑧. To know NPDRF($C_1 \mid \ldots \mid C_n$) from NPDRF($S_1 \mid \ldots \mid S_n$), we need a DRF-preserving simulation $\preccurlyeq$ between non-preemptive programs (see ⑦).

Third, the DRF-preserving simulation $\preccurlyeq$ for non-preemptive whole programs can be derived by composing our footprint-preservation local simulation (step ⑤). Given the the downward whole-program simulation, we flip it to get an *upward* one (step ④), with the extra premise that the local execution in each target module is deterministic. Using the simulation in both directions we derive the equivalence (step ③).

The extended framework. Figure 3 shows our ideas for adapting the results of Fig. 2 to relaxed target models and to allow the target programs to contain confined benign races. The goal of the compilation correctness proof is still to show the refinement $\mathbb{P} \sqsupseteq P^{\mathtt{rmm}}$. Here the source $\mathbb{P}$ contains a library module γ_o of the abstract synchronization primitives, as well as normal client modules $\gamma_1, \ldots, \gamma_m$. Threads in $\mathbb{P}$ can call functions in these modules. In the target code $P^{\mathtt{rmm}}$, each client module γ_i is compiled to π_i using a sequential compiler $\mathsf{SeqComp}_i$. The library module γ_o is implemented as π_o, which may contain benign races. The superscript rmm indicates the use of relaxed memory models.

To prove the refinement, we first apply the results of Fig. 2 and prove $\mathbb{P} \sqsupseteq P^{\mathtt{sc}}$ assuming DRF($\mathbb{P}$) and correctness of each

(*Entry*)	f	$\in$	*String*	(*GEnv*) $ge \in \mathcal{P}(Addr)$
(*Prog*)	$P, \mathbb{P}$	::=	**let** Π **in** $\mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n$	
(*MdSet*)	Π, Γ	::=	$\{(tl_1, ge_1, \pi_1), \ldots, (tl_m, ge_m, \pi_m)\}$	
(*Lang*)	tl, sl	::=	(*Module*, *Core*, InitCore, $\longmapsto$)	
(*Module*)	π, γ	::=	...	(*Core*) $\kappa, \mathbb{k}$ ::= ...
InitCore	$\in$	*Module* $\rightarrow$ *Entry* $\rightharpoonup$ *Core*		
$\longmapsto$	$\in$	*FList* $\times$ (*Core* $\times$ *State*) $\rightarrow$ $\mathcal{P}((Msg \times FtPrt) \times ((Core \times State) \cup \textbf{abort}))$		
(*ThrdID*)	t	$\in$	$\mathbb{N}$	(*Addr*) l ::= ...
(*Val*)	v	::=	$l \mid \ldots$	(*FList*) $F, \mathbb{F} \in \mathcal{P}^{\omega}(Addr)$
(*State*)	σ, Σ	$\in$	$Addr \rightharpoonup_{\mathsf{fin}} Val$	
(*FtPrt*)	δ, Δ	::=	(rs, ws)	where $rs, ws \in \mathcal{P}(Addr)$
(*Msg*)	ι	::=	$\tau \mid e \mid \mathtt{ret} \mid$ EntAtom $\mid$ ExtAtom	
(*Event*)	e	::=	...	
(*Config*)	ϕ, Φ	::=	$(\kappa, \sigma) \mid \textbf{abort}$	

Figure 4. The abstract concurrent language

$\mathsf{SeqComp}_i$ (step ① in Fig. 3). Here the superscript sc means the use of SC assembly semantics. Each client module π_i in $P^{\mathtt{sc}}$ has the same code as in $P^{\mathtt{rmm}}$ but uses the SC semantics. Note that $P^{\mathtt{sc}}$ still uses the abstract library γ_o rather than its racy implementation π_o. We can view that at this step the library is compiled by an identity transformation.

Figure 2 also shows DRF preservation, so we know DRF($P^{\mathtt{sc}}$) given DRF($\mathbb{P}$) (step ② in Fig. 3). The last step ③ is our strengthened DRF-guarantee theorem. We require a compositional simulation relation $\pi_o^{\mathtt{rmm}} \preccurlyeq^{\mathrm{o}} \gamma_o$ holds, saying that π_o in the relaxed semantics indeed implements γ_o.

The approach not only supports racy implementations of synchronization primitives, but also applies in more general cases when π_o is a racy implementation of a general concurrent object such as a stack or a queue. For instance, π_o could be the Treiber stack implementation, and then γ_o could be an atomic abstract stack. Then $\pi_o^{\mathtt{rmm}} \preccurlyeq^{\mathrm{o}} \gamma_o$ can be viewed as a correctness criterion of the object implementation in a relaxed model. It ensures a kind of "contextual refinement", i.e., any client threads using π_o (in relaxed semantics) generate no more observable behaviors than using γ_o instead (in SC semantics), as long as the clients using γ_o is DRF.

Although we expect the approach is general enough to work for different relaxed machine memory models, so far we have only proved the result for x86-TSO, as shown in Sec. 7.3 (and the proofs are given in the technical report).

Note that the notations used here are simplified ones to give a semi-formal overview of the key ideas. We may use different notations in the formal development below.

3 The Language and Semantics

3.1 The abstract language

Figure 4 shows the syntax and the state model of an abstract language for preemptive concurrent programming. A program P consists of n threads, each with an entry f pointing to

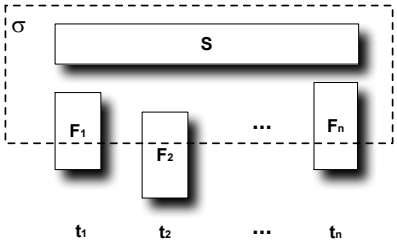


Figure 5. The state model

a module in Π. A module declaration in Π is a triple consisting of the language declaration *tl*, the global environment *ge* containing the addresses of static global variables declared in the module, and the code π. We use $\mathcal{P}(Addr)$ to represent the powerset of memory addresses *Addr*.

The abstract module language *tl* is defined as a tuple (*Module*, *Core*, InitCore, $\longmapsto$), whose components can be instantiated for different concrete languages. *Module* describes the syntax of programs. Following Compositional CompCert [20], *Core* is the set of internal "core" states κ, such as control continuations or register files. The function InitCore returns an initial "core" state κ whenever a thread is created or an external function call is made. In this paper we mainly focus on threads as different modules, and omit the external calls to simplify the presentation. *We do support external calls in our Coq implementation in the same way as in Compositional CompCert.* The labelled transition "$\longmapsto$" models the local execution of a module, which we explain below.

Module-local semantics. The local execution step inside a module is in the form of $F \vdash (\kappa, \sigma) \xmapsto[\delta]{\iota} (\kappa', \sigma')$. The *global* memory state σ is a finite partial mapping from memory addresses to values, where the addresses (*Addr*) and values (*Val*) can be instantiated for concrete languages. Each step is also labeled with a message ι and a footprint δ.

Each module also has a *free list* F, an *infinite* set of memory addresses. It models the preserved space for allocating local stack frames. Initially we require $F \cap \mathrm{dom}(\sigma) = \emptyset$, and the only memory accessible by the module is the static global variables declared in the *ge* of all modules, represented as the *shared* part S in Fig. 5. The local execution of a module may allocate memory from its F, which enlarges the state σ. So "$F - \mathrm{dom}(\sigma)$" is the set of free addresses, depicted in Fig. 5 as the part outside of the boundary of σ. The memory allocated from F is exclusively owned by this module. F for different modules must be *disjoint* (see the Load rule in Fig. 7).

The messages ι contain information about the module-local steps. Here we only consider externally observable events e (such as outputs), termination of threads (`ret`), and the beginning and the end of *atomic blocks* (EntAtom and ExtAtom). Any other steps are silent, labeled with τ. The label τ is often omitted for cleaner presentation. Atomic blocks are the language constructs to ensure sequential execution of code blocks inside them. They can be implemented differently in different module languages. The messages define the

$$\sigma \overset{rs}{=\!=} \sigma' \quad \text{iff} \quad \forall l \in rs.\ l \notin (\mathrm{dom}(\sigma) \cup \mathrm{dom}(\sigma')) \vee l \in (\mathrm{dom}(\sigma) \cap \mathrm{dom}(\sigma')) \wedge \sigma(l) = \sigma'(l)$$

$$\delta \subseteq \delta' \quad \text{iff} \quad (\delta.rs \subseteq \delta'.rs) \wedge (\delta.ws \subseteq \delta'.ws)$$

$$\delta \cup \delta' \overset{\text{def}}{=} (\delta.rs \cup \delta'.rs,\ \delta.ws \cup \delta'.ws)$$

$$\mathtt{forward}(\sigma, \sigma') \ \text{iff} \ (\mathrm{dom}(\sigma) \subseteq \mathrm{dom}(\sigma'))$$

$$\mathtt{LEqPre}(\sigma_1, \sigma_2, \delta, F) \quad \text{iff} \quad \sigma_1 \overset{\delta.rs}{=\!=} \sigma_2 \wedge (\mathrm{dom}(\sigma_1) \cap \delta.ws) = (\mathrm{dom}(\sigma_2) \cap \delta.ws) \wedge (\mathrm{dom}(\sigma_1) \cap F) = (\mathrm{dom}(\sigma_2) \cap F)$$

$$\mathtt{LEqPost}(\sigma_1, \sigma_2, \delta, F) \quad \text{iff} \quad \sigma_1 \overset{\delta.ws}{=\!=} \sigma_2 \wedge (\mathrm{dom}(\sigma_1) \cap F) = (\mathrm{dom}(\sigma_2) \cap F)$$

$$\mathtt{LEffect}(\sigma_1, \sigma_2, \delta, F) \quad \text{iff} \quad \sigma_1 \overset{\mathrm{dom}(\sigma_1) - \delta.ws}{=\!=\!=} \sigma_2 \wedge (\mathrm{dom}(\sigma_2) - \mathrm{dom}(\sigma_1)) \subseteq (\delta.ws \cap F)$$

Figure 6. Auxiliary definitions about states and footprints

protocols of communications with the global whole-program semantics (presented below).

The footprint δ is a pair (rs, ws) of the read set and write set of memory locations in this step. We write `emp` for the footprint where both sets are empty.

Note we use two sets of symbols to distinguish the source and the target level notations. Symbols such as $\mathbb{P}$, $\mathbb{F}$, $\Bbbk$, Σ, Γ and γ are used for the source. Their counterparts, P, F, κ, σ, Π and π, are for the target.

Well-defined languages. Although the abstract module language *tl* can be instantiated with different real languages, the instantiation needs to satisfy certain basic requirements. We formulate these requirements in Def. 1. It gives us an extensional interpretation of footprints.

Definition 1 (Well-Defined Languages). $\mathtt{wd}(tl)$ iff , for any execution step $F \vdash (\kappa, \sigma) \xmapsto[\delta]{\iota} (\kappa', \sigma')$ in this language, all of the following hold (some auxiliary definitions are in Fig. 6):

(1) $\mathtt{forward}(\sigma, \sigma')$;
(2) $\mathtt{LEffect}(\sigma, \sigma', \delta, F)$;
(3) for any σ_1, if $\mathtt{LEqPre}(\sigma, \sigma_1, \delta, F)$, then there exists σ_1' such that $F \vdash (\kappa, \sigma_1) \xmapsto[\delta]{\iota} (\kappa', \sigma_1')$ and $\mathtt{LEqPost}(\sigma', \sigma_1', \delta, F)$.
(4) let $\delta_0 = \bigcup\{\delta \mid \exists \kappa', \sigma'.\ F \vdash (\kappa, \sigma) \xmapsto[\delta]{\tau} (\kappa', \sigma')\}$. For any σ_1, if $\mathtt{LEqPre}(\sigma, \sigma_1, \delta_0, F)$, then for any $\kappa_1', \sigma_1', \iota_1, \delta_1$, $F \vdash (\kappa, \sigma_1) \xmapsto[\delta_1]{\iota_1} (\kappa_1', \sigma_1') \implies \exists \sigma'.\ F \vdash (\kappa, \sigma) \xmapsto[\delta_1]{\iota_1} (\kappa_1', \sigma')$.

It requires that a step may enlarge the memory domain but cannot reduce it (Item (1)). and the additional memory should be allocated from F and included in the write set (Item (2)). Item (2) also requires that the memory out of the write set should keep unchanged. Item (3) says the memory updates and allocation only depend on the memory content in the read set, and the set of memory locations already allocated from F. Item (4) requires that the non-determinism of each step is not affected by memory contents outside of all the possible read sets.

3.2 The global preemptive semantics

Figure 7 shows the global states and selected global semantics rules to model the interaction between modules. The world W consists of the thread pool T, the ID $\mathtt{t}$ of the current thread, a bit d indicating whether the current thread is in an atomic block or not, and the memory state σ. The thread pool T maps a thread ID to a triple recording the module language tl, the free list F, and the current core state κ.

The Load rule in Fig. 7 shows the initialization of the world from the program and an initial program state σ. Global transitions of W are also labeled with footprints and messages o. Each global step executes the module locally and processes the message of the local transition. The EntAt and ExtAt rules correspond to the entry and exit of atomic blocks, respectively. The flag d is flipped in the two steps. Since context-switch can be done only when d is 0, as required by the Switch rule below, we know a thread in its atomic block cannot be preempted. The Switch rule shows that context switch can occur at any program point outside of atomic blocks ($d = 0$).

Below we write $F \vdash \phi \overset{\tau}{\underset{\delta}{\longmapsto}}{}^{*}\phi'$ for zero or multiple silent steps, where δ is the accumulation of the footprint of each step. $F \vdash \phi \overset{\tau}{\underset{\delta}{\longmapsto}}{}^{+}\phi'$ represents at least one step. Similar notations are used for global steps. Also $W \Rightarrow^{*} W'$ is for zero or multiple steps that either are silent or produce $\mathtt{sw}$ events.

Event-trace refinement and equivalence. An externally observable event trace $\mathcal{B}$ is a finite or infinite sequence of external events e, and may end with a termination marker **done** or an abortion marker **abort**. Following the definition in CompCert, we use $(P, \sigma) \sqsubseteq (\mathbb{P}, \Sigma)$ and $(P, \sigma) \approx (\mathbb{P}, \Sigma)$ to represent the event-trace refinement and equivalence.

3.3 The non-preemptive semantics

A key step in our framework is to reduce the semantics preservation under the preemptive semantics to the semantics preservation in non-preemptive semantics. To distinguish from the preemptive concurrency, we write **let** Π **in** $\mathtt{f}_1 \mid \ldots \mid \mathtt{f}_n$ for the program with non-preemptive semantics, denoted by $\hat{P}$. The global world $\widehat{W}$ is defined similarly as the preemptive world W, except that $\widehat{W}$ keeps an atomic bit map $\mathbb{d}$ recording whether each thread's next step is inside an atomic block. We need to record the atomic bits of all threads because the context switch may occur when a thread just enters an atomic block.

The last two rules in Fig. 7 define the non-preemptive global steps $\widehat{W} \overset{o}{\underset{\delta}{:\Longrightarrow}} \widehat{W}'$. More rules are shown in Appendix A. There is no rule like Switch of the preemptive semantics, since context switch occurs only at synchronization points. $\mathsf{EntAt}_{\mathsf{np}}$ and $\mathsf{ExtAt}_{\mathsf{np}}$ execute one step of the current thread $\mathtt{t}$, and then non-deterministically switch to a thread $\mathtt{t}'$. The corresponding global steps produce the $\mathtt{sw}$ events.

$$
\begin{array}{llll}
(\mathit{World}) & W, \mathbb{W} & ::= & (T, \mathtt{t}, d, \sigma) \qquad (\mathit{AtomBit}) \quad d ::= 0 \mid 1 \\
(\mathit{NPWorld}) & \widehat{W}, \widehat{\mathbb{W}} & ::= & (T, \mathtt{t}, \mathbb{d}, \sigma) \qquad (\mathit{GMsg}) \quad o ::= \tau \mid e \mid \mathtt{sw} \\
(\mathit{AtomBits}) & \mathbb{d} & ::= & \{\mathtt{t}_1 \rightsquigarrow d_1, \ldots, \mathtt{t}_n \rightsquigarrow d_n\} \\
(\mathit{ThrdPool}) & T, \mathbb{T} & ::= & \{\mathtt{t}_1 \rightsquigarrow (tl_1, F_1, \kappa_1), \ldots, \mathtt{t}_n \rightsquigarrow (tl_1, F_n, \kappa_n)\}
\end{array}
$$

$$
\frac{\begin{array}{l}\text{for all } i \text{ and } j \text{ in } \{1, \ldots, n\}, \text{ and } i \neq j: \\ \quad F_i \cap F_j = \emptyset \quad \mathrm{dom}(\sigma) \cap F_i = \emptyset \\ \quad tl_i.\mathrm{InitCore}(\pi_i, \mathtt{f}_i) = \kappa_i, \ \text{ where } (tl_i, ge_i, \pi_i) \in \Pi \\ T = \{1 \rightsquigarrow (tl_1, F_1, \kappa_1), \ldots, n \rightsquigarrow (tl_n, F_n, \kappa_n)\} \quad \mathtt{t} \in \mathrm{dom}(T)\end{array}}{(\mathbf{let}\ \Pi\ \mathbf{in}\ \mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n, \sigma) \overset{\mathit{load}}{\Longrightarrow} (T, \mathtt{t}, 0, \sigma)}\ \textsf{Load}
$$

$$
\frac{T(\mathtt{t}) = (tl, F, \kappa) \qquad F \vdash (\kappa, \sigma) \overset{\tau}{\underset{\delta}{\longmapsto}} (\kappa', \sigma')}{(T, \mathtt{t}, d, \sigma) \overset{\tau}{\underset{\delta}{\Rightarrow}} (T\{\mathtt{t} \rightsquigarrow (tl, F, \kappa')\}, \mathtt{t}, d, \sigma')}\ \tau\textsf{-step}
$$

$$
\frac{T(\mathtt{t}) = (tl, F, \kappa) \qquad F \vdash (\kappa, \sigma) \overset{\mathsf{EntAtom}}{\underset{\mathsf{emp}}{\longmapsto}} (\kappa', \sigma)}{(T, \mathtt{t}, 0, \sigma) \overset{\tau}{\underset{\mathsf{emp}}{\Longrightarrow}} (T\{\mathtt{t} \rightsquigarrow (tl, F, \kappa')\}, \mathtt{t}, 1, \sigma)}\ \textsf{EntAt}
$$

$$
\frac{T(\mathtt{t}) = (tl, F, \kappa) \qquad F \vdash (\kappa, \sigma) \overset{\mathsf{ExtAtom}}{\underset{\mathsf{emp}}{\longmapsto}} (\kappa', \sigma)}{(T, \mathtt{t}, 1, \sigma) \overset{\tau}{\underset{\mathsf{emp}}{\Longrightarrow}} (T\{\mathtt{t} \rightsquigarrow (tl, F, \kappa')\}, \mathtt{t}, 0, \sigma)}\ \textsf{ExtAt}
$$

$$
\frac{\mathtt{t}' \in \mathrm{dom}(T)}{(T, \mathtt{t}, 0, \sigma) \overset{\mathtt{sw}}{\underset{\mathsf{emp}}{\Longrightarrow}} (T, \mathtt{t}', 0, \sigma)}\ \textsf{Switch}
$$

$$
\frac{\begin{array}{l}T(\mathtt{t}) = (tl, F, \kappa) \qquad \mathbb{d}(\mathtt{t}) = 0 \qquad \mathtt{t}' \in \mathrm{dom}(T) \\ F \vdash (\kappa, \sigma) \overset{\mathsf{EntAtom}}{\underset{\mathsf{emp}}{\longmapsto}} (\kappa', \sigma) \qquad T' = T\{\mathtt{t} \rightsquigarrow (tl, F, \kappa')\}\end{array}}{(T, \mathtt{t}, \mathbb{d}, \sigma) :\overset{\mathtt{sw}}{\underset{\mathsf{emp}}{\Longrightarrow}} (T', \mathtt{t}', \mathbb{d}\{\mathtt{t} \rightsquigarrow 1\}, \sigma)}\ \textsf{EntAt}_{\mathsf{np}}
$$

$$
\frac{\begin{array}{l}T(\mathtt{t}) = (tl, F, \kappa) \qquad \mathbb{d}(\mathtt{t}) = 1 \qquad \mathtt{t}' \in \mathrm{dom}(T) \\ F \vdash (\kappa, \sigma) \overset{\mathsf{ExtAtom}}{\underset{\mathsf{emp}}{\longmapsto}} (\kappa', \sigma) \qquad T' = T\{\mathtt{t} \rightsquigarrow (tl, F, \kappa')\}\end{array}}{(T, \mathtt{t}, \mathbb{d}, \sigma) :\overset{\mathtt{sw}}{\underset{\mathsf{emp}}{\Longrightarrow}} (T', \mathtt{t}', \mathbb{d}\{\mathtt{t} \rightsquigarrow 0\}, \sigma)}\ \textsf{ExtAt}_{\mathsf{np}}
$$

Figure 7. Preemptive and non-preemptive global semantics

4 The Footprint-Preserving Simulation

In this section, we define a *module-local* simulation as the correctness obligation of each module's compilation, which is compositional and preserves footprints, allowing us to derive a whole-program simulation that preserves DRF.

Footprint consistency. As in CompCert, the simulation requires that the source and the target generate the same external events. In addition, it also requires that the target has the same or smaller footprints than the source, which is important to ensure DRF-preservation. Recall that the memory accessible by a thread $\mathtt{t}_i$ consists of two parts, the *shared* memory $\mathbb{S}$ and the local memory allocated from $\mathbb{F}_i$, as shown in Fig. 5. DRF informally requires that the threads never have conflicting accesses to the memory in $\mathbb{S}$ at the same time.

We introduce the triple μ below to record the key information about the shared memory at the source and the target.

$$
\mu \overset{\text{def}}{=} (\mathbb{S}, S, f), \ \text{where } \mathbb{S}, S \in \mathcal{P}(\mathit{Addr}) \text{ and } f \in \mathit{Addr} \rightharpoonup \mathit{Addr}.
$$

$$f\{\mathbb{S}\} \stackrel{\text{def}}{=} \{l' \mid \exists l.\ (l \in \mathbb{S}) \wedge f(l) = l'\}$$
$$f|_{\mathbb{S}} \stackrel{\text{def}}{=} \{(l, f(l)) \mid l \in (\mathbb{S} \cap \text{dom}(f))\}$$
$$\text{wf}(\mu) \quad \text{iff} \quad \text{injective}(f) \wedge \text{dom}(f) = \mathbb{S} \wedge f\{\mathbb{S}\} \subseteq S$$
$$\text{FPmatch}(\mu, \Delta, \delta) \quad \text{iff} \quad (\delta.rs \cap S \subseteq f\{\Delta.rs\}) \wedge (\delta.ws \cap S \subseteq f\{\Delta.ws\}) \quad \text{where } \mu = (\mathbb{S}, S, f)$$
$$\text{closed}(\mathbb{S}, \Sigma) \quad \text{iff} \quad \text{cl}(\mathbb{S}, \Sigma) \subseteq \mathbb{S}$$
$$\text{cl}(\mathbb{S}, \Sigma) \stackrel{\text{def}}{=} \bigcup_k \text{cl}_k(\mathbb{S}, \Sigma), \quad \text{where } \text{cl}_k(\mathbb{S}, \Sigma) \text{ is inductively defined:}$$
$$\text{cl}_0(\mathbb{S}, \Sigma) \stackrel{\text{def}}{=} \mathbb{S} \quad \text{cl}_{k+1}(\mathbb{S}, \Sigma) \stackrel{\text{def}}{=} \{l' \mid \exists l.\ (l \in \text{cl}_k(\mathbb{S}, \Sigma)) \wedge \Sigma(l) = l'\}$$
$$\text{initM}(\varphi, ge, \Sigma, \sigma) \quad \text{iff} \quad ge \subseteq \text{dom}(\Sigma) \wedge \text{closed}(\text{dom}(\Sigma), \Sigma) \wedge \text{dom}(\sigma) = \varphi\{\text{dom}(\Sigma)\} \wedge \text{Inv}(\varphi, \Sigma, \sigma)$$
$$\text{Inv}(f, \Sigma, \sigma) \quad \text{iff} \quad \forall l, l'.\ l \in \text{dom}(\Sigma) \wedge f(l) = l' \implies l' \in \text{dom}(\sigma) \wedge \Sigma(l) \overset{f}{\hookrightarrow} \sigma(l')$$
$$v_1 \overset{f}{\hookrightarrow} v_2 \quad \text{iff} \quad (v_1 \notin Addr) \wedge (v_1 = v_2) \vee (v_1, v_2 \in Addr \wedge f(v_1) = v_2)$$
$$\text{HG}(\Delta, \Sigma', \mathbb{F}, \mathbb{S}) \quad \text{iff} \quad \Delta \subseteq (\mathbb{F} \cup \mathbb{S}) \wedge \text{closed}(\mathbb{S}, \Sigma')$$
$$\text{LG}(\mu, (\delta, \sigma', F), (\Delta, \Sigma')) \quad \text{iff} \quad \delta \subseteq (F \cup \mu.S) \wedge \text{closed}(\mu.S, \sigma') \wedge \text{FPmatch}(\mu, \Delta, \delta) \wedge \text{Inv}(\mu.f, \Sigma', \sigma')$$
$$\text{R}(\Sigma, \Sigma', \mathbb{F}, \mathbb{S}) \quad \text{iff} \quad (\Sigma \overset{\mathbb{F}}{=\!=} \Sigma') \wedge \text{closed}(\mathbb{S}, \Sigma') \wedge \text{forward}(\Sigma, \Sigma')$$
$$\text{Rely}(\mu, (\Sigma, \Sigma', \mathbb{F}), (\sigma, \sigma', F)) \quad \text{iff} \quad \text{R}(\Sigma, \Sigma', \mathbb{F}, \mu.\mathbb{S}) \wedge \text{R}(\sigma, \sigma', F, \mu.S) \wedge \text{Inv}(\mu.f, \Sigma', \sigma')$$

Figure 8. Footprint matching and rely/guarantee conditions

Here $\mathbb{S}$ and S specify the shared memory locations at the source and the target respectively. The partial mapping f maps locations at the source level to those at the target. We require μ to be well-formed, defined as $\text{wf}(\mu)$ in Fig. 8.

Then, given footprints Δ and δ, we define their consistency with respect to μ as $\text{FPmatch}(\mu, \Delta, \delta)$ in Fig. 8. It says the *shared* locations in δ must be contained in Δ, modulo the mapping $\mu.f$. We only consider the shared locations in $\mu.S$ because accesses of local memory would *not* cause races.

Rely/guarantee conditions. We use rely and guarantee conditions to specify interaction between modules. They need to enforce the view of accessibility of shared and local memory in Fig. 5. More specifically, a module expects others to keep its local memory (in $\mathbb{F}$) intact. In addition, although other modules may update the shared memory $\mathbb{S}$, they must preserve certain properties of $\mathbb{S}$. One such property is that $\mathbb{S}$ cannot contain memory pointers pointing to local memory cells in any $\mathbb{F}_i$[1]. Otherwise a thread $\mathtt{t}_j$ can update the memory in $\mathbb{F}_i$ by tracing these pointers. This requirement is formalized as $\text{closed}(\mathbb{S}, \Sigma)$ in Fig. 8, which says the closure of addresses reachable from $\mathbb{S}$ must be no bigger than $\mathbb{S}$. We encode these requirements in the rely condition Rely in Fig. 8, and define the guarantee conditions HG and LG correspondingly.

The simulation. Below we define $(sl, ge, \gamma) \preccurlyeq_\varphi (tl, ge', \pi)$ to relate the *non-preemptive* executions of the source module (sl, ge, γ) and the target one (tl, ge', π). The injective function φ maps source addresses to the target ones.

[1] We disallow escape of pointers pointing to stack variables.

Definition 2 (Module-Local Downward Simulation)**.** $(sl, ge, \gamma) \preccurlyeq_\varphi (tl, ge', \pi)$ iff,
for all $\mathtt{f}, \Bbbk, \Sigma, \sigma, \mathbb{F}, F$, and $\mu = (\text{dom}(\Sigma), \text{dom}(\sigma), \varphi|_{\text{dom}(\Sigma)})$, if $sl.\text{InitCore}(\gamma, \mathtt{f}) = \Bbbk$, $\varphi\{ge\} = ge'$, $\text{initM}(\varphi, ge, \Sigma, \sigma)$, and $\mathbb{F} \cap \text{dom}(\Sigma) = F \cap \text{dom}(\sigma) = \emptyset$, then there exist $i \in \mathtt{index}$ and κ such that $tl.\text{InitCore}(\pi, \mathtt{f}) = \kappa$, and
$$(\mathbb{F}, (\Bbbk, \Sigma), \mathtt{emp}) \preccurlyeq^i_\mu (F, (\kappa, \sigma), \mathtt{emp}),$$
where $(\mathbb{F}, (\Bbbk, \Sigma), \Delta) \preccurlyeq^i_\mu (F, (\kappa, \sigma), \delta)$ is defined in Def. 3.

It says that, starting from some core states $\Bbbk$ and κ, with any states (Σ and σ) and free lists ($\mathbb{F}$ and F) satisfying some initial constraints, we have the simulation $(\mathbb{F}, (\Bbbk, \Sigma), \mathtt{emp}) \preccurlyeq^i_\mu (F, (\kappa, \sigma), \mathtt{emp})$ defined in Def. 3. Here the initial states Σ and σ need to be related through initM (see Fig. 8). As the last condition of initM, the invariant Inv relates the domain and the memory values between Σ and σ through φ.

Definition 3. $(\mathbb{F}, (\Bbbk, \Sigma), \Delta_0) \preccurlyeq^i_\mu (F, (\kappa, \sigma), \delta_0)$ is the largest relation such that, whenever $(\mathbb{F}, (\Bbbk, \Sigma), \Delta_0) \preccurlyeq^i_\mu (F, (\kappa, \sigma), \delta_0)$, then the following are true:

1. for all $\Bbbk'$, Σ' and Δ, if $\mathbb{F} \vdash (\Bbbk, \Sigma) \overset{\tau}{\underset{\Delta}{\mapsto}} (\Bbbk', \Sigma')$ and $(\Delta_0 \cup \Delta) \subseteq (\mathbb{F} \cup \mu.\mathbb{S})$, then one of the following holds:
   a. $\exists j < i.\ (\mathbb{F}, (\Bbbk', \Sigma'), \Delta_0 \cup \Delta) \preccurlyeq^j_\mu (F, (\kappa, \sigma), \delta_0)$, or
   b. there exist κ', σ', δ and j such that:
      i. $F \vdash (\kappa, \sigma) \overset{\tau}{\underset{\delta}{\mapsto}}{}^+ (\kappa', \sigma')$;
      ii. $(\delta_0 \cup \delta) \subseteq (F \cup \mu.S)$ and $\text{FPmatch}(\mu, \Delta_0 \cup \Delta, \delta_0 \cup \delta)$; and
      iii. $(\mathbb{F}, (\Bbbk', \Sigma'), \Delta_0 \cup \Delta) \preccurlyeq^j_\mu (F, (\kappa', \sigma'), \delta_0 \cup \delta)$.
2. for all $\Bbbk'$ and ι, if $\mathbb{F} \vdash (\Bbbk, \Sigma) \overset{\iota}{\underset{\text{emp}}{\mapsto}} (\Bbbk', \Sigma)$, $\iota \neq \tau$, and $\text{HG}(\Delta_0, \Sigma, \mathbb{F}, \mu.\mathbb{S})$, there exist κ', δ, σ' and κ'' such that:
   a. $F \vdash (\kappa, \sigma) \overset{\tau}{\underset{\delta}{\mapsto}}{}^* (\kappa', \sigma')$, and $F \vdash (\kappa', \sigma') \overset{\iota}{\underset{\text{emp}}{\mapsto}} (\kappa'', \sigma')$, and
   b. $\text{LG}(\mu, (\delta_0 \cup \delta, \sigma', F), (\Delta_0, \Sigma))$, and
   c. for all σ'' and Σ', if $\text{Rely}(\mu, (\Sigma, \Sigma', \mathbb{F}), (\sigma', \sigma'', F))$, then there exists j such that
   $$(\mathbb{F}, (\Bbbk', \Sigma'), \mathtt{emp}) \preccurlyeq^j_\mu (F, (\kappa'', \sigma''), \mathtt{emp}).$$

The simulation $(\mathbb{F}, (\Bbbk, \Sigma), \Delta_0) \preccurlyeq^i_\mu (F, (\kappa, \sigma), \delta_0)$ carries Δ_0 and δ_0, the footprints accumulated at the source and the target, respectively. The definition follows the diagram in Fig. 1(d). For every τ-step in the source (case 1), if the newly generated footprints and the accumulated Δ_0 are *in scope* (i.e. every location must either be from the freelist space $\mathbb{F}$ of current thread, or from the shared memory $\mu.\mathbb{S}$), then the step corresponds to zero or multiple τ-steps in the target, and the simulation holds over the resulting states with the accumulated footprints and a new index j. We carry the well-founded index to ensure the simulation preserves termination.

If the source step corresponds to at least one target steps (case 1-b), the footprints at the target must also be in scope and be consistent with the source level footprints. The accumulation of footprints allows us to establish FPmatch for compiler optimizations that reorder the instructions.

At the switch points when the source generates a non-silent message ι (case 2), if the footprints and states satisfy

the high-level guarantee HG, the target must be able to generate the same ι, and the accumulated footprints and the state satisfy the low-level guarantee LG. We also need to consider the interaction with other modules or threads. For any environment steps satisfying Rely, the simulation must hold over the new states, with some index j and *empty* footprints — Since the effects of the current thread have been made visible to the environments at the switch point, we can clear the accumulated footprints. Rely and the guarantees HG and LG are defined in Fig. 8, as explained before.

Note that each case in Def. 3 has prerequisites about the source level footprints (e.g. the footprints are in scope or satisfy HG). We need to prove that these requirements indeed hold at the source level, to make the simulation meaningful instead of being vacuously true. Following Stewart et al. [20], we formalize these requirements separately as ReachClose. The compilation correctness assumes all source modules satisfy ReachClose (see Lm. 4 and Def. 9).

Lemma 4 (Compositionality, ⑤ in Fig. 2)**.**
For any $\mathtt{f}_1, \ldots, \mathtt{f}_n$, φ, $\Gamma = \{(sl_1, ge_1\gamma_1), \ldots, (sl_m, ge_m\gamma_m)\}$, and $\Pi = \{(tl_1, ge'_1, \pi_1), \ldots, (tl_m, ge'_m\pi_m)\}$, if

$$\forall i \in \{1, \ldots, m\}.\ \mathsf{wd}(sl_i) \wedge \mathsf{wd}(tl_i) \wedge \mathsf{ReachClose}(sl_i, ge_i, \gamma_i) \wedge (sl_i, ge_i, \gamma_i) \preccurlyeq_\varphi (tl_i, ge'_i, \pi_i)\ ,$$

then $\quad \mathbf{let}\ \Gamma\ \mathbf{in}\ \mathtt{f}_1 \mid \ldots \mid \mathtt{f}_n \preccurlyeq_\varphi \mathbf{let}\ \Pi\ \mathbf{in}\ \mathtt{f}_1 \mid \ldots \mid \mathtt{f}_n\ .$

Lemma 4 shows the compositionality of our simulation. The whole program *downward* simulation $\hat{\mathbb{P}} \preccurlyeq_\varphi \hat{P}$ relates the non-preemptive execution of the whole source program $\mathbb{P}$ and target program P. The definition is given in Appendix B.

With determinism of the target module language (written as $\det(tl)$), we can flip $\hat{\mathbb{P}} \preccurlyeq_\varphi \hat{P}$ to derive the upward simulation $\hat{P} \leqslant_\varphi \hat{\mathbb{P}}$. We give the definition of $\det(tl)$ and the Flip Lemma (④ in Fig. 2) in Appendix B. Lemma 5 shows the non-preemptive global simulation ensures the refinement.

Lemma 5 (Soundness, ③ in Fig. 2)**.** If $\hat{P} \leqslant_\varphi \hat{\mathbb{P}}$, then $\hat{P} \sqsubseteq_\varphi \hat{\mathbb{P}}$,

where $P \sqsubseteq_\varphi \mathbb{P}$ iff $\forall \Sigma, \sigma.\ \mathsf{initM}(\varphi, \mathsf{GE}(\mathbb{P}.\Gamma), \Sigma, \sigma) \implies (P, \sigma) \sqsubseteq (\mathbb{P}, \Sigma)$

and $\mathsf{GE}(\{(tl_1, ge_1, \pi_1), \ldots, (tl_m, ge_m, \pi_m)\}) \stackrel{\text{def}}{=} \bigcup_{i=1}^{m} ge_i$

5 Data-Race-Freedom

Below we first define the conflict of footprints.

$$\begin{array}{rcl} \delta_1 \frown \delta_2 & \text{iff} & (\delta_1.ws \cap \delta_2 \neq \emptyset) \vee (\delta_2.ws \cap \delta_1 \neq \emptyset) \\ (\delta_1, d_1) \frown (\delta_2, d_2) & \text{iff} & (\delta_1 \frown \delta_2) \wedge (d_1 = 0 \vee d_2 = 0) \end{array}$$

Recall that, when used as a set, δ represents $\delta.rs \cup \delta.ws$. Since we do *not* treat accesses of the same memory location inside atomic blocks as a race, we instrument a footprint δ with the atomic bit d to record whether the footprint is generated inside an atomic block ($d = 1$) or not ($d = 0$). Two instrumented footprints (δ_1, d_1) and (δ_2, d_2) are conflicting if δ_1 and δ_2 are conflicting and at least one of d_1 and d_2 is 0.

We define data races in Fig. 9 for preemptive semantics. In the Race rule, W steps to Race if there are conflicting

$$\frac{(P, \sigma) \stackrel{load}{\Longrightarrow} W \qquad W \Rightarrow^* W' \qquad W' \Longmapsto \mathsf{Race}}{(P, \sigma) \Longmapsto \mathsf{Race}}$$

$$\frac{\mathtt{predict}(W, \mathtt{t}_1, (\delta_1, d_1)) \qquad \mathtt{predict}(W, \mathtt{t}_2, (\delta_2, d_2)) \qquad \mathtt{t}_1 \neq \mathtt{t}_2 \qquad (\delta_1, d_1) \frown (\delta_2, d_2)}{W \Longmapsto \mathsf{Race}}\ \mathsf{Race}$$

$$\frac{W = (T, \_, 0, \sigma) \quad T(\mathtt{t}) = (F, \kappa) \quad F \vdash (\kappa, \sigma) \overset{\tau}{\underset{\delta}{\mapsto}} (\kappa', \sigma')}{\mathtt{predict}(W, \mathtt{t}, (\delta, 0))}\ \text{Predict-0}$$

$$\frac{W = (T, \_, 0, \sigma) \qquad T(\mathtt{t}) = (F, \kappa) \qquad F \vdash (\kappa, \sigma) \overset{\mathsf{EntAtom}}{\underset{\mathsf{emp}}{\longmapsto}} (\kappa', \sigma) \qquad F \vdash (\kappa', \sigma) \overset{\tau}{\underset{\delta}{\mapsto}}{}^* (\kappa'', \sigma'')}{\mathtt{predict}(W, \mathtt{t}, (\delta, 1))}\ \text{Predict-1}$$

Figure 9. Data Races in Preemptive Semantics

footprints of two threads predicted from the current configuration ($\mathtt{predict}(W, \mathtt{t}, (\delta, d))$). Then,

$$\mathsf{DRF}(P, \sigma) \ \text{ iff } \ \neg((P, \sigma) \Longmapsto \mathsf{Race})$$
$$\mathsf{DRF}(P) \ \text{ iff } \ \forall \sigma.\ \mathsf{GE}(P.\Pi) \subseteq \mathsf{locs}(\sigma) \wedge \mathsf{cl}(\mathsf{dom}(\sigma), \sigma) = \mathsf{dom}(\sigma) \implies \mathsf{DRF}(P, \sigma)$$

$\mathsf{NPDRF}(\hat{P}, \sigma)$ and $\mathsf{NPDRF}(\hat{P})$ are defined similarly. We can prove NPDRF is equivalent to DRF (⑥ and ⑧ in Fig. 2). The following Lemma 6 shows the simulation preserves NPDRF. Given the equivalence, we know it also preserves DRF.

Lemma 6 (NPDRF Preservation, ⑦ in Fig. 2)**.**
For any $\hat{\mathbb{P}}$, $\hat{P}$, φ, Σ and σ, if $\hat{P} \leqslant_\varphi \hat{\mathbb{P}}$, $\mathsf{initM}(\varphi, \mathsf{GE}(\hat{\mathbb{P}}.\Gamma), \Sigma, \sigma)$, and $\mathsf{NPDRF}(\hat{\mathbb{P}}, \Sigma)$, then $\mathsf{NPDRF}(\hat{P}, \sigma)$.

Lemma 7 (Semantics Equivalence, ① and ② in Fig. 2)**.**
For any Π, $\mathtt{f}_1, \ldots, \mathtt{f}_m$, σ, if $\mathsf{DRF}(\mathbf{let}\ \Pi\ \mathbf{in}\ \mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_m, \sigma)$, then $(\mathbf{let}\ \Pi\ \mathbf{in}\ \mathtt{f}_1 \mid \ldots \mid \mathtt{f}_m, \sigma) \approx (\mathbf{let}\ \Pi\ \mathbf{in}\ \mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_m, \sigma)$.

6 The Final Theorem

Putting all the previous results together, we are able to prove our final theorem in the basic framework (Fig. 2). We first model a sequential compiler SeqComp as follows:

$$\begin{array}{rcl} \mathsf{CodeT} & \in & \mathit{Module} \rightharpoonup \mathit{Module} \qquad \varphi \in \mathit{Addr} \rightharpoonup \mathit{Addr} \\ \mathsf{SeqComp} & ::= & (\mathsf{CodeT}, \varphi) \end{array}$$

As the key proof obligation, we need to verify that each SeqComp is Correct. The correctness is defined based on our footprint-preserving module-local simulation.

Definition 8 (Sequential Compiler Correctness)**.**
$\mathsf{Correct}(\mathsf{SeqComp}, sl, tl)$ iff

$$\forall \gamma, \pi, ge, ge'.\ \mathsf{SeqComp.CodeT}(\gamma) = \pi \wedge \mathsf{SeqComp}.\varphi\{ge\} = ge' \implies (sl, ge, \gamma) \preccurlyeq_{\mathsf{SeqComp}.\varphi} (tl, ge', \pi)\ .$$

The desired correctness GCorrect (Def. 9) of *concurrent* program compilation is the semantics preservation of whole programs, i.e., every target concurrent program is a refinement of the source. Here all the SeqComp must agree on the transformation φ of global environments (see Def. 9(1)).

Definition 9 (Concurrent Compiler Correctness)**.**
$\mathsf{GCorrect}((\mathsf{SeqComp}_1, sl_1, tl_1), \ldots, (\mathsf{SeqComp}_m, sl_m, tl_m))$ iff for any φ, $\mathtt{f}_1, \ldots, \mathtt{f}_n$, $\Gamma = \{(sl_1, ge_1, \gamma_1), \ldots, (sl_m, ge_m, \gamma_m)\}$, and $\Pi = \{(tl_1, ge'_1, \pi_1), \ldots, (tl_m, ge'_m, \pi_m)\}$, if

1. $\forall i \in \{1,\ldots,m\}.\ (\mathsf{SeqComp}_i.\mathsf{CodeT}(\gamma_i) = \pi_i)\ \wedge\ \mathsf{injective}(\varphi)$
$\wedge\ (\mathsf{SeqComp}_i.\varphi = \varphi)\ \wedge\ \varphi\{ge_i\} = ge_i'$,
2. $\mathsf{Safe}(\mathbf{let}\ \Gamma\ \mathbf{in}\ \mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n)$, and $\mathsf{DRF}(\mathbf{let}\ \Gamma\ \mathbf{in}\ \mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n)$,
3. $\forall i \in \{1,\ldots,m\}.\ \mathsf{ReachClose}(sl_i, ge_i, \gamma_i)$,

then $\mathbf{let}\ \Gamma\ \mathbf{in}\ \mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n \sqsupseteq_\varphi \mathbf{let}\ \Pi\ \mathbf{in}\ \mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n$.

Our final theorem is then formulated as Thm. 10. It says if a set of sequential compilers are certified to satisfy our correctness obligation Correct, the source and target languages sl_i and tl_i are well-defined, and the target languages are deterministic, then the sequential compilers as a whole is GCorrect for compiling concurrent programs. The proof simply applies the lemmas that correspond to ①-⑧ in Fig. 2.

Theorem 10 (Final Theorem).
For any $\mathsf{SeqComp}_1,\ldots,\mathsf{SeqComp}_m, sl_1,\ldots,sl_m, tl_1,\ldots,tl_m$ such that for any $i \in \{1,\ldots,m\}$ we have $\mathsf{wd}(sl_i)$, $\mathsf{wd}(tl_i)$, $\mathsf{det}(tl_i)$, and $\mathsf{Correct}(\mathsf{SeqComp}_i, sl_i, tl_i)$, then
$\mathsf{GCorrect}((\mathsf{SeqComp}_1, sl_1, tl_1),\ldots,(\mathsf{SeqComp}_m, sl_m, tl_m))$.

7 CompCert Verification

We apply our framework to prove the correctness of CompCert-3.0.1 [21] for compilation of multi-threaded Clight programs to x86-SC, i.e. x86 with SC semantics. We further extend the framework (as shown in Fig. 3) to support x86-TSO [19] as the target language. The source program we compile consists of multiple sequential Clight threads. Inter-thread synchronization can be achieved through *external calls* to an external module. Since the external synchronization module can be shared by the threads, below we also refer to it as an *object* and the Clight threads as *clients*.

As a tiny example, Fig. 10(a) shows a spin-lock implementation in a simple imperative language which we call CImp. $\langle C \rangle$ is an atomic block, which cannot be interrupted by other threads. The EntAtom and ExtAtom events are generated at the beginning and the end of the atomic block respectively. The command `assert`(B) aborts if B is false. This source implementation of locks serves as an abstract specification, which is *manually* translated to x86-TSO, as shown in Fig. 10(b). The implementation is similar to the Linux spin-lock implementation [15]. In particular, its load and store of the lock value in the `spin` and `unlock` blocks are *not* `lock`-prefixed. This optimization introduces benign races. With the external lock module, we can implement a DRF counter `inc` in Clight, as shown in Fig. 10(c). An example whole program $\mathbb{P}$ is $\mathbf{let}\ \{\gamma_C, \gamma_{\mathsf{lock}}\}\ \mathbf{in}$ `inc() ‖ inc()`.

As shown in Fig. 3, we compile the code in two steps. First, we use CompCert to compile the Clight client to x86-SC, but leave object code (e.g., γ_{lock} in Fig. 10) untouched. Second, we transform the resulting x86-SC client code to x86-TSO. Syntactically this is an identity transformation, but the semantics changes. Then we manually transform the source object code to x86-TSO code. We can prove the resulting whole program in x86-TSO preserves the source semantics as long as the x86-TSO object code refines its source.

```
lock(){  r := 0; while(r==0){ <r:=[L]; [L]:=0;> }  }
unlock(){  < r := [L]; assert(r == 0); [L] := 1; >}
```
(a) lock specification γ_{lock}

```
lock:    movl    $L,     %ecx
         movl    $0,     %edx
l_acq:   movl    $1,     %eax
    lock cmpxchgl %edx,   (%ecx)
         je      enter
spin:    movl    (%ecx), %ebx
         cmp     $0,     %ebx
         je      spin
         jmp     l_acq
enter:   retl
unlock:  movl    $L,     %eax
         movl    $1,     (%eax)
         retl
```
(b) lock implementation π_{lock}

```
void inc(){
  int32_t tmp;
  lock();
    tmp = x;
    x ++;
  unlock();
  print(tmp);
}
```
(c) a Clight client γ_C

Figure 10. Lock as an external module of Clight programs

7.1 Language Instantiations

We need to first instantiate our abstract languages of Fig. 4 with Clight, x86-SC and the intermediate languages introduced in CompCert. We also instantiate it with the simple language CImp for the source object code.

The Clight language. The *Module* in Fig. 4 is instantiated with the same Clight syntax as in CompCert. The core state κ is a pair of a local state c and an index N indicating the position of the next block in the freelist F to be allocated, as shown below. InitCore initializes N to 0. Local transitions of Clight are instrumented with footprints.

$$\begin{array}{llllllllll} (\mathit{FList}) & F & ::= & b_1 :: b_2 :: \ldots & (\mathit{Block}) & b & \in & \mathbb{N}^+ \\ (\mathit{BIndex}) & N & \in & \mathbb{N} & (\mathit{Core}) & \kappa & ::= & (c, N) \\ (\mathit{Mem}) & \sigma & \in & \mathit{Block} \rightharpoonup_{\mathsf{fin}} (\mathbb{N} \rightharpoonup \mathtt{val}) & & & & \end{array}$$

The source language CImp for objects. The instantiation of the abstract language with CImp lets the atomic blocks generate the EntAtom and ExtAtom events. To allow benign races in the target x86-TSO code, we need to ensure partition of the client data and the object data. This can be enforced through the permissions in the CompCert memory model. The client programs can only access memory locations whose permission is not None. Therefore we set the permission of the object data (the memory at the location L in our example in Fig. 10) to None. Also, we require the CImp program can *only* access memory locations with None permission. It aborts if trying to access memory locations whose permissions are *not* None .

7.2 Adapting CompCert Compiler

Given the program $\mathbf{let}\ \{\gamma_1,\ldots,\gamma_l,\gamma_o\}\ \mathbf{in}\ \mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n$ consisting of Clight modules γ_i and the module γ_o (we omit sl and ge in the modules to simplify the presentation), the compilation Comp is defined as

$$\mathsf{Comp}(\mathbf{let}\ \{\gamma_1,\ldots,\gamma_l,\gamma_o\}\ \mathbf{in}\ \mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n) \stackrel{\text{def}}{=}$$
$$\mathbf{let}\ \{\mathsf{CompCert}(\gamma_1),\ldots,\mathsf{CompCert}(\gamma_l),\mathsf{IdTrans}(\gamma_o)\}\ \mathbf{in}\ \mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n$$

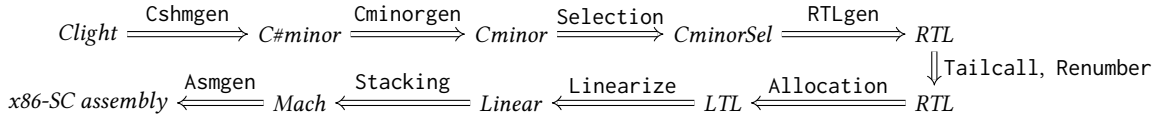


Figure 11. Proved CompCert Compilation Passes

```
Lemma sel_expr_correct:
 forall sp e m a v fp, Cminor.eval_expr sge sp e m a v ->
 Cminor.eval_expr_fp sge sp e m a fp ->
 forall e' le m', env_lessdef e e' -> Mem.extends m m' ->
 exists v', exists fp', FP.subset fp' fp /\
 eval_expr_fp tge sp e' m' le (sel_expr a) fp' /\
 eval_expr tge sp e' m' le (sel_expr a) v' /\ Val.lessdef v v'.
```

Figure 12. Coq code example

where CompCert is the adapted compiler consisting of the original CompCert passes, and IdTrans is the identity translation which returns the object module unchanged.

Lemma 11. Correct(CompCert, Clight, x86-SC).

We have proved Lemma 11, i.e. the original CompCert-3.0.1 passes satisfy our Correct in Def. 8. The verified compilation passes (shown in Fig. 11) include all the translation passes and two optimization passes (Tailcall and Renumber). Proving other optimization passes would be similar and is left as future work.

We also prove the well-definedness of Clight, x86-SC, and CImp, the determinism of x86-SC and CImp, and the correctness of IdTrans for CImp. Together with our framework's final theorem (Thm. 10), we proved the following result:

Theorem 12 (Correctness with x86-SC backend and obj.)**.** GCorrect((CompCert, Clight, x86-SC), (IdTrans, CImp, CImp)).

To prove Lemma 11, we try to *reuse as much the original CompCert correctness proofs as possible*. We address the following two main challenges in reusing CompCert proofs.

Converting memory layout. Many CompCert lemmas rely on the specific definition of the CompCert memory model, which is different from ours. In CompCert, memory allocations in an execution get *consecutive* natural numbers as block numbers. This fact is used extensively in CompCert's fundamental libraries and its compilation correctness proofs. But it does not hold in our model, where each thread has its own freelist F (an infinite sequence of block numbers). Since the F of different threads must be disjoint, we *cannot* make each F an infinite sequence of consecutive natural numbers to directly simulate CompCert.

Our solution is to define a bijection between memories under the two models. As a result, the behaviors of a thread under our model are equivalent to its behaviors under CompCert model, and our module-local simulation can be derived from a simulation based on the CompCert model. This way we reuse most CompCert libraries and compilation proofs without modification.

Footprint preservation. CompCert does not model footprints. Fortunately many of its definitions and lemmas can be slightly modified to support footprint preservation. For instance, Fig. 12 shows a key lemma in the proof of the Selection pass, sel_expr_correct, with our newly-added code highlighted in blue. It says the selected expression must evaluate to a value refined by the Cminor expression. We simply extend the lemma by requiring *the selected expression has smaller footprint while evaluating on related memory.*

7.3 x86-TSO as the Target Language

Now we show how to generate x86-TSO code while preserving the behaviors of the source program. The theorem below shows our final goal. To avoid clutter, below we use sl and tl to represent $sl_{\texttt{Clight}}$ and $tl_{\texttt{x86-TSO}}$ respectively.

Theorem 13 (Correctness with x86-TSO backend and obj.)**.** For any $f_1 \dots f_n, \Gamma = \{(sl, ge_1, \gamma_1), \dots, (sl, ge_m, \gamma_m), (sl_{\texttt{CImp}}, ge_o, \gamma_o)\}$, and $\Pi = \{(tl, ge'_1, \pi_1), \dots, (tl, ge'_m, \pi_m), (tl, ge_o, \pi_o)\}$, if

1. $\forall i \in \{1, \dots, m\}.\ (\texttt{CompCert.CodeT}(\gamma_i) = \pi_i) \ \wedge\ \text{injective}(\varphi)$
   $\wedge\ (\texttt{CompCert}.\varphi = \varphi) \ \wedge\ \varphi\{ge_i\} = ge'_i$,
2. $\text{Safe}(\textbf{let}\ \Gamma\ \textbf{in}\ \texttt{f}_1 \parallel \dots \parallel \texttt{f}_n)$ and $\text{DRF}(\textbf{let}\ \Gamma\ \textbf{in}\ \texttt{f}_1 \parallel \dots \parallel \texttt{f}_n)$,
3. $\forall i \in \{1, \dots, m\}.\ \text{ReachClose}(sl, ge_i, \gamma_i)$,
   and $\text{ReachClose}(sl_{\texttt{CImp}}, ge_o, \gamma_o)$,
4. $(tl, ge_o, \pi_o) \preccurlyeq^{\circ} (sl_{\texttt{CImp}}, ge_o, \gamma_o)$,

then $\quad \textbf{let}\ \Gamma\ \textbf{in}\ \texttt{f}_1 \parallel \dots \parallel \texttt{f}_n \sqsupseteq_{\varphi, ge_o} \textbf{let}\ \Pi\ \textbf{in}\ \texttt{f}_1 \parallel \dots \parallel \texttt{f}_n$.

Here the premises 1-3 are similar to those required in Def. 9. In addition, the premise 4 requires that the x86-TSO code π_o of the object be simulated by γ_o. The simulation $(tl, ge_o, \pi_o) \preccurlyeq^{\circ} (sl_{\texttt{CImp}}, ge_o, \gamma_o)$ is an extension of Liang and Feng [12] with the support of TSO semantics for the low-level code. Due to space limit, we omit the definition here.

The refinement relation $\mathbb{P} \sqsupseteq_{\varphi, ge_o} P$ is defined below:

$$\mathbb{P} \sqsupseteq_{\varphi, ge_o} P \ \text{ iff}$$
$$\forall \Sigma, \sigma.\ \text{initM}(\varphi, \text{GE}(\mathbb{P}.\Gamma), \Sigma, \sigma) \wedge \left(\forall l \in ge_o.\ \Sigma(l) = (\_, \text{None})\right) \implies (P, \sigma) \sqsubseteq' (\mathbb{P}, \Sigma)\,.$$

It is similar to $\sqsupseteq_{\varphi}$ defined at the end of Sec. 4, with the extra premise that the permission of the object data is None . As explained in Sec. 7.1, this enforces the partition between the object data and the client data. Also it uses a weaker version $\sqsubseteq'$ of trace refinement, which does not preserve termination (the formal definition omitted here). This is because our simulation $\preccurlyeq^{\circ}$ for the object code does not preserve termination for now, which we leave as future work.

Theorem 13 can be derived from Thm. 12 (for the compilation from Clight to x86-SC), and from Lemma 14 below, saying the x86-TSO code refines the x86-SC client code and the source object code (we use $tl_{\texttt{sc}}$ and $tl_{\texttt{tso}}$ as shorter notations for $tl_{\texttt{x86-SC}}$ and $tl_{\texttt{x86-TSO}}$ respectively).

Lemma 14 (Restore SC semantics for DRF x86 programs)**.** Let $\Pi_{\texttt{sc}} = \{(tl_{\texttt{sc}}, ge_1, \pi_1), \dots, (tl_{\texttt{sc}}, ge_m, \pi_m), (sl_{\texttt{CImp}}, ge_o, \gamma_o)\}$,

Library Code	Spec		Proof	
	CompCert	Ours	CompCert	Ours
Cshmgenproof.v	515	1021	1071	1498
Cminorgenproof.v	753	1557	1152	1251
Selectionproof.v	336	500	647	780
RTLgenproof.v	428	543	821	857
Tailcallproof.v	173	328	275	403
Renumberproof.v	86	245	117	356
Allocproof.v	704	785	1410	1696
Linearizeproof.v	236	371	349	732
Stackingproof.v	730	1038	1108	2131
Asmgenproof.v	208	338	571	1126
Compositionality (Lem. 4)		580		2249
DRF preservation (Lem. 6)		358		1142
Semantics equiv. (Lem. 7)		1529		4742
Lifting		828		1795

Figure 13. Lines of code (using `coqwc`) in Coq

and $\Pi_{\mathtt{tso}} = \{(tl_{\mathtt{tso}}, ge_1, \pi_1), \ldots, (tl_{\mathtt{tso}}, ge_m, \pi_m), (tl_{\mathtt{tso}}, ge_o, \pi_o)\}$. For any $f_1 \ldots f_n$, if

1. Safe(**let** $\Pi_{\mathtt{sc}}$ **in** $\mathtt{f}_1 \| \ldots \| \mathtt{f}_n$) and DRF(**let** $\Pi_{\mathtt{sc}}$ **in** $\mathtt{f}_1 \| \ldots \| \mathtt{f}_n$),
2. $(tl_{\mathtt{tso}}, ge_o, \pi_o) \preccurlyeq^{\circ} (sl_{\mathtt{CImp}}, ge_o, \gamma_o)$,

then **let** $\Pi_{\mathtt{sc}}$ **in** $\mathtt{f}_1 \| \ldots \| \mathtt{f}_n \sqsupseteq_{id, ge_o}$ **let** $\Pi_{\mathtt{tso}}$ **in** $\mathtt{f}_1 \| \ldots \| \mathtt{f}_n$.

As explained before, Lemma 14 can be viewed as a strengthened DRF-guarantee theorem for `x86-TSO` in that, if we let γ_o contain only skip and $ge_o = \emptyset$, Lemma 14 implies the DRF-guarantee of `x86-TSO`.

7.4 Proof Efforts in Coq

In Coq we have mechanized the framework and proved all the related lemmas shown in Fig. 2. We have also verified all the CompCert passes shown in Fig. 11. For the extended framework (Fig. 3), we give on-paper proofs for Theorem 13 and Lemma 14 in TR. The Coq proofs are still ongoing work.

Statistics of our Coq implementation and proofs are depicted in Fig. 13. Adapting the compilation correctness proofs from CompCert is relatively lightweight. For most passes our proofs are within 300 lines of code more than the original CompCert proofs. The `Stacking` pass introduces more additional proofs, mostly caused by arguments marshalling for supporting cross-language linking. In our experience, adapting CompCert's original compilation proofs to our settings takes less than one person week per translation pass (except for `Stacking`). For simpler passes such as `Tailcallproof.v`, `Linearizeproof.v`, `Allocproof.v`, and `RTLgenproof.v`, it takes less than one person day per pass.

By contrast, implementing our framework is more challenging, which took us about 1 person year. In particular, proving the equivalence between non-preemptive and preemptive semantics for DRF programs took us more time than expected, although it seems to be a well-known folklore theorem. The co-inductive proofs there involve a large number of non-trivial cases of reordering threads' executions.

8 Related Work

Compiler verification. Various work extends CompCert [9] to support separate compilation or concurrency. We have discussed Compositional CompCert [2, 20] in Sec. 1 and 2. SepCompCert [8] extends CompCert with the support of syntactical linking. Their approach requires all the compilation units be compiled by CompCert. They do not support cross-language linking or concurrency as we do.

CompCertTSO [18] compiles ClightTSO programs to the x86-TSO machine. It does not support cross-language linking, and its proof for the two CompCert passes `Stacking` and `Cminorgen` are not compositional. By contrast, we have verified these two passes using our compositional simulation. For the other compositional passes, CompCertTSO relies on a thread-local simulation, which is stronger than ours. It requires that the source and the target always generate the same memory events (excepts for those local variables that can be stored in registers). As a result, some optimizations (such as constant propagation and CSE) in CompCertTSO have to be more restrictive.

As an extension of CompCertTSO, Jagannathan et al. [6] allow the compiler to inject racy code such as the efficient spin lock in Fig. 10. They propose a refinement calculus on the racy code to ensure the compilation correctness. Their work looks similar to our extended framework in Fig. 3, but since they use TSO semantics for both the source and target programs, they do not need to handle the gap between the SC and TSO semantics, so they do not need the source to be DRF as in our work.

Vellvm [25, 26] proves correctness of several optimization passes for *sequential* LLVM programs. Perconti and Ahmed [16] verify separate compilation by embedding languages in a combined language. They do not support concurrency either. Ševčík [17] studies safety of a class of optimizations in concurrent settings using an abstract trace semantics. It is unclear if his approach can be applied to verify general compilation. Lochbihler [13] verifies a compiler for concurrent Java programs. His simulation has similar restrictions as CompCertTSO.

Non-preemptive semantics and data-race-freedom. Non-preemptive (or cooperative) semantics has been developed in various settings for various purposes (e.g., [1, 4, 11, 22, 24]). Both Ferreira et al. [5] and Xiao et al.[23] study the relationships between non-preemptive semantics and DRF, but they do not give any mechanized proofs of termination-preserving semantics equivalence as in our work. DRFx [14] proposes a concept called Region-Conflict-Freedom, which looks similar to our NPDRF, but there is no formal operational formulation as we do. Owens [15] proposes Triangular-Race-Freedom (TRF) and proves that TRF programs behaves the same in `x86-SC` and `x86-TSO`. TRF is weaker than DRF and can be satisfied by the efficient spin lock code in Fig. 10.

References

[1] Martin Abadi and Gordon Plotkin. 2009. A Model of Cooperative Threads. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '09)*. ACM, New York, NY, USA, 29–40.

[2] Lennart Beringer, Gordon Stewart, Robert Dockins, and Andrew W. Appel. 2014. Verified Compilation for Shared-Memory C. In *ESOP*. 107–127.

[3] Hans-Juergen Boehm. 2011. How to Miscompile Programs with "Benign" Data Races. In *Proc. HotPar'11*.

[4] Gérard Boudol. 2007. Fair Cooperative Multithreading. In *CONCUR 2007 – Concurrency Theory*, Luís Caires and Vasco T. Vasconcelos (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 272–286.

[5] Rodrigo Ferreira, Xinyu Feng, and Zhong Shao. 2010. Parameterized Memory Models and Concurrent Separation Logic. In *ESOP*, Andrew D. Gordon (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 267–286.

[6] Suresh Jagannathan, Vincent Laporte, Gustavo Petri, David Pichardie, and Jan Vitek. 2014. Atomicity Refinement for Verified Compilation. *ACM Trans. Program. Lang. Syst.* 36, 2 (2014), 6:1–6:30.

[7] Cliff B. Jones. 1983. Tentative Steps Toward a Development Method for Interfering Programs. *ACM Trans. Program. Lang. Syst.* 5, 4 (1983), 596–619.

[8] Jeehoon Kang, Yoonseung Kim, Chung-Kil Hur, Derek Dreyer, and Viktor Vafeiadis. 2016. Lightweight Verification of Separate Compilation. In *POPL*. 178–190.

[9] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115.

[10] Xavier Leroy. 2009. A Formally Verified Compiler Back-end. *J. Autom. Reason.* 43 (December 2009), 363–446. Issue 4.

[11] Peng Li and Steve Zdancewic. 2007. Combining Events and Threads for Scalable Network Services Implementation and Evaluation of Monadic, Application-level Concurrency Primitives. In *PLDI '07*. 189–199.

[12] Hongjin Liang and Xinyu Feng. 2013. Modular Verification of Linearizability with Non-Fixed Linearization Points. In *PLDI*. 459–470.

[13] Andreas Lochbihler. 2010. Verifying a Compiler for Java Threads. In *ESOP*. 427–447.

[14] Daniel Marino, Abhayendra Singh, Todd Millstein, Madanlal Musuvathi, and Satish Narayanasamy. 2010. DRFX: A Simple and Efficient Memory Model for Concurrent Programming Languages. In *PLDI '10*. 351–362.

[15] Scott Owens. 2010. Reasoning about the Implementation of Concurrency Abstractions on x86-TSO. In *ECOOP*. 478–503.

[16] James T. Perconti and Amal Ahmed. 2014. Verifying an Open Compiler Using Multi-language Semantics. In *Programming Languages and Systems*, Zhong Shao (Ed.). 128–148.

[17] Jaroslav Ševčík. 2011. Safe optimisations for shared-memory concurrent programs. In *PLDI*. 306–316.

[18] Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. 2013. CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency. *J. ACM* 60, 3 (2013), 22.

[19] 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 (2010), 89–97.

[20] Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. 2015. Compositional CompCert. In *POPL*. 275–287.

[21] CompCert Developers. 2017. CompCert-3.0.1. http://compcert.inria.fr/release/compcert-3.0.1.tgz

[22] Jérôme Vouillon. 2008. Lwt: A Cooperative Thread Library. In *ML*. 3–12.

[23] Siyang Xiao, Hanru Jiang, Hongjin Liang, and Xinyu Feng. 2018. Non-Preemptive Semantics for Data-Race-Free Programs. In *ICTAC*. to appear.

$$\frac{T(\mathtt{t}) = (tl, F, \kappa) \quad F \vdash (\kappa, \sigma) \xmapsto[\mathsf{emp}]{e} (\kappa', \sigma) \quad T' = T\{\mathtt{t} \rightsquigarrow (tl, F, \kappa')\}}{(T, \mathtt{t}, 0, \sigma) \xRightarrow[\mathsf{emp}]{e} (T', \mathtt{t}, 0, \sigma)}\ \text{Print}$$

$$\frac{T(\mathtt{t}) = (tl, F, \kappa) \quad \mathtt{t}' \in \mathsf{dom}(T\backslash\mathtt{t}) \quad F \vdash (\kappa, \sigma) \xmapsto[\mathsf{emp}]{\mathtt{ret}} (\kappa', \sigma)}{(T, \mathtt{t}, 0, \sigma) \xRightarrow[\mathsf{emp}]{\mathtt{sw}} (T\backslash\mathtt{t}, \mathtt{t}', 0, \sigma)}\ \text{Term}$$

$$\frac{T(\mathtt{t}) = (tl, F, \kappa) \quad \mathsf{dom}(T) = \{\mathtt{t}\} \quad F \vdash (\kappa, \sigma) \xmapsto[\mathsf{emp}]{\mathtt{ret}} (\kappa', \sigma)}{(T, \mathtt{t}, 0, \sigma) \xRightarrow[\mathsf{emp}]{\tau} \mathbf{done}}\ \text{Done}$$

$$\frac{T(\mathtt{t}) = (tl, F, \kappa) \qquad F \vdash (\kappa, \sigma) \xmapsto[\delta]{} \mathbf{abort}}{(T, \mathtt{t}, 0, \sigma) \xRightarrow[\delta]{\tau} \mathbf{abort}}\ \text{Abort}$$

Figure 14. More Rules of the Preemptive Global Semantics

[24] Jaeheon Yi, Caitlin Sadowski, and Cormac Flanagan. 2011. Cooperative Reasoning for Preemptive Execution. In *PPoPP*. 147–156.

[25] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. 2012. Formalizing the LLVM Intermediate Representation for Verified Program Transformations. In *POPL*. 427–440.

[26] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. 2013. Formal Verification of SSA-based Optimizations for LLVM. In *PLDI*. 175–186.

A More about the Language Settings

Global preemptive semantics. Fig. 14 gives the omitted rules of the global preemptive semantics.

The Print rule shows the step generating an external event. It requires that the flag d must be 0, i.e., external events can be generated only outside of atomic blocks. Also the step generating an external event does *not* access memory, so its footprint is empty (emp).

When the current thread terminates, we either remove it from the thread pool and then switch to another thread if there is one (see the Term rule), or terminate the whole program if not, as shown in the Done rule. The Term rule generates the sw message to record the context switch.

The Abort rule says the whole program aborts if a local module aborts.

We write $F \vdash \phi \xmapsto[\delta]{\tau}{}^{*} \phi'$ for zero or multiple silent steps, where δ is the accumulation of the footprint of each step. $F \vdash \phi \xmapsto[\delta]{\tau}{}^{+} \phi'$ represents at least one step. Similar notations are used for global steps. We also write $W \Rightarrow^{+} W'$ for multiple steps that either are silent or produce sw events. It must contain at least one silent step. $W \overset{e}{\Rightarrow}{}^{+} W'$ represents multiple steps with exactly one e event produced (while other steps either are silent or produce sw events).

Event-trace refinement and equivalence. Below we give the omitted definitions of event-trace refinement and equivalence.

$$PEtr((P,\sigma),\mathcal{B}) \text{ iff } \exists W.\ ((P,\sigma) \overset{load}{\Longrightarrow} W) \wedge Etr(W,\mathcal{B})$$

$$\frac{W \Rightarrow^{+} \mathbf{abort}}{Etr(W,\mathbf{abort})} \qquad \frac{W \overset{e}{\Rightarrow}^{+} W' \qquad Etr(W',\mathcal{B})}{Etr(W,e::\mathcal{B})}$$

$$\frac{W \Rightarrow^{+} \mathbf{done}}{Etr(W,\mathbf{done})} \qquad \frac{W \Rightarrow^{+} W' \qquad Etr(W',\epsilon)}{Etr(W,\epsilon)}$$

The co-inductive definition of $Etr(W,\mathcal{B})$ says that $\mathcal{B}$ can be produced by executing W. Note we distinguish the traces of non-terminating (diverging) executions from those of terminating ones. If the execution of W diverges, its observable event trace $\mathcal{B}$ is either of infinitely length, or finite but does not end with **done** or **abort** (called *silent divergence*, see the right-most rule above).

Then we define the refinement $(\mathbb{P},\Sigma) \sqsupseteq (P,\sigma)$ and the equivalence $(\mathbb{P},\Sigma) \approx (P,\sigma)$ below. They ensure that if (P,σ) has a diverging execution, so does $(\mathbb{P},\Sigma)$. Thus the refinement and the equivalence relations preserve termination.

Definition 15 (Event-Trace Refinement and Equivalence)**.**
$(\mathbb{P},\Sigma) \sqsupseteq (P,\sigma)$ iff $\forall\mathcal{B}.\ PEtr((P,\sigma),\mathcal{B}) \implies PEtr((\mathbb{P},\Sigma),\mathcal{B})$.
$(\mathbb{P},\Sigma) \approx (P,\sigma)$ iff $\forall\mathcal{B}.\ PEtr((P,\sigma),\mathcal{B}) \iff PEtr((\mathbb{P},\Sigma),\mathcal{B})$.

Safety. Below we use the event traces to define Safe($\mathbb{W}$) and Safe($\mathbb{P},\Sigma$).

Definition 16 (Safety)**.** Safe($\mathbb{W}$) iff $\neg\exists tr.\ Etr(\mathbb{W}, tr::\mathbf{abort})$.

Safe($\mathbb{P},\Sigma$) iff $(\exists\mathbb{W}.\ (\mathbb{P},\Sigma) \overset{load}{\Longrightarrow} \mathbb{W})$ and $\forall\mathbb{W}.\ ((\mathbb{P},\Sigma) \overset{load}{\Longrightarrow} \mathbb{W}) \implies$ Safe($\mathbb{W}$).

Non-preemptive semantics. Fig. 15 gives the omitted non-preemptive semantics rules. Like EntAt_{np} and ExtAt_{np}, the rules Print_{np} and Term_{np} execute one step of the current thread t, and then non-deterministically switch to a thread t′ (which could just be t). The corresponding global steps produce the sw events (or the external event e in the (Print_{np}) rule). Other rules are very similar to their counterparts in the preemptive semantics.

B More about the Simulation

The formal definition of ReachClose. The module-local simulations could be composed to derive the whole-program simulation by proving the Rely condition of a module is guaranteed by other modules' guarantee conditions HG and LG. However, in our module-local simulation, LG is established if the source module satisfies HG. We require that HG always hold during the execution of the source, and formulate this requirement as ReachClose in Def. 17. It is a simplified version of the *reach-close* concept by Stewart et al. [20] (simplified because we do not allow the leak of local stack pointers into the shared memory).

Definition 17 (Reach Closed Module)**.** ReachClose(sl, ge, γ) iff , for all f, $\Bbbk$, Σ, $\mathbb{F}$ and $\mathbb{S}$, if sl.InitCore(γ, f) = $\Bbbk$, $\mathbb{S}$ = dom(Σ), $ge \subseteq \mathbb{S}$, $\mathbb{F} \cap \mathbb{S} = \emptyset$, and closed($\mathbb{S},\Sigma$), then RC($\mathbb{F},\mathbb{S},(\Bbbk,\Sigma)$).

Here RC is defined as the largest relation such that, whenever RC($\mathbb{F},\mathbb{S},(\Bbbk,\Sigma)$), then for all Σ' such that R($\Sigma,\Sigma',\mathbb{F},\mathbb{S}$),

$$\frac{\begin{array}{l}\text{for all } i \text{ and } j \text{ in } \{1,\ldots,n\}, \text{ and } i \neq j:\\ \quad F_i \cap F_j = \emptyset \quad \text{dom}(\sigma) \cap F_i = \emptyset\\ \quad tl_i.\text{InitCore}(\pi_i, \mathtt{f}_i) = \kappa_i, \quad \text{where } (tl_i, ge_i, \pi_i) \in \Pi\\ T = \{1 \rightsquigarrow (tl_1, F_1, \kappa_1), \ldots, n \rightsquigarrow (tl_n, F_n, \kappa_n)\}\\ \mathtt{t} \in \{1,\ldots,n\} \quad \mathbb{d} = \{\mathtt{t}_1 \rightsquigarrow 0, \ldots, \mathtt{t}_n \rightsquigarrow 0\}\end{array}}{(\mathbf{let}\ \Pi\ \mathbf{in}\ \mathtt{f}_1 \mid \ldots \mid \mathtt{f}_n, \sigma) \overset{load}{:\Longrightarrow} (T, \mathtt{t}, \mathbb{d}, \sigma)}\ \text{Load}_{np}$$

$$\frac{T(\mathtt{t}) = (tl, F, \kappa) \qquad F \vdash (\kappa,\sigma) \overset{\tau}{\underset{\delta}{\mapsto}} (\kappa',\sigma') \qquad T' = T\{\mathtt{t} \rightsquigarrow (tl, F, \kappa')\}}{(T, \mathtt{t}, \mathbb{d}, \sigma) \overset{\tau}{\underset{\delta}{:\Longrightarrow}} (T', \mathtt{t}, \mathbb{d}, \sigma')}\ \tau\text{-step}_{np}$$

$$\frac{T(\mathtt{t}) = (tl, F, \kappa) \qquad \mathbb{d}(\mathtt{t}) = 0 \qquad \mathtt{t}' \in \text{dom}(T) \qquad F \vdash (\kappa,\sigma) \overset{e}{\underset{\text{emp}}{\mapsto}} (\kappa',\sigma) \qquad T' = T\{\mathtt{t} \rightsquigarrow (tl, F, \kappa')\}}{(T, \mathtt{t}, \mathbb{d}, \sigma) \overset{e}{\underset{\text{emp}}{:\Longrightarrow}} (T', \mathtt{t}', \mathbb{d}, \sigma)}\ \text{Print}_{np}$$

$$\frac{T(\mathtt{t}) = (tl, F, \kappa) \qquad \mathbb{d}(\mathtt{t}) = 0 \qquad F \vdash (\kappa,\sigma) \overset{\mathtt{ret}}{\underset{\text{emp}}{\longmapsto}} (\kappa',\sigma) \qquad \mathtt{t}' \in \text{dom}(T \backslash \mathtt{t})}{(T, \mathtt{t}, \mathbb{d}, \sigma) \overset{\mathsf{sw}}{\underset{\text{emp}}{:\Longrightarrow}} (T \backslash \mathtt{t}, \mathtt{t}', \mathbb{d} \backslash \mathtt{t}, \sigma)}\ \text{Term}_{np}$$

$$\frac{T(\mathtt{t}) = \{\mathtt{t} \rightsquigarrow (tl, F, \kappa)\} \qquad \mathbb{d} = \{\mathtt{t} \rightsquigarrow 0\} \qquad F \vdash (\kappa,\sigma) \overset{\mathtt{ret}}{\underset{\text{emp}}{\longmapsto}} (\kappa',\sigma)}{(T, \mathtt{t}, \mathbb{d}, \sigma) \overset{\tau}{\underset{\text{emp}}{:\Longrightarrow}} \mathbf{done}}\ \text{Done}_{np}$$

$$\frac{T(\mathtt{t}) = (tl, F, \kappa) \qquad F \vdash (\kappa,\sigma) \underset{\delta}{\longmapsto} \mathbf{abort}}{(T, \mathtt{t}, \mathbb{d}, \sigma) \overset{\tau}{\underset{\delta}{:\Longrightarrow}} \mathbf{abort}}\ \text{Abort}_{np}$$

Figure 15. More Rules of the Non-Preemptive Global Semantics

and for all $\Bbbk'$, Σ', Σ'', ι and Δ such that $\mathbb{F} \vdash (\Bbbk,\Sigma') \overset{\iota}{\underset{\Delta}{\mapsto}} (\Bbbk',\Sigma'')$, we have HG($\Delta,\Sigma'',\mathbb{F},\mathbb{S}$), and RC($\mathbb{F},\mathbb{S},(\Bbbk',\Sigma'')$).

The relation RC($\mathbb{F},\mathbb{S},(\Bbbk,\Sigma)$) essentially says during every step of the execution of ($\Bbbk,\Sigma$), HG always holds over the resulting footprints Δ and states, even with possible interference from the environment, as long as the environment steps satisfy the rely condition R defined in Fig. 8.

Transitivity. Our simulation is transitive. One can decompose the whole compiler correctness proofs into proofs for individual compilation passes.

Lemma 18 (Transitivity)**.** $\forall sl, sl', tl, \gamma, \gamma', \pi.$
if $(sl, ge, \gamma) \preccurlyeq_{\varphi} (sl', ge', \gamma')$ and $(sl', ge', \gamma') \preccurlyeq_{\varphi'} (tl, ge'', \pi)$, then $(sl, ge, \gamma) \preccurlyeq_{\varphi' \circ \varphi} (tl, ge'', \pi)$.

Proof of Lemma 4. Proof of Lemma 4 relies on the following non-interference lemma.

Lemma 19 (Non-Interference). For any $\Bbbk, \Bbbk'$ of some well-defined language *sl*, and any κ, κ' of some well-defined language *tl*, if $\mathbb{F}_1 \vdash (\Bbbk, \Sigma_0) \xmapsto[\Delta_0]{\iota}{}^{+}(\Bbbk', \Sigma)$, and $F_1 \vdash (\kappa, \sigma_0) \xmapsto[\delta_0]{\iota} {}^{+}(\kappa', \sigma)$, then for any $\mu, \mathbb{F}_2$ and F_2 such that $(\mu.\mathbb{S} \cup \mathbb{F}_1) \cap \mathbb{F}_2 = \emptyset$, and $(\mu.S \cup F_1) \cap F_2 = \emptyset$, we have

$$\mathsf{HG}(\Delta_0, \Sigma, \mathbb{F}_1, \mu.\mathbb{S}) \wedge \mathsf{LG}(\mu, (\delta_0, \sigma, F_1), (\Delta_0, \Sigma)) \implies \mathsf{Rely}(\mu, (\Sigma_0, \Sigma, \mathbb{F}_2), (\sigma_0, \sigma, F_2)) .$$

Here we use $\mathbb{F} \vdash (\Bbbk, \Sigma_0) \xmapsto[\Delta_0]{\iota}{}^{+}(\Bbbk', \Sigma)$ to say:

either $\mathbb{F} \vdash (\Bbbk, \Sigma_0) \xmapsto[\Delta_0]{\tau}{}^{+}(\Bbbk', \Sigma)$,

or $\mathbb{F} \vdash (\Bbbk, \Sigma_0) \xmapsto[\Delta_0]{\tau}{}^{*}(\Bbbk'', \Sigma)$ and $\mathbb{F} \vdash (\Bbbk'', \Sigma) \xmapsto[\mathsf{emp}]{\iota} (\Bbbk', \Sigma)$ for some $\Bbbk''$, and $\iota \neq \tau$.

The non-interference lemma says, after the execution of $(\Bbbk, \Sigma_0)$ and (κ, σ_0) until some interaction point with resulting states Σ and σ, we are able to establish the Rely condition over the initial and resulting states with respect to μ and some freelists $\mathbb{F}_2$ and F_2, if the freelists $\mathbb{F}_2$ and F_2 contains no shared location and are disjoint with the freelists $\mathbb{F}_1$ and F_1 respectively, and the execution guarantees HG and LG. Here LG is established by our module-local simulation if HG is guaranteed, and HG is established by RC by the following lemma, which says we can establish HG for multiple steps if the module and corresponding state satisfies RC.

Lemma 20 (RC-guarantee). If $\mathsf{RC}(\mathbb{S}, \mathbb{F}, (\Bbbk, \Sigma_0))$ and $\mathbb{F} \vdash (\Bbbk, \Sigma_0) \xmapsto[\Delta_0]{\iota}{}^{+}(\Bbbk', \Sigma)$ then $\mathsf{HG}(\Delta_0, \Sigma, \mathbb{F}, \mathbb{S})$.

The non-preemptive global downward simulation. The definition of the whole program simulation is similar to the module local simulation, except the case for environment interference (Rely steps), which is unnecessary for whole program simulation. Every source step should correspond to multiple target steps. As in module local simulation, we always require the target footprints matches those of the source, defined as FPmatch. Also the footprint should always be in scope. As a special requirement for the whole program simulation, we always require the source and the target do lock-step context switch and they always switch to the same thread.

Below we use $\mathsf{curF}(\widehat{W})$ for the freelist of the current thread, i.e., $\mathsf{curF}((\mathbb{T}, \mathtt{t}, \_, \_)) \stackrel{\text{def}}{=} \mathbb{T}(\mathtt{t}).\mathbb{F}$.

Definition 21 (Whole-Program Downward Simulation). Let $\hat{\mathbb{P}} = \textbf{let } \Gamma \textbf{ in } \mathtt{f}_1 \mid \ldots \mid \mathtt{f}_m$, $\hat{P} = \textbf{let } \Pi \textbf{ in } \mathtt{f}_1 \mid \ldots \mid \mathtt{f}_m$, $\Gamma = \{(sl_1, ge_1, \gamma_1), \ldots, (sl_m, ge_m, \gamma_m)\}$ and $\Pi = \{(tl_1, ge'_1, \pi_1), \ldots, (tl_m, ge'_m, \pi_m)\}$.
We say $\hat{\mathbb{P}} \preccurlyeq_\varphi \hat{P}$ iff $\forall 1 \leq i \leq m.\ \varphi\{ge_i\} = ge'_i$, and for any $\Sigma, \sigma, \mu, \widehat{\mathbb{W}}$, if $\mathsf{initM}(\varphi, \bigcup ge_i, \Sigma, \sigma)$, $\mu = (\mathsf{dom}(\Sigma), \mathsf{dom}(\sigma), \varphi)$, and $(\hat{\mathbb{P}}, \Sigma) \overset{load}{:\!\!\Longrightarrow} \widehat{\mathbb{W}}$, then there exists $\widehat{W}$, $i \in \mathsf{index}$ such that $(\hat{P}, \sigma) \overset{load}{:\!\!\Longrightarrow} \widehat{W}$ and $(\widehat{\mathbb{W}}, \mathsf{emp}) \preccurlyeq^i_\mu (\widehat{W}, \mathsf{emp})$.

Here we define $(\widehat{\mathbb{W}}, \Delta_0) \preccurlyeq^i_\mu (\widehat{W}, \delta_0)$ as the largest relation such that whenever $(\widehat{\mathbb{W}}, \Delta_0) \preccurlyeq^i_\mu (\widehat{W}, \delta_0)$, then the following are true:

1. $\widehat{\mathbb{W}}.\mathtt{t} = \widehat{W}.\mathtt{t}$, and $\widehat{\mathbb{W}}.\mathbb{d} = \widehat{W}.\mathbb{d}$;
2. $\forall \widehat{\mathbb{W}}', \Delta.$ if $\widehat{\mathbb{W}} \overset{\tau}{\underset{\Delta}{:\!\!\Longrightarrow}} \widehat{\mathbb{W}}'$ then one of the following holds:
   a. $\exists j.\ j < i$, and $(\widehat{\mathbb{W}}', \Delta_0 \cup \Delta) \preccurlyeq^j_\mu (\widehat{W}, \delta_0)$; or
   b. $\exists \widehat{W}', \delta, j.$
      i. $\widehat{W} \overset{\tau}{\underset{\delta}{\Longrightarrow}}{}^{+} \widehat{W}'$, and
      ii. $(\delta_0 \cup \delta) \subseteq (\mathsf{curF}(\widehat{W}) \cup \mu.S)$, and $\mathsf{FPmatch}(\mu, \Delta_0 \cup \Delta, \delta_0 \cup \delta)$; and
      iii. $(\widehat{\mathbb{W}}', \Delta_0 \cup \Delta) \preccurlyeq^j_\mu (\widehat{W}', \delta_0 \cup \delta)$;
3. $\forall \mathbb{T}', \mathbb{d}', \Sigma', o, \mathtt{t}'.$ if $o \neq \tau$ and $\widehat{\mathbb{W}} \overset{o}{\underset{\mathsf{emp}}{:\!\!\Longrightarrow}} (\mathbb{T}', \mathtt{t}', \mathbb{d}', \Sigma')$, then
   $\exists \widehat{W}', \delta, T', \sigma', j.$ for any $\mathtt{t}'' \in \mathsf{dom}(\mathbb{T}')$, we have
   a. $\widehat{W} \overset{\tau}{\underset{\delta}{\Longrightarrow}}{}^{*} \widehat{W}'$, and $\widehat{W}' \overset{o}{\underset{\mathsf{emp}}{:\!\!\Longrightarrow}} (T', \mathtt{t}'', \mathbb{d}', \sigma')$, and
   b. $(\delta_0 \cup \delta) \subseteq \mathsf{curF}(\widehat{W}) \cup \mu.S$, and $\mathsf{FPmatch}(\mu, \Delta_0, \delta_0 \cup \delta)$; and
   c. $((\mathbb{T}', \mathtt{t}'', \mathbb{d}', \Sigma'), \mathsf{emp}) \preccurlyeq^j_\mu ((T', \mathtt{t}'', \mathbb{d}', \sigma'), \mathsf{emp})$;
4. if $\widehat{\mathbb{W}} \overset{\tau}{\underset{\mathsf{emp}}{:\!\!\Longrightarrow}} \textbf{done}$, then $\exists \widehat{W}', \delta.$
   a. $\widehat{W} \overset{\tau}{\underset{\delta}{\Longrightarrow}}{}^{*} \widehat{W}'$, and $\widehat{W}' \overset{\tau}{\underset{\mathsf{emp}}{:\!\!\Longrightarrow}} \textbf{done}$, and
   b. $(\delta_0 \cup \delta) \subseteq \mathsf{curF}(\widehat{W}) \cup \mu.S$, and $\mathsf{FPmatch}(\mu, \Delta_0, \delta_0 \cup \delta)$.

The condition on context switches (lock step, and to the same thread) in our whole-program simulations is not so strong as it sounds. It can be naturally derived from composing our module-local simulations. In our module-local simulation, we already have lock-step EntAtom/ExtAtom steps (see case 2 in Def. 3), which will be switch points in whole programs. Besides, since the local simulation is supposed to relate the source and target of the same thread, it seems natural to require the source and target to always switch to the same thread in the whole-program simulation.

The switching-to-the-same-thread condition makes it easy to flip the whole-program simulation (i.e. derive the whole-program upward simulation from the downward one). The upward and downward simulations require us to map each target thread to some source thread and also map each source thread to a target one. Since the number of target threads is the same as the number of source threads, we would need to require the mapping between the source and target threads to be a bijection. The easiest way to ensure the bijection is to let the thread identifiers to be the same.

The non-preemptive global upward simulation. We define the whole program upward simulation as $\hat{P} \leqslant_\varphi \hat{\mathbb{P}}$ in Def. 22. It is similar to the downward simulation $\hat{\mathbb{P}} \preccurlyeq_\varphi \hat{P}$, with the positions of source and target swapped. Note that we do *not* flip the FPmatch condition, since we always require footprint of target program being a refinement of the

footprint of the source program, in order to prove DRF of the target program. Correspondingly, the address mapping φ is *not* flipped either.

Definition 22 (Whole-Program Upward Simulation).
$\hat{P} \leqslant_\varphi \hat{\mathbb{P}}$ iff
there exist $\mathtt{f}_1, \ldots, \mathtt{f}_n, \Gamma = \{(sl_1, ge_1, \gamma_1), \ldots, (sl_m, \gamma_m)\}$ and $\Pi = \{(tl_1, ge'_1, \pi_1), \ldots, (tl_m, \pi_m)\}$, such that $\hat{\mathbb{P}} = \mathbf{let}\ \Gamma\ \mathbf{in}\ \mathtt{f}_1 \mid \ldots \mid \mathtt{f}_m$, $\hat{P} = \mathbf{let}\ \Pi\ \mathbf{in}\ \mathtt{f}_1 \mid \ldots \mid \mathtt{f}_m$, $\varphi\{ge_i\} = ge'_i$ and
for any $\Sigma, \sigma, \mu, \widehat{\mathbb{W}}$, if $\mathsf{initM}(\varphi, ge, \Sigma, \sigma)$, and
$\mu = (\mathsf{dom}(\Sigma), \mathsf{dom}(\sigma), \varphi)$, and $(\hat{P}, \sigma) :\overset{load}{\Longrightarrow} \widehat{W}$,
then there exists $\widehat{\mathbb{W}}$, $i \in \mathtt{index}$ such that $(\hat{\mathbb{P}}, \Sigma) :\overset{load}{\Longrightarrow} \widehat{\mathbb{W}}$ and $(\widehat{\mathbb{W}}, \mathsf{emp}) \preccurlyeq^i_\mu (\widehat{\mathbb{W}}, \mathsf{emp})$.

Here we define $(\widehat{W}, \Delta_0) \leqslant^i_\mu (\widehat{\mathbb{W}}, \delta_0)$ as the largest relation such that whenever $(\widehat{W}, \Delta_0) \leqslant^i_\mu (\widehat{\mathbb{W}}, \delta_0)$, then the following are true:

1. $\widehat{\mathbb{W}}.\mathtt{t} = \widehat{W}.\mathtt{t}$, and $\widehat{\mathbb{W}}.\mathbb{d} = \widehat{W}.\mathbb{d}$;
2. $\forall \widehat{W}', \delta.$ if $\widehat{W} :\overset{\tau}{\underset{\delta}{\Rightarrow}} \widehat{W}'$, then one of the following holds:
   a. $\exists j.\ j < i$, and $(\widehat{W}', \Delta_0 \cup \Delta) \leqslant^j_\mu (\widehat{\mathbb{W}}, \delta)$;
   b. $\exists \widehat{\mathbb{W}}', \Delta, j.$
      i. $\widehat{\mathbb{W}} :\overset{\tau}{\underset{\Delta}{\Rightarrow}}{}^{+} \widehat{\mathbb{W}}'$, and
      ii. $(\delta_0 \cup \delta) \subseteq \mathsf{curF}(\widehat{W}) \cup \mu.S$, and $\mathsf{FPmatch}(\mu, \Delta_0 \cup \Delta, \delta_0 \cup \delta)$; and
      iii. $(\widehat{W}', \mathsf{emp}) \leqslant^j_\mu (\widehat{\mathbb{W}}', \mathsf{emp})$;
3. $\forall T', \mathbb{d}', \sigma', o, \mathtt{t}'.$ if $o \neq \tau$ and $\widehat{W} :\overset{o}{\underset{\mathsf{emp}}{\Longrightarrow}} (T', \mathtt{t}', \mathbb{d}', \sigma')$, then
   $\exists \widehat{\mathbb{W}}', \Delta, \mathbb{T}', \Sigma', j.$ for any $\mathtt{t}'' \in \mathsf{dom}(T')$, we have
   a. $\widehat{\mathbb{W}} :\overset{\tau}{\underset{\Delta}{\Rightarrow}}{}^{*} \widehat{\mathbb{W}}'$, and $\widehat{\mathbb{W}}' :\overset{o}{\underset{\mathsf{emp}}{\Longrightarrow}} (\mathbb{T}', \mathtt{t}'', \mathbb{d}', \Sigma')$, and
   b. $\delta_0 \subseteq \mathsf{curF}(\widehat{W}) \cup \mu.S$, and $\mathsf{FPmatch}(\mu, \Delta_0 \cup \Delta, \delta_0)$; and
   c. $((T', \mathtt{t}'', \mathbb{d}', \sigma'), \mathsf{emp}) \leqslant^j_\mu ((\mathbb{T}', \mathtt{t}'', \mathbb{d}', \Sigma'), \mathsf{emp})$;
4. if $\widehat{W} :\overset{\tau}{\underset{\mathsf{emp}}{\Longrightarrow}} \mathbf{done}$, then $\exists \widehat{\mathbb{W}}', \Delta.$
   a. $\widehat{\mathbb{W}} :\overset{\tau}{\underset{\Delta}{\Rightarrow}}{}^{*} \widehat{\mathbb{W}}'$, and $\widehat{\mathbb{W}}' :\overset{\tau}{\underset{\mathsf{emp}}{\Longrightarrow}} \mathbf{done}$, and
   b. $\delta_0 \subseteq \mathsf{curF}(\widehat{W}) \cup \mu.S$, and $\mathsf{FPmatch}(\mu, \Delta_0 \cup \Delta, \delta_0)$

Determinism of the target language, and flip of the global simulation.

Definition 23 (Deterministic Languages). $\det(tl)$ iff, for all configuration ϕ in tl (see the definition of ϕ in Fig. 4), and for all F, $F \vdash \phi \overset{\iota_1}{\underset{\delta_1}{\mapsto}} \phi_1 \wedge F \vdash \phi \overset{\iota_2}{\underset{\delta_2}{\mapsto}} \phi_2 \implies \phi_1 = \phi_2 \wedge \iota_1 = \iota_2 \wedge \delta_1 = \delta_2$.

Lemma 24 (Flip, ④ in Fig. 2).
For any $\mathtt{f}_1, \ldots, \mathtt{f}_n, ge, \varphi, \Gamma = \{(sl_1, ge_1, \gamma_1), \ldots, (sl_m, ge_m, \gamma_m)\}$, $\Pi = \{(tl_1, ge'_1, \pi_1), \ldots, (tl_m, ge'_m, \pi_m)\}$, if $\forall i.\ \det(tl_i)$, and
$\mathbf{let}\ \Gamma\ \mathbf{in}\ \mathtt{f}_1 \mid \ldots \mid \mathtt{f}_m \preccurlyeq_\varphi \mathbf{let}\ \Pi\ \mathbf{in}\ \mathtt{f}_1 \mid \ldots \mid \mathtt{f}_m$,
then $\mathbf{let}\ \Pi\ \mathbf{in}\ \mathtt{f}_1 \mid \ldots \mid \mathtt{f}_m \leqslant_\varphi \mathbf{let}\ \Gamma\ \mathbf{in}\ \mathtt{f}_1 \mid \ldots \mid \mathtt{f}_m$.

$$\frac{(\hat{P}, \sigma) :\overset{load}{\Longrightarrow} \widehat{W} \qquad \widehat{W} :\Rightarrow^* \widehat{W}' \qquad \widehat{W}' :\Longmapsto \mathsf{Race}}{(\hat{P}, \sigma) :\Longmapsto \mathsf{Race}}$$

$$\frac{(\hat{P}, \sigma) :\overset{load}{\Longrightarrow} \widehat{W} \quad \mathsf{NPpred}(\widehat{W}, \mathtt{t}_1, (\delta_1, d_1)) \quad \mathtt{t}_1 \neq \mathtt{t}_2 \quad \mathsf{NPpred}(\widehat{W}, \mathtt{t}_2, (\delta_2, d_2)) \quad (\delta_1, d_1) \frown (\delta_2, d_2)}{(\hat{P}, \sigma) :\Longmapsto \mathsf{Race}}$$

$$\frac{\widehat{W} :\overset{o}{\underset{\mathsf{emp}}{\Longrightarrow}} \widehat{W}' \quad o \neq \tau \quad \mathtt{t}_1 \neq \mathtt{t}_2 \quad (\delta_1, d_1) \frown (\delta_2, d_2) \quad \mathsf{NPpred}(\widehat{W}', \mathtt{t}_1, (\delta_1, d_1)) \quad \mathsf{NPpred}(\widehat{W}', \mathtt{t}_2, (\delta_2, d_2))}{\widehat{W} :\Longmapsto \mathsf{Race}}\ \mathsf{Race}_{\mathsf{np}}$$

$$\frac{\widehat{W} = (T, \_, \mathbb{d}, \sigma) \qquad T(\mathtt{t}) = (F, \kappa) \qquad F \vdash (\kappa, \sigma) \overset{\tau}{\underset{\delta}{\mapsto}}{}^{*} (\kappa', \sigma') \qquad \mathbb{d}(\mathtt{t}) = d}{\mathsf{NPpred}(\widehat{W}, \mathtt{t}, (\delta, d))}\ \mathsf{Predict}_{\mathsf{np}}$$

Figure 16. Data Race in Non-Preemptive Semantics

C More about DRF and NPDRF

Data races in the non-preemptive semantics ($(\hat{P}, \sigma) :\Longmapsto \mathsf{Race}$) are defined in Fig. 16. Similar to the definition for the preemptive semantics, a non-preemptive program configuration $\widehat{W}$ steps to Race ($\widehat{W} :\Longmapsto \mathsf{Race}$) if two threads are predicted having conflicting footprints, as shown in rule $\mathsf{Race}_{\mathsf{np}}$. Predictions are made only at the switch points of non-preemptive semantics ($\widehat{W} :\overset{o}{\underset{\mathsf{emp}}{\Longrightarrow}} \widehat{W}'$ where $o \neq \tau$), i.e., program points at atomic block boundaries, after an observable event, or after thread termination. The prediction rule $\mathsf{Predict}_{\mathsf{np}}$ is a unified version of its counterpart Predict-0 and Predict-1 in Fig. 9, where $\mathbb{d}$ indicates whether the predicted steps are inside an atomic block. Similar to the Predict-1 rule, we do *not* insist on predicting the footprint generated by the execution reaching the next switch point, because the code segments between switch points could be nonterminating. In addition to at the switch points, we also need to be able to perform a prediction at the initial state as well (the second rule in Fig. 16), because the first executing thread is non-deterministically picked at the beginning, which has similar effect as thread switching.

A program $\hat{P}$ with the initial state σ is NPDRF if it never steps to Race, and a program $\hat{P}$ is NPDRF if, for any proper initial state σ, $\mathsf{NPDRF}(\hat{P}, \sigma)$:

$$\mathsf{NPDRF}(\hat{P}, \sigma) \ \text{ iff } \ \neg((\hat{P}, \sigma) :\Longmapsto \mathsf{Race})$$
$$\mathsf{NPDRF}(P) \ \text{ iff } \ \forall \sigma.\ \mathsf{GE}(P.\Pi) \subseteq \mathsf{locs}(\sigma) \wedge \mathsf{cl}(\mathsf{dom}(\sigma), \sigma) = \mathsf{dom}(\sigma) \implies \mathsf{NPDRF}(P, \sigma)$$

Note that defining NPDRF is not for studying the absence of data races in the non-preemptive semantics (which is probably not very interesting since the execution is non-preemptive anyway). Rather, it is intended to serve as an equivalent notion of DRF but formulated in the non-preemptive semantics. The following lemma shows the equivalence.

Lemma 25 (Equivalence between DRF and NPDRF, ⑥ and ⑧ in Fig. 2)**.** For any $\mathtt{f}_1, \ldots, \mathtt{f}_m, \sigma, \Pi = \{(tl_1, \pi_1), \ldots, (tl_m, \pi_m)\}$ such that $\forall i.\ \mathsf{wd}(tl_i)$, and $P = \textbf{let}\ \Pi\ \textbf{in}\ \mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_m$,

$$\mathsf{DRF}(P, \sigma) \iff \mathsf{NPDRF}(P, \sigma)\ .$$

D More about the Extended Framework

D.1 x86-TSO semantics

Our x86-TSO semantics follows the x86-TSO model presented in [15, 19] and CompCertTSO [18].

To make the refinement proof easy, we choose to present the x86-TSO global semantics in a similar way as our preemptive global semantics. The machine states and the global transition rules are shown in Fig. 17 and 18 respectively. Here

- All modules are x86-TSO modules.
- The whole porgram configuration $\mathbb{W}$ is instrumented with write buffers B. Each thread `t` has its own write buffer $\mathtt{B}(\mathtt{t})$.
- We do not need the atomic bit and EntAtom, ExtAtom events in the x86-TSO model.

The Unbuffer rule in the global semantics in Fig. 18 says that the machine non-deterministically apply the writes in a thread's buffer to the real memory.

The x86 instruction sets and the local semantics are shown in Fig. 19. For normal writes, a thread always writes to its buffer rather than the real memory. For `lock`-prefixed instructions, the thread requires the buffer is empty and it updates the memory directly (see the LOCKDEC rule).

$$
\begin{array}{rrcl@{\qquad}rrcl}
(\mathit{Prog}) & P & ::= & \textbf{let}\ \Pi\ \textbf{in}\ \mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n & (\mathit{Entry}) & \mathtt{f} & \in & \mathit{String} \\
(\mathit{MdSet}) & \Pi & ::= & \{(ge_1, \pi_1), \ldots, (ge_m, \pi_m)\} & (\mathit{Module}) & \pi & ::= & \ldots \\
(\mathit{Lang}) & tl_{\mathtt{x86\text{-}TSO}} & ::= & (\mathit{Module}, \mathit{Core}, \mathsf{InitCore}, \longmapsto_{\mathtt{tso}}) & (\mathit{Core}) & \kappa & ::= & (\pi, \ldots) \\
 & ge & \in & \mathcal{P}(\mathit{Addr}) & \multicolumn{4}{l}{\mathsf{InitCore} \in \mathit{Module} \to \mathit{Entry} \rightharpoonup \mathit{list}(\mathit{Val}) \rightharpoonup \mathit{Core}}
\end{array}
$$

$$
\longmapsto_{\mathtt{tso}} \in \mathit{FList} \times (\mathit{Core} \times \mathit{State} \times \mathit{Buf}) \to \{\mathcal{P}((\mathit{Msg} \times \mathit{FtPrt}) \times (\mathit{Core} \times \mathit{State} \times \mathit{Buf})), \textbf{abort}\}
$$

$$
\begin{array}{rrcl@{\qquad}rrcl}
(\mathit{ThrdID}) & \mathtt{t} & \in & \mathbb{N} & (\mathit{Addr}) & l & ::= & \ldots \\
(\mathit{State}) & \sigma & \in & \mathit{Addr} \rightharpoonup_{\mathsf{fin}} \mathit{Val} & (\mathit{Val}) & v & ::= & l \mid \ldots \\
(\mathit{Buf}) & \mathtt{b} & ::= & \epsilon \mid (l, v) :: \mathtt{b} & (\mathit{FList}) & F, \mathbb{F} & \in & \mathcal{P}^{\omega}(\mathit{Addr}) \\
(\mathit{Msg}) & \iota & ::= & \tau \mid e \mid \mathsf{call}(\mathtt{f}, \vec{v}) \mid \mathtt{ret}(v) \mid \mathtt{ub} & (\mathit{Event}) & e & ::= & \textbf{print}(v) \\
(\mathit{Config}) & \phi, \Phi & ::= & (\kappa, \sigma) \mid \textbf{abort} & & & & \\
 & & & & & & & \\
(\mathit{World}) & W_{\mathtt{tso}} & ::= & (T, \mathtt{t}, \mathrm{B}, \sigma) & & & & \\
(\mathit{ThrdPool}) & T, \mathbb{T} & ::= & \{\mathtt{t}_1 \rightsquigarrow K_1, \ldots, \mathtt{t}_n \rightsquigarrow K_n, & & & & \\
(\mathit{RtMdStk}) & K, \mathbb{K} & ::= & \epsilon \mid (F, \kappa) :: K & & & & \\
(\mathit{Buffers}) & \mathrm{B} & ::= & \{\mathtt{t}_1 \rightsquigarrow \mathtt{b}_1, \ldots, \mathtt{t}_n \rightsquigarrow \mathtt{b}_n\} & & & & \\
(\mathit{FSpace}) & \mathcal{F}, \mathbb{FS} & ::= & F :: \mathcal{F} \quad \text{(co-inductive)} & & & & \\
(\mathit{GMsg}) & o & ::= & \tau \mid e \mid \mathtt{sw} \mid \mathtt{ub} & (\mathit{ASet}) & S, \mathbb{S} & \in & \mathcal{P}(\mathit{Addr})
\end{array}
$$

Figure 17. x86TSO machine

$$\frac{\begin{array}{c}\text{for all } i \text{ and } j \text{ in } \{1,\ldots,n\}, \text{ and } i \neq j\text{:}\\ F_i \cap F_j = \emptyset \quad \mathrm{dom}(\sigma) \cap F_i = \emptyset \qquad tl_{\texttt{x86-TSO}}.\mathsf{InitCore}(\pi_i, \mathsf{f}_i) = \kappa_i, \quad \text{where } (tl_{\texttt{x86-TSO}}, ge_i, \pi_i) \in \Pi\\ T = \{1 \rightsquigarrow (tl_{\texttt{x86-TSO}}, F_1, \kappa_1) :: \epsilon, \ldots, n \rightsquigarrow (tl_{\texttt{x86-TSO}}, F_n, \kappa_n) :: \epsilon\} \qquad \mathsf{t} \in \{1,\ldots,n\}\}\end{array}}{(\textbf{let } \Pi \textbf{ in } \mathsf{f}_1 \parallel \ldots \parallel \mathsf{f}_n, \sigma) \overset{load}{\Longrightarrow}_{\mathtt{tso}} (\Pi, \mathcal{F}, T, \mathsf{t}, \{\mathsf{t}_1 \rightsquigarrow \epsilon, \ldots \mathsf{t}_n \rightsquigarrow \epsilon\}, \sigma)}\ \text{Load}$$

$$\frac{\begin{array}{c}T(\mathsf{t}) = (F, \kappa) :: K \quad \mathrm{B}(\mathsf{t}) = \mathsf{b}\\ F \vdash (\kappa, (\mathsf{b}, \sigma)) \overset{\iota}{\underset{\delta}{\longmapsto}}_{\mathtt{tso}} (\kappa', (\mathsf{b}', \sigma'))\\ T' = T\{\mathsf{t} \rightsquigarrow (F, \kappa') :: K\} \quad \iota = \tau\end{array}}{(\Pi, \mathcal{F}, T, \mathsf{t}, \mathrm{B}, \sigma)) \overset{\iota}{\underset{\delta}{\Rightarrow}}_{\mathtt{tso}} (\Pi, \mathcal{F}, T', \mathsf{t}, \mathrm{B}\{\mathsf{t} \rightsquigarrow \mathsf{b}'\}\sigma'))}\ \tau\text{-step}$$

$$\frac{\begin{array}{c}T(\mathsf{t}) = (F, \kappa) :: K \quad \mathrm{B}(\mathsf{t}) = \epsilon\\ F \vdash (\kappa, (\epsilon, \sigma)) \overset{e}{\underset{\mathtt{emp}}{\longmapsto}}_{\mathtt{tso}} (\kappa', (\epsilon, \sigma))\\ T' = T\{\mathsf{t} \rightsquigarrow (F, \kappa') :: K\}\end{array}}{(\Pi, \mathcal{F}, T, \mathsf{t}, \mathrm{B}, \sigma)) \overset{e}{\underset{\mathtt{emp}}{\Longrightarrow}}_{\mathtt{tso}} (\Pi, \mathcal{F}, T', \mathsf{t}, \mathrm{B}, \sigma))}\ \text{Print}$$

$$\frac{\begin{array}{c}T(\mathsf{t}) = (F, \kappa) :: K \qquad \mathrm{B}(\mathsf{t}) = \mathsf{b} F \vdash (\kappa, (\mathsf{b}, \sigma)) \overset{\mathsf{call}(\mathsf{f}, \vec{v})}{\underset{\mathtt{emp}}{\longmapsto}} (\kappa', (\mathsf{b}, \sigma))\\ \mathcal{F} = F_1 :: \mathcal{F}' \qquad \mathsf{initRtMd}(\Pi, \mathsf{f}, \vec{v}, \kappa_1) \qquad T' = \{\mathsf{t} \rightsquigarrow (F_1, \kappa_1) :: (F, \kappa') :: K, \}\end{array}}{(\Pi, \mathcal{F}, T, \mathsf{t}, \mathrm{B}, \sigma) \overset{\tau}{\underset{\mathtt{emp}}{\Longrightarrow}}_{\mathtt{tso}} (\Pi, \mathcal{F}', T', \mathsf{t}, \mathrm{B}, \sigma)}\ \text{EFCall}$$

$$\frac{\begin{array}{c}T(\mathsf{t}) = (F_1, \kappa_1) :: (F, \kappa) :: K, \qquad F_1 \vdash (\kappa_1, (\mathsf{b}, \sigma)) \overset{\mathsf{ret}(v)}{\underset{\mathtt{emp}}{\longmapsto}} (\kappa_1', (\mathsf{b}, \sigma))\\ \mathsf{AftExtn}(\kappa, v) = \kappa' \qquad T' = T\{\mathsf{t} \rightsquigarrow (F, \kappa') :: K, \}\end{array}}{(\Pi, \mathcal{F}, T, \mathsf{t}, \mathrm{B}, \sigma) \overset{\tau}{\underset{\mathtt{emp}}{\Longrightarrow}} (\Pi, \mathcal{F}, T', \mathsf{t}, \mathrm{B}, \sigma)}\ \text{EFRet}$$

$$\frac{\begin{array}{c}T(\mathsf{t}) = (F, \kappa) :: \epsilon \quad \mathrm{B}(\mathsf{t}) = \epsilon \quad \mathsf{t}' \in \mathrm{dom}(T \backslash \mathsf{t})\\ F \vdash (\kappa, (\epsilon, \sigma)) \overset{\mathsf{ret}(v)}{\underset{\mathtt{tso}}{\longmapsto}} (\kappa', (\epsilon, \sigma))\end{array}}{(\Pi, \mathcal{F}, T, \mathsf{t}, \mathrm{B}, \sigma) \overset{\mathsf{sw}}{\underset{\mathtt{emp}}{\Longrightarrow}}_{\mathtt{tso}} (\Pi, \mathcal{F}, T \backslash \mathsf{t}, \mathsf{t}', \mathrm{B} \backslash \mathsf{t}, \sigma)}\ \text{Term}$$

$$\frac{\begin{array}{c}T(\mathsf{t}) = (F, \kappa) :: \epsilon \quad \mathrm{B}(\mathsf{t}) = \epsilon \quad \mathrm{dom}(T) = \mathrm{dom}(\mathrm{B}) = \{\mathsf{t}\}\\ F \vdash (\kappa, (\epsilon, \sigma)) \overset{\mathsf{ret}(v)}{\underset{\mathtt{tso}}{\longmapsto}} (\kappa', (\epsilon, \sigma))\end{array}}{(\Pi, \mathcal{F}, T, \mathsf{t}, \mathrm{B}, \sigma) \overset{\tau}{\Rightarrow}_{\mathtt{tso}} \textbf{done}}\ \text{Done}$$

$$\frac{\mathsf{t}' \in \mathrm{dom}(T)}{(\Pi, \mathcal{F}, T, \mathsf{t}, \mathrm{B}, \sigma) \overset{\mathsf{sw}}{\underset{\mathtt{emp}}{\Longrightarrow}}_{\mathtt{tso}} (\Pi, \mathcal{F}, T, \mathsf{t}', \mathrm{B}, \sigma)}\ \text{Switch}$$

$$\frac{\mathrm{B}(\mathsf{t}') = (l, v) :: b \quad \sigma' = \mathsf{apply\_buffer}((l, v), \sigma) \quad \mathrm{B}' = \mathrm{B}\{\mathsf{t}' \rightsquigarrow b\}}{(\Pi, \mathcal{F}, T, \mathsf{t}, \mathrm{B}, \sigma) \overset{\mathsf{ub}}{\underset{\mathtt{emp}}{\Longrightarrow}}_{\mathtt{tso}} (\Pi, \mathcal{F}, T, \mathsf{t}, \mathrm{B}', \sigma')}\ \text{Unbuffer}$$

$$\frac{T(\mathsf{t}) = (F, \kappa) :: \epsilon, \quad \mathrm{B}(\mathsf{t}) = \mathsf{b} \quad F \vdash (\kappa, (\mathsf{b}, \sigma)) \underset{\mathtt{tso}}{\longmapsto} \textbf{abort}}{(\Pi, \mathcal{F}, T, \mathsf{t}, \mathrm{B}, \sigma) \overset{\tau}{\Rightarrow}_{\mathtt{tso}} \textbf{abort}}\ \text{Abort}$$

Figure 18. x86TSO global semantics

$$
\begin{array}{rrcl}
(\textit{Instr}) & c & ::= & \texttt{mov}\ \mathsf{r}_d, \mathsf{r}_s \mid \texttt{Iadd}\ \mathsf{r}_d, \mathsf{r}_s \mid \texttt{call f} \mid \texttt{ret} \mid \ldots \\
& & \mid & \texttt{lock xchg}\ l\ \mathsf{r}_s \mid \texttt{lock xadd}\ l\ \mathsf{r}_s \mid \texttt{lock dec}\ l \mid \texttt{mfence} \\
(\textit{Core}) & \kappa & ::= & \textit{Register} \to \textit{Val} \\
(\textit{Register}) & \mathsf{r} & ::= & \mathsf{PC} \mid \mathsf{SP} \mid \ldots
\end{array}
$$

$$
\frac{\begin{array}{c}\texttt{find\_instr}(ge, \kappa(\mathsf{PC})) = (\texttt{mov}\ \mathsf{r}_d,\ [\mathsf{r}_s]) \quad \kappa(\mathsf{r}_s) = l \\ l \notin \mathrm{dom}(\mathrm{b}) \quad \sigma(l) = v \quad \delta = (\{l\}, \emptyset)\end{array}}{(ge, F) \vdash (\kappa, (\mathrm{b}, \sigma)) \xmapsto[\delta]{\tau}_{\mathsf{tso}} (\kappa\{\mathsf{r} \rightsquigarrow v, \mathsf{PC} \rightsquigarrow \mathsf{PC} + 1\}, (\mathrm{b}, \sigma))}\ \textsc{READMEM}
$$

$$
\frac{\begin{array}{c}\texttt{find\_instr}(ge, \kappa(\mathsf{PC})) = (\texttt{mov}\ \mathsf{r}_d,\ [\mathsf{r}_s]) \quad \kappa(\mathsf{r}_s) = l \\ \mathrm{b} = \mathrm{b}_1 +\!\!+ (l, v) :: \mathrm{b}_0 \quad l \notin \mathrm{dom}(\mathrm{b}_0) \quad \delta = (\{l\}, \emptyset)\end{array}}{(ge, F) \vdash (\kappa, (\mathrm{b}, \sigma)) \xmapsto[\delta]{\tau}_{\mathsf{tso}} (\kappa\{\mathsf{r} \rightsquigarrow v, \mathsf{PC} \rightsquigarrow \mathsf{PC} + 1\}, (\mathrm{b}, \sigma))}\ \textsc{READBUF}
$$

$$
\frac{\begin{array}{c}\texttt{find\_instr}(ge, \kappa(\mathsf{PC})) = (\texttt{mov}\ [\mathsf{r}_d],\ \mathsf{r}_s) \quad \kappa(\mathsf{r}_s) = v \quad \kappa(\mathsf{r}_d) = l \\ \delta = (\emptyset, \{l\}) \quad \texttt{apply\_buffer}(\mathrm{b} +\!\!+ (l, v), \sigma) = \sigma'\end{array}}{(ge, F) \vdash (\kappa, (\mathrm{b}, \sigma)) \xmapsto[\delta]{\tau}_{\mathsf{tso}} (\kappa\{\mathsf{PC} \rightsquigarrow \mathsf{PC} + 1\}, (\mathrm{b} +\!\!+ (l, v), \sigma))}\ \textsc{WRITEBUF}
$$

$$
\frac{\begin{array}{c}\texttt{find\_instr}(ge, R(\mathsf{PC})) = \texttt{lock dec}\ l\ \mathsf{r} \quad R(\mathsf{SF}) = \sigma(l) \le 0 \\ \kappa' = \kappa\{\mathsf{PC} \rightsquigarrow \mathsf{PC} + 1, \mathsf{r} \rightsquigarrow \sigma(l)\} \quad \sigma' = \sigma\{l \rightsquigarrow \sigma(l) - 1\} \quad \delta = (\{l\}, \{l\})\end{array}}{(ge, F) \vdash (\kappa, (\epsilon, \sigma)) \xmapsto[\delta]{\tau}_{\mathsf{tso}} (\kappa', (\epsilon, \sigma'))}\ \textsc{LOCKDEC}
$$

$$
\frac{\texttt{find\_instr}(ge, R(\mathsf{PC})) = \texttt{mfence}}{(ge, F) \vdash (\kappa, (\epsilon, \sigma)) \xmapsto[\mathsf{emp}]{\tau}_{\mathsf{tso}} (\kappa\{\mathsf{PC} \rightsquigarrow \mathsf{PC} + 1\}, (\epsilon, \sigma))}\ \textsc{BARRIER}
\qquad
\frac{\begin{array}{c}\texttt{find\_instr}(ge, R(\mathsf{PC})) = \texttt{call}\ \mathbf{print}() \\ \kappa' = \kappa\{\mathsf{PC} \rightsquigarrow \mathsf{PC} + 1\} \quad \sigma(\text{first argument}) = v\end{array}}{(ge, F) \vdash (\kappa, (\epsilon, \sigma)) \xmapsto[\mathsf{emp}]{\mathbf{print}(v)}_{\mathsf{tso}} (\kappa', (\epsilon, \sigma))}\ \textsc{PRINT}
$$

$$
\begin{array}{c}
\texttt{apply\_buffer}((l, v) :: \mathrm{b}, \sigma) ::= \texttt{apply\_buffer}(\mathrm{b}, \sigma\{l \rightsquigarrow v\}) \\
\texttt{apply\_buffer}(\epsilon, \sigma) ::= \sigma
\end{array}
$$

Figure 19. x86TSO thread local semantics

$$
\begin{array}{rcl}
\mathtt{objM}(l,\Sigma) & \stackrel{\text{def}}{=} & l \in \mathrm{dom}(\Sigma) \wedge \Sigma(l) = (\_, \mathrm{None}) \\
\mathtt{objFP}(\delta,\Sigma) & \stackrel{\text{def}}{=} & \forall l \in (\delta.rs \cup \delta.ws)\,.\,\mathtt{objM}(l,\Sigma) \\
\mathtt{clientFP}(\delta,\Sigma) & \stackrel{\text{def}}{=} & \forall l \in (\delta.rs \cup \delta.ws)\,.\,\neg\mathtt{objM}(l,\Sigma) \\
\mathrm{conflict}_c(\delta,\mathtt{t},\mathrm{B}) & \stackrel{\text{def}}{=} & (\delta.rs \cup \delta.ws) \cap \bigcup_{\mathtt{t}'\neq\mathtt{t}} \mathrm{dom}(\mathrm{B}(\mathtt{t}')) \neq \emptyset \\
\mathrm{bufConflict}(\mathrm{B},\Sigma) & \stackrel{\text{def}}{=} & \exists l, \mathtt{t}_1, \mathtt{t}_2. \\
 & & l \in \mathrm{dom}(\mathrm{B}(\mathtt{t}_1)) \cap \mathrm{dom}(\mathrm{B}(\mathtt{t}_2)) \\
 & & \wedge \neg\mathtt{objM}(l,\Sigma) \\
\mathrm{G} \Rightarrow \mathrm{R} & \text{iff} & \mathrm{G}((\Sigma,\sigma),(\Sigma',\sigma')) \implies \mathrm{R}((\Sigma,\sigma),(\Sigma',\sigma')) \\
\mathrm{I} \ltimes \mathrm{I}'((\Sigma,\sigma),(\Sigma',\sigma')) & \stackrel{\text{def}}{=} & \mathrm{I}((\Sigma,\sigma)) \wedge \mathrm{I}'((\Sigma',\sigma'))
\end{array}
$$

Figure 20. Auxiliary definitions for module local simulations

D.2 Object correctness definition

Definition 26 (Object corrrectness).
$(tl_{\mathtt{tso}}, ge_o, \pi_o) \preccurlyeq^{\mathrm{o}} (sl_{\mathtt{CImp}}, ge_o, \gamma_o)$ iff $\exists \mathrm{R_o}, \mathrm{G_o}, \mathrm{I_o}$. such that

1. for any $\mathtt{t}$, $(sl_{\mathtt{CImp}}, ge_o, \gamma_o) \succcurlyeq^{\mathrm{o}}_{\mathrm{R_o^t};\mathrm{G_o^t};\mathrm{I_o}} (tl_{\mathtt{tso}}, ge_o, \pi_o)$, and
2. $\mathtt{RespectPartition}_o(\mathrm{R_o}, \mathrm{G_o}, \mathtt{objM})$ and
3. $\mathtt{Sta}(\mathrm{I_o}, \mathrm{R_o^t})$ and
4. $\forall \mathtt{t}_1 \neq \mathtt{t}_2.\ \mathrm{G_o^{t_1}} \Rightarrow \mathrm{R_o^{t_2}}$.

Where $\mathtt{RespectPartition}_o$ is defined in definition 27.

$(sl_{\mathtt{CImp}}, ge_o, \gamma_o) \succcurlyeq^{\mathrm{o}}_{\mathrm{R_o^t};\mathrm{G_o^t};\mathrm{I_o}} (tl_{\mathtt{tso}}, ge_o, \pi_o)$ is an extension of Liang and Feng [12] with the support of TSO semantics for the low-level code, and is defined in definition 28.

Definition 27 (Respect partition).
$\mathtt{RespectPartition}_o(\mathrm{R_o}, \mathrm{G_o}, p_o)$ iff

1. $\forall \Sigma, \mathrm{B}, \sigma, \Sigma', \mathrm{B}', \sigma', \mathtt{t}.$
$$\Sigma' \xlongequal{\{l \mid p_o(l,\Sigma)\}} \Sigma \wedge \sigma' \xlongequal{\{l \mid p_o(l,\Sigma)\}} \sigma \wedge \left(\begin{array}{l} \mathrm{B}' = \mathrm{B} \\ \vee \exists l, v, \mathtt{t}' \neq \mathtt{t}.\ \neg p_o(l,\Sigma) \wedge \mathrm{B}' = \mathrm{B}\{\mathtt{t}' \rightsquigarrow \mathrm{B}(\mathtt{t}') +\!\!+ (l,v)\} \end{array} \right) \implies \mathrm{R_o^t}((\Sigma,(\mathrm{B},\sigma)),(\Sigma',(\mathrm{B}',\sigma')))$$
2. $\forall \Sigma, \Sigma', \mathrm{B}, \sigma, \mathrm{B}', \sigma', \mathtt{t}.$
$$\mathrm{G_o^t}((\Sigma,(\mathrm{B},\sigma))(\Sigma',(\mathrm{B}',\sigma')) \implies \left(\begin{array}{l} \Sigma' \xlongequal{\{l \mid \neg p_o(l,\Sigma)\}} \Sigma \wedge \sigma' \xlongequal{\{l \mid \neg p_o(l,\Sigma)\}} \sigma \\ \wedge (\mathrm{B}' = \mathrm{B} \vee \exists l, v.\ \mathtt{objM}(l,\Sigma) \wedge \mathrm{B}' = \mathrm{B}\{\mathtt{t} \rightsquigarrow \mathrm{B}(\mathtt{t}) +\!\!+ (l,v)\}) \end{array} \right)$$

Here we partition memory into object memory and client memory using assertion p_o. And this definition says object rely/guarantee relations $\mathrm{R_o}/\mathrm{G_o}$ respect this partition if and only if the object rely $\mathrm{R_o}$ holds for any change of non-object memory, and the object guarantee $\mathrm{G_o}$ guarantees non-object moemry unchanged.

We instantiated the partition as $\mathtt{objM}$, defined in Fig. 20, which determines whether a location is in object memory by the location's permission.

Definition 28 (Object module simulation).
$(sl_{\mathtt{CImp}}, ge_o, \gamma_o) \succcurlyeq^{\mathrm{o}}_{\mathrm{R_o^t};\mathrm{G_o^t};\mathrm{I_o}} (tl_{\mathtt{tso}}, ge_o, \pi_o)$ iff
$\forall \Sigma, \mathrm{B}, \sigma, \mathtt{f}, \vec{v}, \kappa$. if $\mathrm{InitCore}(\pi, \mathtt{f}, \vec{v}) = \kappa$ and $\mathrm{I_o}(\Sigma,(\mathrm{B},\sigma))$ then
$\exists \mathbb{k}$. $\mathrm{InitCore}(\gamma, \mathtt{f}, \vec{v}) = \mathbb{k}$ and $(\mathbb{k},\Sigma) \succcurlyeq^{\mathrm{o}}_{\mathrm{t};\mathrm{R_o^t};\mathrm{G_o^t};\mathrm{I_o}} (\kappa,(\mathrm{B},\sigma))$.

Here $(\mathbb{k},\Sigma) \succcurlyeq^{\mathrm{o}}_{\mathrm{t};\mathrm{R_o^t};\mathrm{G_o^t};\mathrm{I_o}} (\kappa,(\mathrm{B},\sigma))$ is defined as the largest relation such that whenever $(\mathbb{k},\Sigma) \succcurlyeq^{\mathrm{o}}_{\mathrm{t};\mathrm{R_o^t};\mathrm{G_o^t};\mathrm{I_o}} (\kappa,(\mathrm{B},\sigma))$, then all the following hold:

1. if $(\kappa,(\mathrm{B}(\mathtt{t}),\sigma)) \xmapsto[\delta]{\tau} (\kappa',(\mathrm{b}',\sigma'))$ then
   either $(\mathbb{k},\Sigma) \longmapsto^* \mathbf{abort}$,
   or $\exists \mathbb{k}', \Sigma', \Delta.$ such that
   a. $(\mathbb{k},\Sigma) \xmapsto[\Delta]{\tau}{}^*(\mathbb{k}',\Sigma')$ or
   $(\mathbb{k},\Sigma) \xrightarrow{\mathrm{EntAtom}} (\mathbb{k}_0,\Sigma) \xmapsto[\Delta]{\tau}{}^*(\mathbb{k}_1,\Sigma') \xrightarrow{\mathrm{ExtAtom}} (\mathbb{k}',\Sigma')$,
   and
   b. $\mathrm{G_o^t}((\Sigma,(\mathrm{B},\sigma)),(\Sigma',(\mathrm{B}\{\mathtt{t} \rightsquigarrow \mathrm{b}'\},\sigma')))$, and $\mathtt{objFP}(\delta,\Sigma)$,
   and
   c. $(\mathbb{k}',\Sigma') \succcurlyeq^{\mathrm{o}}_{\mathrm{t};\mathrm{R_o^t};\mathrm{G_o^t};\mathrm{I_o}} (\kappa',(\mathrm{B}\{\mathtt{t} \rightsquigarrow \mathrm{b}'\},\sigma'))$.
2. if $(\kappa,(\mathrm{B}(\mathtt{t}),\sigma)) \xmapsto[\mathtt{emp}]{\mathtt{ret}(v)} (\kappa',(\mathrm{B}(\mathtt{t}),\sigma)$ then
   either $(\mathbb{k},\Sigma) \longmapsto^* \mathbf{abort}$,
   or $\exists \mathbb{k}'.$ such that
   a. $(\mathbb{k},\Sigma) \xmapsto[\mathtt{emp}]{\mathtt{ret}(v)}{}^*(\mathbb{k}',\Sigma)$, and
   b. $(\mathbb{k}',\Sigma) \succcurlyeq^{\mathrm{o}}_{\mathrm{t};\mathrm{R_o^t};\mathrm{G_o^t};\mathrm{I_o}} (\kappa',(\mathrm{B},\sigma))$.
3. if $\mathrm{R_o^t}((\Sigma,(\mathrm{B},\sigma)),(\Sigma',(\mathrm{B}',\sigma')))$ then
   $(\kappa,\Sigma') \succcurlyeq^{\mathrm{o}}_{\mathrm{t};\mathrm{R_o^t};\mathrm{G_o^t};\mathrm{I_o}} (\kappa,(\mathrm{B}',\sigma'))$.
4. $\neg(\kappa,(\mathrm{B}(\mathtt{t}),\sigma)) \xmapsto[\mathtt{emp}]{\mathtt{call}(\mathtt{f},\vec{v})} (\kappa',(\mathrm{B}(\mathtt{t}),\sigma)$.
5. if $(\kappa,(\mathrm{B}(\mathtt{t}),\sigma)) \longmapsto \mathbf{abort}$, then $(\kappa,\Sigma) \longmapsto \mathbf{abort}$.

D.3 Proof of Theorem 13

Theorem 29 (Correctness with x86-TSO backend and obj.).
For any $f_1 \ldots f_n, \Gamma = \{(sl, ge_1, \gamma_1), \ldots, (sl, ge_m, \gamma_m), (sl_{\mathtt{CImp}}, ge_o, \gamma_o)\}$, and $\Pi = \{(tl, ge'_1, \pi_1), \ldots, (tl, ge'_m, \pi_m), (tl, ge_o, \pi_o)\}$, if

1. $\forall i \in \{1,\ldots,m\}.\ (\mathtt{CompCert.CodeT}(\gamma_i) = \pi_i) \wedge \mathrm{injective}(\varphi) \wedge (\mathtt{CompCert}.\varphi = \varphi) \wedge \varphi\{ge_i\} = ge'_i$,
2. $\mathrm{Safe}(\mathbf{let}\ \Gamma\ \mathbf{in}\ \mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n)$,
3. $\mathrm{DRF}(\mathbf{let}\ \Gamma\ \mathbf{in}\ \mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n)$,
4. $\forall i \in \{1,\ldots,m\}.\ \mathrm{ReachClose}(sl, ge_i, \gamma_i)$, and $\mathrm{ReachClose}(sl_{\mathtt{CImp}}, ge_o, \gamma_o)$,
5. $(tl, ge_o, \pi_o) \preccurlyeq^{\mathrm{o}} (sl_{\mathtt{CImp}}, ge_o, \gamma_o)$, and
6. $\forall l \in ge_o.\ \Sigma(l) = (\_, \mathrm{None})$,

then $\mathbf{let}\ \Gamma\ \mathbf{in}\ \mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n \sqsupseteq_\varphi \mathbf{let}\ \Pi\ \mathbf{in}\ \mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n$.

Proof. By transitivity of refinement, it suffices to prove

$$\mathbf{let}\ \Gamma\ \mathbf{in}\ \mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n \sqsupseteq_\varphi \mathbf{let}\ \Pi_{\mathtt{sc}}\ \mathbf{in}\ \mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n \qquad \text{(D.1)}$$

and

$$\mathbf{let}\ \Pi_{\mathtt{sc}}\ \mathbf{in}\ \mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n \sqsupseteq_{id, ge_o} \mathbf{let}\ \Pi\ \mathbf{in}\ \mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n \qquad \text{(D.2)}$$

Where
$\Pi_{\mathtt{sc}} = \{(tl_{\mathtt{sc}}, ge'_1, \pi_1), \ldots, (tl_{\mathtt{sc}}, ge'_m, \pi_m), (sl_{\mathtt{CImp}}, ge_o, \gamma_o)\}$

(D.1) is proved by applying Theorem 19, and premise 1-4.

To prove (D.2), by Lemma 30 below, it suffice to prove $\mathrm{Safe}(\mathbf{let}\ \Gamma\ \mathbf{in}\ \mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n)$ and $\mathrm{DRF}(\mathbf{let}\ \Gamma\ \mathbf{in}\ \mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n)$. The first is implied by premise 2 and (D.1); the second is implied by premise 3 and DRF preservation results as we discussed in section 5. □

Lemma 30 (Restore SC semantics for DRF x86 programs).
For any $\Pi_{\mathtt{sc}} = \{(tl_{\mathtt{sc}}, ge_1, \pi_1), \ldots, (tl_{\mathtt{sc}}, ge_m, \pi_m), (sl_{\mathtt{CImp}}, ge_o, \gamma_o)\}$,

$\Pi_{\mathtt{tso}} = \{(tl_{\mathtt{tso}}, ge_1, \pi_1), \ldots, (tl_{\mathtt{tso}}, ge_m, \pi_m), (tl_{\mathtt{tso}}, ge_o, \pi_o)\}$, and for any $f_1, \ldots, f_n$,

1. Safe(**let** $\Pi_{\mathtt{sc}}$ **in** $\mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n$), and
2. DRF(**let** $\Pi_{\mathtt{sc}}$ **in** $\mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n$), and
3. $(tl, ge_o, \pi_o) \preccurlyeq^{\mathrm{o}} (sl_{\mathtt{CImp}}, ge_o, \gamma_o)$

then

let $\Pi_{\mathtt{sc}}$ **in** $\mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n \sqsupseteq_{id, ge_o}$ **let** $\Pi_{\mathtt{tso}}$ **in** $\mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n$.

Proof. By lemma 32, it suffices to prove **let** $\Pi_{\mathtt{tso}}$ **in** $\mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n \leqslant_{id, ge_o}$ **let** $\Pi_{\mathtt{sc}}$ **in** $\mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n$, as defined below.

By compositionality theorem 37 and lemma 35, 36, 42, it suffices to prove DRF(**let** $\Pi_{\mathtt{sc}}$ **in** $\mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n$), and Safe(**let** $\Pi_{\mathtt{sc}}$ **in** $\mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n$), which are exactly premises 1 and 2. □

D.3.1 Whole-program simulation between TSO and SC

Definition 31 (Whole program simulation).
For any $f_1, \ldots, f_n, \Pi_{\mathtt{sc}}, \Pi_{\mathtt{tso}}, ge_o$,
let $\Pi_{\mathtt{tso}}$ **in** $\mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n \leqslant_{id, ge_o}$ **let** $\Pi_{\mathtt{sc}}$ **in** $\mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n$ iff
$\forall \Sigma, \sigma, T, \mathtt{t}$. if $\mathsf{initM}(id, \mathsf{GE}(\Pi_{\mathtt{sc}}), \Sigma, \sigma)$ and
$\left(\forall l \in ge_{\mathtt{lock}}.\, \Sigma(l) = (\_, \mathrm{None})\right)$ and
$(\textbf{let } \Pi_{\mathtt{tso}} \textbf{ in } \mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n, \sigma) \overset{load}{\Longrightarrow}_{\mathtt{tso}} (\Pi_{\mathtt{tso}}, \mathcal{F}, T, \mathtt{t}, (\mathrm{B}, \sigma))$ then
$\exists \mathbb{T}.\ (\textbf{let } \Pi_{\mathtt{sc}} \textbf{ in } \mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n, \Sigma) \overset{load}{\Longrightarrow} (\Pi_{\mathtt{x86\text{-}SC}}, \mathcal{F}, \mathbb{T}, \mathtt{t}, 0, \Sigma))$ and
$(\Pi_{\mathtt{tso}}, \mathcal{F}, T, \mathtt{t}, (\mathrm{B}, \sigma)) \leqslant_{id} (\Pi_{\mathtt{sc}}, \mathcal{F}, \mathbb{T}, \mathtt{t}, 0, \Sigma))$.

Here $W \leqslant_{id} \mathbb{W}$ is defined as the largest relation such that whenever $W \leqslant_{id} \mathbb{W}$, then the following hold:

1. if $W \overset{\tau/\mathsf{ub}}{\Longrightarrow} W'$ then there exists $\mathbb{W}'$ such that $\mathbb{W} \overset{\tau}{\Rightarrow}^* \mathbb{W}'$ and $W' \leqslant_{id} \mathbb{W}'$.
2. if $W \overset{\iota}{\Longrightarrow} W'$ and $\iota \neq \tau$ and $\iota \neq \mathsf{ub}$ then there exists $\mathbb{W}'$ such that $\mathbb{W} \overset{\iota}{\Longrightarrow} \mathbb{W}'$ and $W' \leqslant_{id} \mathbb{W}'$.
3. if $W \Longrightarrow$ **abort**, then $\mathbb{W} \Longrightarrow^*$ **abort**.
4. if $W \Longrightarrow$ **done**, then $\mathbb{W} \Longrightarrow^*$ **done**.

Lemma 32 (whole program simulation implies ref.).
For any $f_1, \ldots, f_n, \Pi_{\mathtt{sc}}, \Pi_{\mathtt{tso}}, ge_o$,
let $\Pi_{\mathtt{tso}}$ **in** $\mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n \leqslant_{id, ge_o}$ **let** $\Pi_{\mathtt{sc}}$ **in** $\mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n$
$\Longrightarrow$ **let** $\Pi_{\mathtt{sc}}$ **in** $\mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n \sqsupseteq_{id, ge_o}$ **let** $\Pi_{\mathtt{tso}}$ **in** $\mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n$

D.3.2 Client simulation and composition

Definition 33 (Client module simulation).
$(tl_{\mathtt{sc}}, ge, \pi) \succcurlyeq^{\mathrm{c}}_{\mathrm{R}^{\mathtt{t}}_{\mathrm{c}}; \mathrm{G}^{\mathtt{t}}_{\mathrm{c}}; \mathrm{I}_{\mathrm{c}}} (tl_{\mathtt{tso}}, ge, \pi)$ iff
$\forall \Sigma, \mathrm{B}, \sigma, \mathtt{f}, \vec{v}, \kappa$. if $\mathsf{InitCore}(\pi, \mathtt{f}, \vec{v}) = \kappa$ and $\mathrm{I}_{\mathrm{c}}(\Sigma, (\mathrm{B}, \sigma))$ then
$(\kappa, \Sigma) \succcurlyeq^{\mathrm{c}}_{\mathrm{t}; \mathrm{R}^{\mathtt{t}}_{\mathrm{c}}; \mathrm{G}^{\mathtt{t}}_{\mathrm{c}}; \mathrm{I}_{\mathrm{c}}} (\kappa, (\mathrm{B}, \sigma))$.
Here $(\kappa, \Sigma) \succcurlyeq^{\mathrm{c}}_{\mathrm{t}; \mathrm{R}^{\mathtt{t}}_{\mathrm{c}}; \mathrm{G}^{\mathtt{t}}_{\mathrm{c}}; \mathrm{I}_{\mathrm{c}}} (\kappa, (\mathrm{B}, \sigma))$ is the largest relation such that whenever $(\kappa, \Sigma) \succcurlyeq^{\mathrm{c}}_{\mathrm{t}; \mathrm{R}^{\mathtt{t}}_{\mathrm{c}}; \mathrm{G}^{\mathtt{t}}_{\mathrm{c}}; \mathrm{I}_{\mathrm{c}}} (\kappa, (\mathrm{B}, \sigma))$, then the following hold:

1. if $(\kappa, (\mathrm{B}(\mathtt{t}), \sigma)) \overset{\iota}{\underset{\delta}{\longmapsto}} (\kappa', (\mathtt{b}', \sigma'))$ then
   either $(\kappa, \Sigma) \longmapsto$ **abort**
   or $\exists \kappa'', \Sigma', \Delta.$ such that $(\kappa, \Sigma) \overset{\iota}{\underset{\Delta}{\longmapsto}}{}^{+} (\kappa'', \Sigma')$ and $\delta.rs \cup \delta.ws \subseteq \Delta.rs \cup \Delta.ws$ and $\mathtt{clientFP}(\delta, \Sigma)$ and one of the following (a) and (b) holds:
   a. $\neg \mathrm{conflict}_c(\delta, \mathtt{t}, \mathrm{B})$, and $\kappa'' = \kappa'$, and $\Delta = \delta$, and $\mathrm{G}^{\mathtt{t}}_{\mathrm{c}}((\Sigma, (\mathrm{B}, \sigma)), (\Sigma', (\mathrm{B}\{\mathtt{t} \rightsquigarrow \mathtt{b}'\}, \sigma')))$, and $(\kappa', \Sigma') \succcurlyeq^{\mathrm{c}}_{\mathrm{t}; \mathrm{R}^{\mathtt{t}}_{\mathrm{c}}; \mathrm{G}^{\mathtt{t}}_{\mathrm{c}}; \mathrm{I}_{\mathrm{c}}} (\kappa', (\mathrm{B}\{\mathtt{t} \rightsquigarrow \mathtt{b}'\}, \sigma'))$, or
   b. $\mathrm{conflict}_c(\delta, \mathtt{t}, \mathrm{B}, \mathtt{b}')$.
2. if $(\kappa, (\mathrm{B}(\mathtt{t}), \sigma)) \longmapsto$ **abort**, then $(\kappa, \Sigma) \longmapsto^*$ **abort**.
3. if $\mathrm{R}^{\mathtt{t}}_{\mathrm{c}}((\Sigma, (\mathrm{B}, \sigma)), (\Sigma', (\mathrm{B}', \sigma')))$, then $(\kappa, \Sigma') \succcurlyeq^{\mathrm{c}}_{\mathrm{t}; \mathrm{R}^{\mathtt{t}}_{\mathrm{c}}; \mathrm{G}^{\mathtt{t}}_{\mathrm{c}}; \mathrm{I}_{\mathrm{c}}} (\kappa, (\mathrm{B}', \sigma'))$.

Definition 34 ($\mathrm{I}_{\mathrm{c}}, \mathrm{R}_{\mathrm{c}}, \mathrm{G}_{\mathrm{c}}$ definitions).

$$\mathrm{I}_{\mathrm{c}}(\Sigma, (\mathrm{B}, \sigma)) \overset{\mathrm{def}}{=} \neg \mathrm{bufConflict}(\mathrm{B}, \Sigma) \wedge \left(\begin{array}{l} \forall \sigma', l.\, \mathtt{apply\_buffers}(\mathrm{B}, \sigma, \sigma') \wedge \neg \mathtt{objM}(l, \Sigma) \\ \quad \Longrightarrow \Sigma(l) = \sigma'(l) \end{array} \right)$$

$$\mathrm{R}^{\mathtt{t}}_{\mathrm{c}} \overset{\mathrm{def}}{=} \mathrm{I}_{\mathrm{c}} \ltimes \mathrm{I}_{\mathrm{c}}$$

$$\mathrm{G}^{\mathtt{t}}_{\mathrm{c}}((\Sigma, (\mathrm{B}, \sigma)), (\Sigma', (\mathrm{B}', \sigma'))) \overset{\mathrm{def}}{=} \mathrm{I}_{\mathrm{c}}(\Sigma, (\mathrm{B}, \sigma)) \wedge \mathrm{I}_{\mathrm{c}}(\Sigma', (\mathrm{B}', \sigma')) \wedge \sigma' = \sigma \wedge \Sigma' \overset{\{l \mid \mathtt{objM}(l, \Sigma)\}}{=\!=\!=} \Sigma \wedge \left(\begin{array}{l} \mathrm{B}' = \mathrm{B}\{\mathtt{t} \rightsquigarrow \mathrm{B}(\mathtt{t}) +\!\!+ (l, v)\} \wedge \neg \mathtt{objM}(l, \Sigma) \\ \vee \\ \mathrm{B}' = \mathrm{B} \end{array} \right)$$

Where `apply_buffers` is inductively defined as:

$$\frac{\mathrm{B} = \{\mathtt{t}_1 \rightsquigarrow \epsilon, \ldots, \mathtt{t}_n \rightsquigarrow \epsilon\}}{\mathtt{apply\_buffers}(\mathrm{B}, \sigma, \sigma)}$$

$$\frac{\mathrm{B}(\mathtt{t}) = (l, v) :: \mathtt{b} \qquad \mathtt{apply\_buffers}(\mathrm{B}\{\mathtt{t} \rightsquigarrow \mathtt{b}\}, \sigma\{l \rightsquigarrow v\}, \sigma')}{\mathtt{apply\_buffers}(\mathrm{B}, \sigma, \sigma')}$$

I_{c} says buffered writes in different threads are not conflicted (unless they are object writes, i.e.write to location l which is $\mathtt{objM}(l, \Sigma)$) and client memory are equal between SC and TSO when buffers are applied to TSO memory.

$\mathrm{G}^{\mathtt{t}}_{\mathrm{c}}$ says a thread can only append write (l, v) to buffer when the location l is not object memory location.

Lemma 35 (client-object R/G properties).
For any $\mathrm{R}_{\mathrm{o}}, \mathrm{G}_{\mathrm{o}}, \mathrm{I}_{\mathrm{o}}$, if $\mathtt{RespectPartition}_o(\mathrm{R}_{\mathrm{o}}, \mathrm{G}_{\mathrm{o}}, \mathtt{objM})$ then
$\forall \mathtt{t}_1 \neq \mathtt{t}_2.\ \mathrm{G}^{\mathtt{t}_1}_{\mathrm{c}} \Rightarrow \mathrm{R}^{\mathtt{t}_2}_{\mathrm{o}}$ and $\forall \mathtt{t}_1, \mathtt{t}_2.\ \mathrm{G}^{\mathtt{t}_1}_{\mathrm{o}} \Rightarrow \mathrm{R}^{\mathtt{t}_2}_{\mathrm{c}}$

Proof. Trivial by definition of $\mathtt{RespectPartition}_o$ and $\mathrm{R}_{\mathrm{c}}, \mathrm{G}_{\mathrm{c}}$. □

Lemma 36 (client R/G/I properties).
$\mathsf{Sta}(\mathrm{I}_{\mathrm{c}}, \mathrm{R}^{\mathtt{t}}_{\mathrm{c}}) \wedge \forall \mathtt{t}_1, \mathtt{t}_2.\ \mathrm{G}^{\mathtt{t}_1}_{\mathrm{c}} \Rightarrow \mathrm{R}^{\mathtt{t}_2}_{\mathrm{c}}$

Proof. Trivial by definition. □

Theorem 37 (TSO obj. compositionality).
For any $f_1, \ldots, f_n$,
$\Pi_{\mathtt{sc}} = \{(tl_{\mathtt{sc}}, ge_1, \pi_1), \ldots, (tl_{\mathtt{sc}}, ge_m, \pi_m), (sl_{\mathtt{CImp}}, ge_o, \gamma_o)\}$,
$\Pi_{\mathtt{tso}} = \{(tl_{\mathtt{tso}}, ge_1, \pi_1), \ldots, (tl_{\mathtt{tso}}, ge_m, \pi_m), (tl_{\mathtt{tso}}, ge_o, \pi_o)\}$,

$$\frac{\begin{array}{c}\forall i \in \{1,\ldots,m\}.\ (tl_{\mathtt{sc}}, ge_i, \pi_i) \succcurlyeq^{\mathtt{c}}_{\mathtt{R}^{\mathtt{t}}_{\mathtt{c}};\mathtt{G}^{\mathtt{t}}_{\mathtt{c}};\mathtt{l}_{\mathtt{c}}} (tl_{\mathtt{tso}}, ge_i, \pi_i) \\ (tl_{\mathtt{tso}}, ge_o, \pi_o) \preccurlyeq^{\mathtt{o}} (sl_{\mathtt{CImp}}, ge_o, \gamma_o) \\ \mathsf{DRF}(\textbf{let}\ \Pi_{\mathtt{sc}}\ \textbf{in}\ \mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n) \\ \mathsf{Safe}(\textbf{let}\ \Pi_{\mathtt{sc}}\ \textbf{in}\ \mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n)\end{array}}{\textbf{let}\ \Pi_{\mathtt{tso}}\ \textbf{in}\ \mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n \leqslant_{ge_o} \textbf{let}\ \Pi_{\mathtt{sc}}\ \textbf{in}\ \mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n}$$

Proof. Mostly proved in Coq, assuming theorem 38 hold. The proof is similar to RGSim compositionality proof, except we need to prove the TSO execution will not have conflicting client footprints, which is the result of theorem 38. □

D.3.3 DRF implies no conflicting client footprint on TSO machine

Theorem 38 (DRF implies no conflicting client footprint)**.** For any $f_1, \ldots, f_n$,
$\Pi_{\mathtt{sc}} = \{(tl_{\mathtt{sc}}, ge_1, \pi_1), \ldots, (tl_{\mathtt{sc}}, ge_m, \pi_m), (sl_{\mathtt{CImp}}, ge_o, \gamma_o)\}$,
$\Pi_{\mathtt{tso}} = \{(tl_{\mathtt{tso}}, ge_1, \pi_1), \ldots, (tl_{\mathtt{tso}}, ge_m, \pi_m), (tl_{\mathtt{tso}}, ge_o, \pi_o)\}$,
if

1. $\forall i \in \{1,\ldots,m\}.\ (tl_{\mathtt{sc}}, ge_i, \pi_i) \succcurlyeq^{\mathtt{c}}_{\mathtt{R}^{\mathtt{t}}_{\mathtt{c}};\mathtt{G}^{\mathtt{t}}_{\mathtt{c}};\mathtt{l}_{\mathtt{c}}} (tl_{\mathtt{tso}}, ge_i, \pi_i)$, and
2. $(tl_{\mathtt{tso}}, ge_o, \pi_o) \preccurlyeq^{\mathtt{o}} (sl_{\mathtt{CImp}}, ge_o, \gamma_o)$, and
3. $\mathsf{Safe}(\textbf{let}\ \Pi_{\mathtt{sc}}\ \textbf{in}\ \mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n)$, and
4. $\mathsf{DRF}(\textbf{let}\ \Pi_{\mathtt{sc}}\ \textbf{in}\ \mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n)$, and

then $\forall \Sigma, \sigma.$, if $\mathsf{initM}(id, \mathsf{GE}(\Pi_{\mathtt{sc}}), \Sigma, \sigma)$, and $\left(\forall l \in ge_{\mathtt{lock}}.\ \mathtt{objM}(l, \Sigma)\right)$, and $(\textbf{let}\ \Pi_{\mathtt{sc}}\ \textbf{in}\ \mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n, \Sigma) \overset{load}{\Longrightarrow} \mathbb{W}_0$, and $(\textbf{let}\ \Pi_{\mathtt{tso}}\ \textbf{in}\ \mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n, \sigma) \overset{load}{\Longrightarrow}_{\mathtt{tso}} W_0$, then $\forall W', W''.\ W \Rightarrow^* W' \overset{\iota}{\underset{\delta}{\Rightarrow}} W''$ implies $\neg(\mathtt{clientFP}(\delta, \Sigma) \wedge \mathsf{conflict}_c(\delta, W'.\mathtt{t}, W'.\mathrm{B}))$

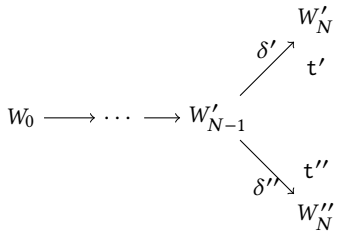
Proof. We prove by contradiction. Assume $\mathtt{clientFP}(\delta, \Sigma) \wedge \mathsf{conflict}_c(\delta, W'.\mathtt{t}, W'.\mathrm{B})$, we show we are able to construct SC execution with data race, which contradicts with premise 4.

By $\mathtt{clientFP}(\delta, \Sigma) \wedge \mathsf{conflict}_c(\delta, W'.\mathtt{t}, W'.\mathrm{B})$, we know there exists $W_1, \ldots, W_N, \delta_0, \ldots, \delta_{N-1}, \iota_0, \ldots, \iota_{N-1}$ such that
$\forall i \in \{0, \ldots, N-2\}$.
$W_i \overset{\iota_i}{\underset{\delta_i}{\Rightarrow}} W_{i+1} \wedge \neg(\mathtt{clientFP}(\delta_i, \Sigma) \wedge \mathsf{conflict}_c(\delta_i, W_i.\mathtt{t}, W_i.\mathrm{B}))$ and
$W_{N-1} \overset{\iota_{N-1}}{\underset{\delta_{N-1}}{\Longrightarrow}} W_N$ and
$\mathtt{clientFP}(\delta_{N-1}, \Sigma) \wedge \mathsf{conflict}_c(\delta_{N-1}, W_{N-1}.\mathtt{t}, W_{N-1}.\mathrm{B})$.

By $\mathsf{conflict}_c(\delta_{N-1}, W_{N-1}.\mathtt{t}, W_{N-1}.\mathrm{B})$, there exists one buffer item (l, v) in buffer of some thread such that $l \in \delta_{N-1}.rs \cup \delta_{N-1}.ws$. The buffer item (l, v) must be inserted during the execution from W_0 to W_N, i.e., there exists $M < N$ such that $W_M.\mathtt{t} \neq W_{N-1}.\mathtt{t}$ and $\delta_M \frown \delta_{N-1}$ and $\mathtt{clientFP}(\delta_M, \Sigma)$ and $l \in \delta_M.ws$, and (l, v) not unbuffered during the execution from W_M to W_N.

$$W_0 \longrightarrow \cdots \longrightarrow W_M \underset{\delta_M}{\longrightarrow} W_{M+1} \longrightarrow \cdots \longrightarrow W_{N-1} \underset{\delta_{N-1}}{\longrightarrow} W_N$$

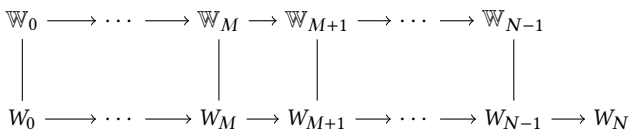
Now we reorder the execution, to postpone the execution of thread $W_M.\mathtt{t}$ after W_M, such that we could construct an execution trace as show below, such that $\forall i \in \{0, \ldots, N-2\}. \neg(\mathtt{clientFP}(\delta'_i, \Sigma) \wedge \mathsf{conflict}_c(\delta'_i, W'_i.\mathtt{t}, W'_i.\mathrm{B}))$ and $\delta' \frown \delta''$ and $l \in \delta'.ws$ and $\neg\mathsf{conflict}_c(\delta', \mathtt{t}', W'_{N-1}.\mathrm{B})$, where $\mathtt{clientFP}(\delta', \Sigma)$ and $\mathtt{clientFP}(\delta'', \Sigma)$.



Since thread local steps would not change TSO-memory (they only insert buffer items to buffer), we can safely reorder steps of different threads except unbuffer steps. To reorder normal (not unbuffer) step with unbuffer steps, we apply lemma 39 to reorder non-unbuffer step with unbuffer of the same thread, and lemma 40 to reorder non-unbuffer step with unbuffer of another thread.

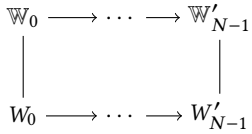
Since (l, v) not unbuffered during the execution from W_0 to W_N, we could always apply lemma 39 to reorder step M with unbuffer steps of the same thread.

To show premise of lemma 40 hold, it suffice to prove $\forall i.\ \neg\mathsf{bufConflict}(W_i.\mathrm{B}, \Sigma)$. Consider the invariant $\mathcal{R}$ between SC and TSO program configurations, as defined in definition 41 below. It is obvious that $\mathcal{R}(\mathbb{W}_0, W_0)$ holds.
By $\forall i \in \{0, \ldots, N-1\}. \neg(\mathtt{clientFP}(\delta_i, \Sigma) \wedge \mathsf{conflict}_c(\delta_i, W_i.\mathtt{t}, W_i.\mathrm{B}))$ and safety premise $\mathsf{Safe}(\textbf{let}\ \Pi_{\mathtt{sc}}\ \textbf{in}\ \mathtt{f}_1 \parallel \ldots \parallel \mathtt{f}_n)$, we could do induction on N and apply client or object local simulations to construct SC execution $\Sigma_1, \ldots, \Sigma_N$ such that $\forall i \in \{0, \ldots, N-1\}. \mathcal{R}(\mathbb{W}_i, W_i, \mathrm{R}_{\mathtt{o}}, \mathrm{G}_{\mathtt{o}}, \mathrm{I}_{\mathtt{o}})$, as shown below.

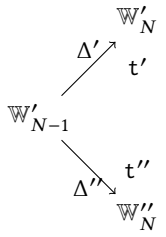


By definition of $\mathcal{R}$ and $\mathrm{I}_{\mathtt{c}}$, $\forall i < N.\ \neg\mathsf{bufConflict}(W_i.\mathrm{B}, \Sigma)$. Now we have reordered the execution to $W'_1, \ldots, W'_{N-1}$.

We do the above induction again on $W'_1, \ldots W'_{N-1}$, and construct SC execution $\mathbb{W}'_1, \ldots \mathbb{W}'_{N-1}$ such that $\forall i \in \{0, \ldots, N-1\}. \mathcal{R}(\mathbb{W}'_i, W'_i, \mathrm{R}_{\mathtt{o}}, \mathrm{G}_{\mathtt{o}}, \mathrm{I}_{\mathtt{o}})$, as shown below.



By $\mathcal{R}(\mathbb{W}'_{N-1}, W'_{N-1}, \mathrm{R}_{\mathtt{o}}, \mathrm{G}_{\mathtt{o}}, \mathrm{I}_{\mathtt{o}})$, $\mathtt{clientFP}(\delta', \Sigma)$, and $\mathtt{clientFP}(\delta', \Sigma)$, we can apply client simulation and there exists $\mathbb{W}'_N, \mathbb{W}''_N, \Delta', \Delta''$ such that $\Delta' = \delta'$ and $\delta''.rs \cup \delta''.ws \subseteq \Delta''.rs \cup \Delta''.ws$ and



Since $l \in \delta'.ws$ and $l \in \delta''.ws \cup \delta''.ws$, we have $\Delta' \frown \Delta''$, which indicates a racing execution that contradicts with premise 4.

□

Lemma 39 (Reordering unbuffer and write buffer step of the same thread).
$\forall \kappa, \mathrm{b}, \sigma, \delta, l, v, l', v'.$
if $(\kappa, ((l, v) :: \mathrm{b}, \sigma)) \xmapsto[\delta\ \mathtt{tso}]{\tau} (\kappa, ((l, v) :: \mathrm{b} +\!\!+ (l', v'), \sigma))$,
then $(\kappa, (\mathrm{b}, \sigma\{l \rightsquigarrow v\})) \xmapsto[\delta\ \mathtt{tso}]{\tau} (\kappa, (\mathrm{b} +\!\!+ (l', v'), \sigma\{l \rightsquigarrow v\}))$.

Proof. Trivial by x86-TSO semantics. □

Lemma 40 (Reordering unbuffer and write buffer step of different threads).
$\forall \kappa, \mathrm{b}, \sigma, \delta, l, v, l', v'.$ if $(\kappa, (\mathrm{b}, \sigma)) \xmapsto[\delta\ \mathtt{tso}]{\tau} (\kappa, (\mathrm{b} +\!\!+ (l', v'), \sigma))$ and $l \neq l'$,
then $(\kappa, (\mathrm{b}, \sigma\{l \rightsquigarrow v\})) \xmapsto[\delta\ \mathtt{tso}]{\tau} (\kappa, (\mathrm{b} +\!\!+ (l', v'), \sigma\{l \rightsquigarrow v\}))$.

Proof. It suffice to guarantee $l \notin \delta.rs \cup \delta.ws$. By x86-TSO semantics, a step with non-empty read and write footprint must have the same read/write set, i.e. $\delta.rs = \delta.ws = \{l'\}$. And we have $l \notin \delta.rs \cup \delta.ws$ by premise. □

Definition 41 (whole program simulation invariant).
$\mathcal{R}((\Gamma, \mathbb{FS}, \mathbb{T}, \mathtt{t}, d, \Sigma), (\Pi, \mathcal{F}, T, \mathtt{t}', \mathrm{B}, \sigma), \mathrm{R_o}, \mathrm{G_o}, \mathrm{I_o})$ iff

1. $\mathtt{t} = \mathtt{t}'$, and $d = 0$, and $\mathbb{FS} = \mathcal{F}$, and $\mathrm{I_c}(\Sigma, (\mathrm{B}, \sigma))$, and $\mathrm{I_o}(\Sigma, (\mathrm{B}, \sigma))$, and
2. $\forall \mathtt{t}_1.\ (\mathbb{T}(\mathtt{t}_1), \Sigma) \approx (T(\mathtt{t}_1), \mathrm{B}, \sigma)$, and
3. $\neg(\Gamma, \mathbb{FS}, \mathbb{T}, \mathtt{t}, d, \Sigma) \Longmapsto \mathsf{Race}$ and $\neg(\Gamma, \mathbb{FS}, \mathbb{T}, \mathtt{t}, d, \Sigma) \Rightarrow^* \mathbf{abort}$.

Here $(\mathbb{T}(\mathtt{t}), \Sigma) \approx (T(\mathtt{t}), \mathrm{B}, \sigma)$ iff
if $T(\mathtt{t}) = (F_1, \kappa_1) :: \ldots :: (F_k, \kappa_k) :: \epsilon$
and $\mathbb{T}(\mathtt{t}) = (\mathbb{F}_1, \mathbb{k}_1) :: \ldots :: (\mathbb{F}'_k, \mathbb{k}'_k) :: \epsilon$,
then

1. $k = k'$, and
2. $F_j = \mathbb{F}_j$ forall $j = 1 \ldots k$, and
3. $(\mathbb{k}_j, \Sigma) \succcurlyeq^{\mathrm{c}}_{\mathtt{t};\mathrm{R_c^t};\mathrm{G_c^t};\mathrm{I_c}} (\kappa_j, (\mathrm{B}, \sigma))$ forall $j > 1$, and
4. $(\mathbb{k}_1, \Sigma) \succcurlyeq^{\mathrm{c}}_{\mathtt{t};\mathrm{R_c^t};\mathrm{G_c^t};\mathrm{I_c}} (\kappa_1, (\mathrm{B}, \sigma))$ or $(\mathbb{k}_1, \Sigma) \succcurlyeq^{\mathrm{o}}_{\mathtt{t};\mathrm{R_o^t};\mathrm{G_o^t};\mathrm{I_o}} (\kappa_1, (\mathrm{B}, \sigma))$.

D.3.4 x86 assembly module client simulation

Lemma 42 (Client module simulation).
For any ge, π, $\mathtt{t}$, $(tl_{\mathtt{x86\text{-}SC}}, ge, \pi) \succcurlyeq^{\mathrm{c}}_{\mathrm{R_c^t};\mathrm{G_c^t};\mathrm{I_c}} (tl_{\mathtt{tso}}, ge, \pi)$

Proof. Since client code are identical, and CompCert memory model guaranteed x86-SC semantics to abort when accessing memory location with None permission (i.e., objM), it suffice to prove if $\mathrm{I_c}(\Sigma, (\mathrm{B}, \sigma))$ then reading/writing the same location l from/to Σ or (B, σ) yields results related by $\mathrm{I_c}$ and $\mathrm{G_c}$, if l does not conflict with B. I.e.,

1. $\forall l, \mathtt{t}.\ \neg\mathtt{objM}(l, \Sigma) \implies \Sigma(l) = \mathtt{apply\_buffer}(\mathrm{B}(\mathtt{t}), \sigma)(l)$, and
2. $\forall l, v, \mathtt{t}.\ \neg\mathtt{objM}(l, \Sigma) \wedge \neg\mathrm{conflict_c}((\emptyset, \{l\}), \mathtt{t}, \mathrm{B}) \implies \mathrm{I_c}(\Sigma\{l \rightsquigarrow v\}, (\mathrm{B}\{\mathtt{t} \rightsquigarrow \mathrm{B}(\mathtt{t}) +\!\!+ (l, v)\}, \sigma)) \wedge \mathrm{G_c^t}((\Sigma, (\mathrm{B}, \sigma)), (\Sigma\{l \rightsquigarrow v\}, (\mathrm{B}\{\mathtt{t} \rightsquigarrow \mathrm{B}(\mathtt{t}) +\!\!+ (l, v)\}, \sigma)))$

1 is obvious by definition of $\mathrm{I_c}$.

For 2, to show $\mathrm{I_c}(\Sigma\{l \rightsquigarrow v\}, (\mathrm{B}\{\mathtt{t} \rightsquigarrow \mathrm{B}(\mathtt{t}) +\!\!+ (l, v)\}, \sigma))$, it suffices to prove $\neg\mathrm{bufConflict}(\mathrm{B}\{\mathtt{t} \rightsquigarrow \mathrm{B}(\mathtt{t}) +\!\!+ (l, v)\})$, which is obvious by $\neg\mathrm{conflict_c}((\emptyset, \{l\}), \mathtt{t}, \mathrm{B})$. $\mathrm{G_c^t}((\Sigma, (\mathrm{B}, \sigma)), (\Sigma\{l \rightsquigarrow v\}, (\mathrm{B}\{\mathtt{t} \rightsquigarrow \mathrm{B}(\mathtt{t}) +\!\!+ (l, v)\}, \sigma)))$. The latter is also obvious by $\neg\mathtt{objM}(l, \Sigma)$.

□

D.4 Spinlock spec. and impl. satisfies our obj. correctness

We prove the lock specification and implementation in Fig. 10 satisfies our object correctness, i.e., $(tl_{\mathtt{tso}}, ge_{lock}, \pi_{lock}) \preccurlyeq^{\mathrm{o}} (sl_{\mathtt{CImp}}, ge_{lock}, \gamma_{lock})$.

Lemma 43 (Lock impl. correctness).
$(tl_{\mathtt{tso}}, ge_{lock}, \pi_{lock}) \preccurlyeq^{\mathrm{o}} (sl_{\mathtt{CImp}}, ge_{lock}, \gamma_{lock})$

Proof. To prove object correctness, the key is to find correct $\mathrm{R_o}, \mathrm{G_o}, \mathrm{I_o}$.

For the spinlock spec. and impl., we define $\mathrm{R_o}, \mathrm{G_o}, \mathrm{I_o}$ in Fig. 21, where L is the lock variable.

The invariant $\mathrm{I_o}$ says the lock variable has three possible states, which is

1. $\mathrm{I_{available}}$: L in both SC and TSO memory is available, and there is no buffered write to L
2. $\mathrm{I_{unavailable}}$: L in both SC and TSO memory is unavailable, and there is no buffered write to L
3. $\mathrm{I_{buffered}}$: L is available in SC memory, but unavailable in TSO memory and there is no buffered write to L

It is obvious that $\mathtt{RespectPartition_o}(\mathrm{R_o}, \mathrm{G_o}, \mathtt{objM})$ and $\mathtt{Sta}(\mathrm{I_o}, \mathrm{R_o^t})$ and $\forall \mathtt{t}_1 \neq \mathtt{t}_2.\ \mathrm{G_o^{t_1}} \Rightarrow \mathrm{R_o^{t_2}}$.

The proof of lock spec./impl. has the simulation $(sl_{\mathtt{CImp}}, ge_o, \gamma_o) \succcurlyeq^{\mathrm{o}}_{\mathrm{R_o^t};\mathrm{G_o^t};\mathrm{I_o}} (tl_{\mathtt{tso}}, ge_o, \pi_o)$ is also straight forward, and is mechanized in Coq, with a few admitted lemmas about TSO-memory.

□

$$
\begin{array}{rcl}
\mathsf{I}_o & \stackrel{\text{def}}{=} & \mathsf{I}_{\text{available}} \vee \mathsf{I}_{\text{buffered}} \vee \mathsf{I}_{\text{unavailable}} \\
\mathsf{G}^t_o & \stackrel{\text{def}}{=} & \mathsf{G}^t_{\texttt{lock}} \vee \mathsf{G}^t_{\texttt{unlock}} \vee \texttt{Id} \\
\mathsf{R}^t_o & \stackrel{\text{def}}{=} & \left(\bigvee_{t \neq t'} \mathsf{G}^{t'}_o\right) \vee \mathsf{R}_{\text{buffer}} \\
\mathsf{I}_{\text{available}}(\Sigma, (\mathrm{B}, \sigma)) & \stackrel{\text{def}}{=} & \Sigma(L) = (1, \text{None}) \wedge \sigma(L) = 1 \wedge (\forall\, t.\ L \notin \text{dom}(\mathrm{B}(t))) \\
\mathsf{I}_{\text{buffered}}(\Sigma, (\mathrm{B}, \sigma)) & \stackrel{\text{def}}{=} & \Sigma(L) = (1, \text{None}) \wedge \sigma(L) = 0 \wedge \exists\, \mathrm{t}, \texttt{buffered\_unlock}(\mathrm{t}, \mathrm{B}) \\
\mathsf{I}_{\text{unavailable}}(\Sigma, (\mathrm{B}, \sigma)) & \stackrel{\text{def}}{=} & \Sigma(L) = (0, \text{None}) \wedge \sigma(L) = 0 \wedge (\forall\, t.\ L \notin \text{dom}(\mathrm{B}(t))) \\
\texttt{buffered\_unlock}(\mathrm{t}, \mathrm{B}) & \stackrel{\text{def}}{=} & \mathrm{B}(t) = \mathrm{b} +\!\!+ (L, 1) :: \mathrm{b}' \wedge L \notin \text{dom}(\mathrm{b}) \cup \text{dom}(\mathrm{b}') \wedge \forall\, t' \neq t.\ L \notin \text{dom}(\mathrm{B}(t')) \\
\texttt{Id}((\Sigma, (\mathrm{B}, \sigma)), (\Sigma', (\mathrm{B}', \sigma))) & \stackrel{\text{def}}{=} & \Sigma = \Sigma' \wedge \sigma = \sigma' \wedge \mathrm{B} = \mathrm{B}' \\
\mathsf{G}^t_{\texttt{lock}}((\Sigma, (\mathrm{B}, \sigma)), (\Sigma', \mathrm{B}', \sigma')) & \stackrel{\text{def}}{=} & \Sigma(L) = 1 \wedge \sigma(L) = 1 \wedge \Sigma' = \Sigma\{L \rightsquigarrow (0, \text{None})\} \wedge \sigma' = \sigma\{L \rightsquigarrow 0\} \wedge \mathrm{B} = \mathrm{B}' \\
\mathsf{G}^t_{\texttt{unlock}}((\Sigma, (\mathrm{B}, \sigma)), (\Sigma', \mathrm{B}', \sigma')) & \stackrel{\text{def}}{=} & \Sigma(L) = 0 \wedge \sigma(L) = 0 \wedge \Sigma' = \Sigma\{L \rightsquigarrow 1\} \wedge \sigma = \sigma' \wedge \mathrm{B}' = \mathrm{B}\{t \rightsquigarrow \mathrm{B}(t) +\!\!+ (L, 1)\} \\
\mathsf{R}_{\text{buffer}}((\Sigma, (\mathrm{B}, \sigma)), (\Sigma', \mathrm{B}', \sigma')) & \stackrel{\text{def}}{=} & \mathsf{I}_{\text{buffered}} \ltimes \mathsf{I}_{\text{available}} \\
 & & \vee\, \mathsf{I}_{\text{available}} \ltimes \mathsf{I}_{\text{available}} \vee \mathsf{I}_{\text{buffered}} \ltimes \mathsf{I}_{\text{buffered}} \vee \mathsf{I}_{\text{unavailable}} \ltimes \mathsf{I}_{\text{unavailable}}
\end{array}
$$

Figure 21. $\mathsf{R}_o, \mathsf{G}_o, \mathsf{I}_o$ instantiations for spinlock