Per-Location Simulation

Liyi Li, Elsa L. Gunter {liyili2,egunter}@illinois.edu

Department of Computer Science,
University of Illinois at Urbana-Champaign

Abstract. Simulation/bisimulation is one of the most widely used frameworks for proving program equivalence/semantic preservation. In this paper, we propose a new per-location simulation (PLS) relation that is simple and suitable for proving that a compiled program semantically preserves its original program under a CFG-based language with a real-world, C/C++ like, weak memory model. To the best of our knowledge, PLS is the first simulation framework weaker than the CompCert [26]/CompCertTSO [47] one that is used for proving compiler correctness. With a combination of PLS, the compiler proof-framework Morpheus [33], and a language semantics with a weak memory model, we are able to prove that programs are semantically preserved through a transformation. All the definitions and proofs have been implemented in Isabelle/HOL.

1 Introduction

When the preservation of concurrency behavior was being verified in the CompCert compiler [26], the researchers found that it is not enough to tell the whole story just to use a traditional bisimulation framework to prove the program equivalence between a compiled program and its original one, so they designed a new bisimulation framework by treating safe programs and programs that might reach error states differently. CompCert's concurrency model was sequential consistency. The extent to which traditional bisimulation is inappropriate is even clearer when dealing with weak concurrency models. Weak concurrency models have been studied broadly for real world imperative programming languages (C/C++/LLVM) [1,9,5,18,44,37,10,3,45,36,23,22,24,39,19,21,14,40]. When using these models to prove compiler correctness, a problem arises. Historically, the semantics of these languages has been determined by the behavior of their compilers, so the behavioral effects of compiler optimizations also need to be considered in the concurrency models. For example, in the program piece (b) in Fig. 1, variables a and b can both read 1 if we consider the fact that a simple optimization removes the Boolean guards in (b), transforming it as the program piece (a). The well-known confusion about out-of-thin-air behaviors [8] is a typical consequence of the problem.

Researchers [39,19,21,40] have tried to solve the thin-air problems by merging the extra behaviors caused by compiler optimizations into their concurrency models. These models have several problems. Vafeiadis *et al.* [41] has shown

```
/* number after the blue "//" along with a read restricts the value of the read to be the number */
                           /* initially, x = 0 and y = 0 */

             (a)                               (b)                                (c)
 a :=rlx y//1 || b :=rlx x//1    a :=rlx y//1 || b :=rlx x//1     a :=rlx y//1 || b :=rlx x//1
 x :=rlx 1    || y :=rlx 1       if (a=a)     || if (b=b)         if (a=1)     || y :=rlx 1
                                    x :=rlx 1 ||    y :=rlx 1        x :=rlx 1 ||

     x :=rlx 1
     y :=rlx 1       a :=rlx y//1 || b :=rlx x//1       a :=rlx y//z || b :=rlx x//z || if (ram())
 (d) a :=rlx y   (e) if (a=1)     || if (b=1)       (f) if (a=a)     || if (b=b)     ||    z :=rlx 1
     b :=rlx x          x :=rlx 1 ||    y :=rlx 1          x :=rlx z ||    y :=rlx z ||  else
                                                                                     ||    z :=rlx 2
```

Fig. 1: Motivating Examples

that most of the compiler optimizations are invalid in these weak models. Moreover, Batty *et al.* [4] proved that it does not exists a candidate execution style axiomatic C++ concurrency model to incooperate the thin-air problems raised by (b) (Fig. 1). Additionally, these models are built upon a very limited set of memory actions or language pieces, and provide correct compiler schemes generated from the models based on the limited set. It requires a great effort to extend the schemes to prove a real-world compiler optimization preserving the multi-threaded program semantics for a real-world language under a real-world concurrency model. In some cases, even if the underlying language is extended a little, these models failed to show all supposedly allowed behaviors. For example, the promising model [21] is designed specifically to prove that the two reads in (d) (Fig. 1) can both read 1, but it fails to prove that the variables a and b in (f) can read any possible value from location z because z does not have a fixed value in all possible executions. The IMM model [40] is able to prove that the two reads in (a) can both read 1, but it fails to prove the two reads in (b) (Fig. 1) can both read 1. PLS is able to handle all these cases.

In this paper, we propose a simulation framework, named Per-Location Simulation (PLS), that is able to prove semantic preservation between compiled programs and their original programs under a language semantics with a weak concurrency model. We focus on safe traces (traces not going wrong) here, and assume that there is an outer layer on top of PLS to deal with reaching-error-state traces the same as the forward simulation framework in defined by CompCert [26]. As a main example, we provide a clear border for acceptable behaviors and out-of-thin-air behaviors in a CFG-based language with a weak concurrency model by using PLS to prove the semantic preservation of a simple optimization. The border is summarized by the examples in Fig. 1, which can be divided into two parts. The first is the PLS core part (Sec. 2.1). By the traditional simulation framework, the example (c) cannot be proven to simulate (a) (meaning that (a) semantically preserves (c)), because the memory trace (d) can be generated by (a), but it cannot be observed from (c). By analyzing closely the output of the two reads and two writes in both (a) and (c), all values that can be observed in these reads and writes of (c) can also be observed in (a). Thus, we should have a kind of similarity between (a) and (c). The PLS core produces such kind. It filters traces of programs into sub-traces based on locations. Instead of comparing the whole traces (as (d)), the PLS core compares the sub-traces of location x (and y) in (a) and (c) to determine if (c) per-location simulates (a). The second part is the full PLS definition (Sec. 2.3), which addresses the focal point of thin-air problems. (b) (Fig. 1) is supposed to be proved to be semantically preserved by (a), but (e) is not; because the two Boolean guards in (d) can be compiled away, but such guards in (e) cannot be removed. There are traces appearing in (a) but not appearing in (e). To validate the proof, we augment the PLS core with additional equations that capture some very simple compiler optimization syntactic dependencies. Instead of proving the simulation from (d) to (a), we prove the simulation from an equivalent representative of (d) to (a). To the best of our knowledge, PLS is the first simulation framework weaker than the one in CompCert/CompCertTSO [47], and to be used to prove compiler correctness under a CFG-based imperative language with a weak memory model, and is able to correctly distinguish thin-air and correct behaviors.

2 The Per-Location Simulation Definition

This section provides an introduction of PLS. We first introduce PLS core, then we provide an example language, and then we introduce the full PLS definition.

2.1 PLS Core

Transition System

States: $\sigma \in \varSigma$ Labels: $\alpha \in A$ Labeled Transition Systems (`LTS`): $(\varSigma, A, \xrightarrow{\alpha})$

Locations: Loc Label's Value: $\texttt{val}(\alpha)$ Label's Type: $\texttt{type}(\alpha)$ Label's Location: $\texttt{loc}(\alpha) \in Loc$

Transition System Property: $(\forall\alpha.\ \texttt{type}(\alpha) = \tau \Rightarrow \texttt{val}(\alpha) = \bot \wedge \texttt{loc}(\alpha) = \bot) \wedge (\bot \notin Loc)$

Transition System Syntactic Sugar

$\sigma \xrightarrow{\tau} \sigma' \triangleq \exists\alpha.\ \sigma \xrightarrow{\alpha} \sigma' \wedge \texttt{type}(\alpha) = \tau$ $\sigma \rightarrow_{\texttt{not}(x)} \sigma' \triangleq \exists\alpha.\ \sigma \xrightarrow{\alpha} \sigma' \wedge \texttt{loc}(\alpha) \neq x$

$\sigma \xrightarrow{\alpha}_x \sigma' \triangleq \exists\sigma_n\ \alpha.\ \sigma \rightarrow^*_{\texttt{not}(x)} \sigma_n \xrightarrow{\alpha} \sigma' \wedge \texttt{type}(\alpha) \neq \tau \wedge \texttt{loc}(\alpha) = x$

PLS Definition

Label Equivalence: $\alpha \equiv \beta \triangleq \texttt{val}(\alpha) = \texttt{val}(\beta) \wedge \texttt{type}(\alpha) = \texttt{type}(\beta) \wedge \texttt{loc}(\alpha) = \texttt{loc}(\beta)$

$\texttt{LTS}_{\varXi}$: $(\varXi, A, \xrightarrow{\alpha}^{\varXi})$ $\texttt{LTS}_{\varSigma}$: $(\varSigma, B, \xrightarrow{\beta}^{\varSigma})$

$\sqsubseteq_x$ is a $\texttt{PLS}_x$ relation on two transition systems $\texttt{LTS}_{\varXi}$ and $\texttt{LTS}_{\varSigma}$:

a.k.a. $\texttt{PLS}_x(\sqsubseteq_x) \triangleq$

$$\forall\xi\ \xi_1 \in \varXi.\ (\forall\sigma \in \varSigma\ (\forall\alpha \in A.\ \xi \sqsubseteq_x \sigma \wedge \xi \xrightarrow{\alpha}{}^{\varXi}_{x} \xi_1 \Rightarrow (\exists\beta\ \sigma_1.\ \sigma \xrightarrow{\beta}{}^{\varSigma}_{x} \sigma_1 \wedge \alpha \equiv \beta \wedge \xi_1 \sqsubseteq_x \sigma_1)))$$

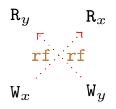
$\texttt{PLS}_{Loc}(\sqsubseteq) \triangleq \forall x \in Loc.\ \texttt{PLS}_x(\sqsubseteq_x)$

Fig. 2: Per-Location Simulation Core Definition

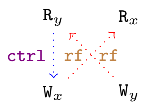
Here, we introduce the PLS core definition and utility examples. Fig. 2 includes the PLS core definition. We assume that there is a labeled transition system (LTS) $(\varSigma, A, \xrightarrow{\alpha})$, including a set of states ($\varSigma$), a set of labels (A), and a labeled transition function ($\xrightarrow{\alpha}$). The transition system is parameterized by a set of locations *Loc*. Every label in the transition system has three properties: its value (accessed by `val`), its type (at least having a τ type and normally having additional read and write types), and its memory location (be in the set *Loc*). For simplicity, we assume that if the type of a label is τ, then the value and location of the label are $\bot$ in a given transition system. To best describe the PLS core definition, we define some syntactic sugar on top of the transition system $\xrightarrow{\alpha}$ in Fig. 2. We first describe a predicate $\mathtt{PLS}_x$ defining PLS core on a single location x. A relation $\sqsubseteq_x$ is a PLS core relation on x over two labeled transition systems ($\mathtt{LTS}_\Xi$ and $\mathtt{LTS}_\Sigma$ in Fig. 2), if for any two states ($\xi \in \Xi$ and $\sigma \in \Sigma$) in the relation ($\xi \sqsubseteq_x \sigma$), ξ can transition by an x step (defined by $\xrightarrow{\alpha}_x$), then σ can also transition by an x step, where the two labels are equivalent ($\equiv$) and the resulting states are again related by $\sqsubseteq_x$. A family of relations ($\sqsubseteq$), one for each location in *Loc*, is a PLS core relation if each indexed relation ($\sqsubseteq_x$) satisfies $\mathtt{PLS}_x$ for each x in *Loc*, where *Loc* is a finite set of memory locations.

(wr_a) $\begin{array}{l} x :=_{\mathtt{rlx}} 1 \\ y :=_{\mathtt{rlx}} 1 \\ z :=_{\mathtt{rlx}} 1 \end{array}$ (wr_b) $\begin{array}{l} y :=_{\mathtt{rlx}} 1 \\ x :=_{\mathtt{rlx}} 1 \\ z :=_{\mathtt{rlx}} 1 \end{array}$ (prop) $x :=_{\mathtt{rlx}} 1 \wedge y :=_{\mathtt{rlx}} 1 \wedge z :=_{\mathtt{rlx}} 1$

We first discuss the single-threaded cases. The program pieces above ((wr_a) and (wr_b)) describe two sequences of memory writes. Regarding the underlying memory concurrency model, the outputs of the two program pieces should be the same, i.e. to write `1` to the locations x, y, and z. However, (wr_a) and (wr_b) cannot be proved to be similar with each other using the traditional simulation framework under the assumption of sequential consistency. Only if we assume a relaxed concurrency model can we prove that (wr_a) and (wr_b) are similar. In PLS, both (wr_a) and (wr_b) are viewed as three sub-traces as shown in (prop) (for simplicity, in each sub-trace, we only show instructions without mentioning other state environments), each of which describes a write for a location; so that we are able to prove that (wr_a) and (wr_b) are per-loc similar to each other.

(a_dd) 

$$\begin{tikzcd} \mathtt{R}_y & \mathtt{R}_x \\ \mathtt{W}_x \arrow[ur, dotted, "\mathtt{rf}"] & \mathtt{W}_y \arrow[ul, dotted, "\mathtt{rf}"'] \end{tikzcd}$$

(b_dd) 

$$\begin{tikzcd} \mathtt{R}_y \arrow[d, dotted, "\mathtt{ctrl}"'] & \mathtt{R}_x \\ \mathtt{W}_x \arrow[ur, dotted, "\mathtt{rf}"] & \mathtt{W}_y \arrow[ul, dotted, "\mathtt{rf}"'] \end{tikzcd}$$

(prop_a) $\begin{array}{l} x :=_{\mathtt{rlx}} 1 \\ b :=_{\mathtt{rlx}} x \end{array} \wedge \begin{array}{l} y :=_{\mathtt{rlx}} 1 \\ a :=_{\mathtt{rlx}} y \end{array}$

We now discuss multi-threaded cases under a weak relaxed memory model [24]. One of the example per-loc simulation relations that PLS enables is the one between programs (a) and (c) in Fig. 1, whose execution diagrams are listed as (a_dd) and (b_dd) above. An execution diagram is a graph representation of the execution of a program (only listing memory instructions), with arrows representing the memory instruction order that the execution must obey. In (a_dd), `W` (or `R`) represents a write (or a read) instruction with the subscripts (x or y) representing the memory locations (details are the $\mathcal{A}ct$ type in Fig. 3). The `rf` arrow between $\mathtt{R}_y$ and $\mathtt{W}_y$ in (a_dd) means that the read from y reads the value written by the write, so the read must happen after the write. The `ctrl` arrow in (b_dd) is a control dependency so that $\mathtt{R}_y$ must happen before $\mathtt{W}_x$ in any valid execution of (c) (Fig. 1). This is the reason that program (c) does not simulate program (a) by traditional simulation methods, i.e. because the execution ((d) in Fig. 1) happens in (a) but never happens in (c). On the other hand, PLS deals the two programs by first splitting all executions in both programs into a sub-trace per-location like the one in (prop_a). Thus, (c) per-loc simulates (a) ((a) semantically preserves (c)).

2.2 Example Language Syntax

Before we describe the full PLS definition, we first provide the example language that we will use in the paper. We focus on the syntax here, and the operational semantics in Sec. 3. The language is described in Fig. 3. In the figure, every name in *Chancery* font is a type defined for a language component; everything in `tt` font is a constructor or terminal in the language; and everything in *Italics* is a variable representing a term. The figure also introduces ranging conventions that will be employed throughout the paper.

Domains and Syntax

$a, b \in \mathcal{Var} \triangleq \mathcal{Name}$	Variables	$x, y \in \mathcal{Loc} \triangleq \mathbb{N}$	Memory Locations
$v \in \mathcal{Val} \triangleq \mathbb{Z}$	Integer Values	$\mathcal{O}_r \ni o_r \triangleq \mathtt{rlx} \mid \mathtt{acq}$	Read Orderings
$\mathcal{O}_w \ni o_w \triangleq \mathtt{rlx} \mid \mathtt{rel}$	Write Orderings		

Expressions: $\mathcal{Exp} \ni e \triangleq v \mid a \mid e \mathtt{+} e \mid e \mathtt{*} e \mid e \mathtt{=} e \mid e \mathtt{<} e$

Instructions: $\mathcal{Inst} \ni in \triangleq a \mathtt{:=} e \mid a \mathtt{:=}\ \mathtt{\&}x \mid a \mathtt{:=}_{o_r} (b \mid x) \mid (a \mid x) \mathtt{:=}_{o_w} b \mid \mathtt{skip}$

Terminations: $\mathcal{CInst} \ni c \triangleq \mathtt{if}\ e \mid \mathtt{br} \mid \mathtt{exit}$

CFG Labels: $\mathcal{L} \ni l \triangleq \mathtt{seq} \mid \mathtt{yes} \mid \mathtt{no}$	CFG Nodes: $\pi \in N \subseteq \mathbb{N}$
CFG Basic Block: $B \in \mathcal{B} \triangleq \mathcal{Inst}\ \mathcal{List} \times \mathcal{CInst}$	CFG Edges: $E \subseteq N \times \mathcal{L} \times N$
CFG Label Function: $\lambda : N \rightarrow (\mathcal{Inst}\ \mathcal{List} \times \mathcal{CInst})$	
Control Flow Graph (CFG): $\mathcal{CFG} \ni C \triangleq (N, \pi_0, \lambda, E)$	Thread IDs: $tid \in Tid \subseteq \mathcal{Tid} \triangleq \mathcal{Name}$
Memory Actions: $\mathcal{Act} \ni ac \triangleq \tau \mid \mathtt{R}^x_{v,o_r} \mid \mathtt{W}^x_{v,o_w}$	Programs: $\mathcal{Prog} \ni \mu \triangleq Tid \rightarrow \mathcal{CFG}$

Fig. 3: Example Language Syntax

The language contains a set of variables ($\mathcal{Var}$), integer values ($\mathcal{Val}$), and memory locations ($\mathcal{Loc}$). There are two kinds of memory orderings: one for read instructions ($\mathcal{O}_r$), and the other for write instructions ($\mathcal{O}_w$). The memory orderings are similar to the C++ memory orderings [24], but we only describe the relaxed (`rlx`), acquire (`acq`), and release (`rel`) orderings in this paper. There are two kinds of instructions: normal instructions and terminations. Normal instructions are in the LLVM style, where every instruction can produce no more than one assignment definition. For example, the purpose of the instruction $a \mathtt{:=}\ \mathtt{\&}x$ is to get the address of location x and put it in the variable a as an integer. Terminations include a binary branching instruction (`if`), an unconditional branching (`br`), and a control flow graph (CFG, type: $\mathcal{CFG}$) exit instruction (`exit`). A **control flow graph** (CFG) is a tuple (N, π_0, λ, E) where N is a finite set of nodes, π_0 is the start node, λ is a labeling of each node having a basic block that comprises a list of sequential instructions ended by a termination, and E is a set of edges labeled `seq`, `yes`, or `no` such that, if $\mathsf{snd}(\lambda(n)) = \mathtt{br}$ then there is a unique out-edge from n labeled `seq`; if $\mathsf{snd}(\lambda(n)) = \mathtt{exit}$ then there are no out-edges from n; and otherwise there are exactly two out-edges, one labeled `yes` and one labeled `no`. In a **basic block** ($\mathscr{B}$), we assign each instruction a **position number** (i), the sequential instructions their position in the list (starting from 0), and the termination the length of the list of sequential instructions, which is one greater than the position number of the last instruction in the list. Programs ($\mathscr{Prog}$) is a function from a set of thread-IDs (Tid) to CFGs. All program piece examples appearing in the paper are syntactic sugars of the programs in Fig. 3. Memory actions ($\mathscr{Act}$) are the core parts of the labels in the transition system described in the beginning of the section, and are viewed as a way for threads to communicate with the main memory.

2.3 Full PLS

The PLS core definition is suitable for building the per-loc simulation between (a) and (c) in Fig. 2, but the relation between (a) and (b) (Fig. 2) cannot be handled by the PLS core. To enhance the usability of PLS, we associate a reflexive relation `eq` with the PLS core definition as the Full PLS definition (Fig. 4).

PLS Definition

Reflexive Relations On Two LTSs: `eq`

$\mathtt{LTS}_{\Xi}$: $(\Xi, A, \xrightarrow{\alpha}{}^{\Xi})$ $\mathtt{LTS}_{\Sigma}$: $(\Sigma, B, \xrightarrow{\beta}{}^{\Sigma})$ $\mathtt{LTS}_{\Upsilon}$: $(\Upsilon, K, \xrightarrow{\kappa}{}^{\Upsilon})$

$$\begin{array}{l}
\mathtt{PLS}^{\mathsf{eq}}_{x}(\sqsubseteq_x) \triangleq \\
\quad \forall \xi\ \xi_1 \in \Xi.\ (\forall \alpha \in A.\ (\forall \sigma \in \Sigma.\ \xi \sqsubseteq_x \sigma \wedge \xi \xrightarrow{\alpha}{}^{\Xi}_{x} \xi_1 \Rightarrow \\
\qquad (\exists\ (\mathtt{LTS}_{\Sigma}, \mathtt{LTS}_{\Upsilon}, \preceq^{\mathsf{eq}}) \in \mathsf{eq}.\ (\exists \upsilon\ \upsilon_1 \in \Upsilon.\ (\exists \kappa \in K. \\
\qquad\quad \sigma \preceq^{\mathsf{eq}} \upsilon \wedge \upsilon \xrightarrow{\kappa}{}^{\Upsilon}_{x} \upsilon_1 \wedge (\exists \sigma_1 \in \Sigma.\ \sigma_1 \preceq^{\mathsf{eq}} \upsilon_1 \wedge \xi_1 \sqsubseteq_x \sigma_1)))))) \\
\mathtt{PLS}^{\mathsf{eq}}_{Loc}(\sqsubseteq) \triangleq \forall x \in Loc.\ \mathtt{PLS}^{\mathsf{eq}}_{x}(\sqsubseteq_x)
\end{array}$$

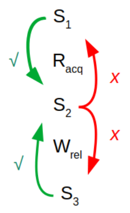
$$\begin{tikzcd}
\xi \arrow[r, "\forall"] & \xi_1 \\
\sqsubseteq & \sqsubseteq \\
\sigma & \sigma_1 \\
\preceq^{\mathsf{eq}} & \preceq^{\mathsf{eq}} \\
\upsilon \arrow[r, dotted, "\exists"] & \upsilon_1
\end{tikzcd}$$

Fig. 4: Full Per-Location Simulation Definition

The `eq` relation is at least a reflexive relation describing program transformations and capturing the syntactic dependencies of program instructions that are hard to be discovered by only the program concurrency semantics, such as the example (b) in Fig. 1. `eq` including the identity relation (as $\preceq^{\mathsf{eq}}$) relates two systems $\mathtt{LTS}_{\Sigma}$ and $\mathtt{LTS}_{\Upsilon}$, such as the tuple $(\mathtt{LTS}_{\Sigma}, \mathtt{LTS}_{\Upsilon}, \preceq^{\mathsf{eq}})$ in Fig. 4. $\mathtt{PLS}^{\mathsf{eq}}_{x}$ can be understood by the right diagram in Fig. 4. Assume that we have two systems $\mathtt{LTS}_{\Xi}$ and $\mathtt{LTS}_{\Sigma}$. We want to show the per-loc simulation ($\sqsubseteq_x$) from $\mathtt{LTS}_{\Xi}$ to $\mathtt{LTS}_{\Sigma}$ by showing that for every transition ξ to ξ_1, there exists a transition σ to σ_1, such that the two transition labels are equivalent ($\equiv$). However, we cannot directly have a transition from σ to σ_1 in some cases. Instead, through the `eq` set, we find a relation $\preceq^{\texttt{eq}}$ that relates $\texttt{LTS}_{\Sigma}$ with another system $\texttt{LTS}_{\Upsilon}$; and the transition from υ to υ_1 is found in $\texttt{LTS}_{\Upsilon}$, where υ and υ_1 are related to σ and σ_1 through $\preceq^{\texttt{eq}}$, respectively, and ξ_1 and σ_1 are also related by $\sqsubseteq_x$. $\texttt{PLS}_x^{\texttt{eq}}$ is a generalization of the $\texttt{PLS}_x$ predicate in Fig. 2, if we just select the tuple in `eq` as $(\texttt{LTS}_{\Sigma}, \texttt{LTS}_{\Sigma}, =)$. By selecting such tuple, the two systems $\texttt{LTS}_{\Sigma}$ and $\texttt{LTS}_{\Upsilon}$ are the same. Finally, $\texttt{PLS}_{Loc}^{\texttt{eq}}$ includes the functionality as $\texttt{PLS}_{Loc}$, but it builds a family of relations over the predicate $\texttt{PLS}_x^{\texttt{eq}}$.

$$
\begin{aligned}
&\texttt{eq} \triangleq \ldots\\
&\quad (N, \pi_0, \lambda \cup \{\pi \mapsto (ins, \texttt{if } e)\}, E' \cup \{(\pi, \texttt{ yes}, \pi_1), (\pi, \texttt{ no}, \pi_2)\})\\
&\quad \cong_{\texttt{eq}} (N, \pi_0, \lambda \cup \{\pi \mapsto (ins, \texttt{br})\}, E' \cup \{(\pi, \texttt{ seq}, \pi_1)\})\\
&\quad \texttt{IF eval}(e) = \texttt{true};\\
&\quad (N, \pi_0, \lambda \cup \{\pi \mapsto (ins, \texttt{if } e)\}, E' \cup \{(\pi, \texttt{ yes}, \pi_1), (\pi, \texttt{ no}, \pi_2)\})\\
&\quad \cong_{\texttt{eq}} (N, \pi_0, \lambda \cup \{\pi \mapsto (ins, \texttt{br})\}, E' \cup \{(\pi, \texttt{ seq}, \pi_2)\})\\
&\quad \texttt{IF eval}(e) = \texttt{false};\\
&\quad (N, \pi_0, \lambda \cup \{\pi \mapsto (ins, \texttt{if } e)\}, E' \cup \{(\pi, \texttt{ yes}, \pi_1), (\pi, \texttt{ no}, \pi_2)\})\\
&\quad \cong_{\texttt{eq}} (N, \pi_0, \lambda \cup \{\pi \mapsto (ins, \texttt{br})\}, E' \cup \{(\pi, \texttt{ seq}, \pi_1)\})\\
&\quad \texttt{IF } \lambda(\pi_1) = \lambda(\pi_2) \wedge (\forall l\ \pi'.(\pi_1, l, \pi') \in E \Leftrightarrow (\pi_2, l, \pi') \in E);\\
&\quad \ldots
\end{aligned}
$$

(a) Example `eq` Relation



(b) Roach Model on Acquire/Release Atomics

Fig. 5: Example and Roach Model

Fig. 5a provides a partial definition of an example `eq` set. The set contains equations to relate two labeled transition systems $\texttt{LTS}_{\Xi}$ and $\texttt{LTS}_{\Sigma}$ by relating the two program texts in any two states ξ and σ from the systems. The conditional equations shown in Fig. 5a is to equate two CFGs for a thread in any two program texts, i.e. two program texts μ and μ' are equivalent, if for any thread tid in the domain of μ/μ', $\mu(tid) \cong_{\texttt{eq}} \mu'(tid)$ ($\cong_{\texttt{eq}}$ means equivalence closed under the conditional equations in Fig. 5a). The first two conditional equations in Fig. 5a describe the equivalence relation that if a Boolean guard of a binary branching is always evaluated to `true` or `false` statically (by the `eval` function), then the CFG is related to the version formed by transforming the branching operation to a unconditional branching operation. The third rule describes the relation that if the outgoing edges of a branching block have the same target, then the CFG can be rewritten as a version only going through one branch.

The following single-threaded programs ((pa_a), (pa_b), and (pa_c)) are examples for which traditional simulation frameworks cannot provide satisfactory explanations. Using a sequential consistency model, a traditional simulation framework enables the proof of similarity between programs (pa_a) and (pa_b) (let's assume that the executions of a program generate an LTS), because an execution of (pa_a) always executes a write to x, then a read from y, and then a write to z, which is the same sequence as the one produced by (pa_b). The problem is that we also want to show that (pa_a) and (pa_c) are similar, which the traditional framework cannot enable.

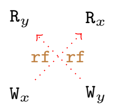
(pa_a)
```
x :=rlx c
if (a=b ∧ b=c)
    a :=rlx y
 else
    a :=rlx y
z :=rlx b
```

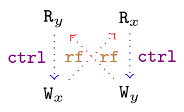
(pa_b)
```
x :=rlx c
a :=rlx y
z :=rlx b
```

(pa_c)
```
a :=rlx y
z :=rlx b
x :=rlx c
```

(prop_pa) $x :=_{\texttt{rlx}} c \wedge a :=_{\texttt{rlx}} y \wedge z :=_{\texttt{rlx}} b$

Under a weak memory model, like RC11 [24], a transitional simulation method enables the proof that (pa_c) simulates (pa_a) but not the opposite, because the Boolean guard in (pa_a) contains the variables a and b, so it has data dependency on the later instructions (read from y and write to z). Thus, they cannot move to execute before the Boolean guard as well as the write to x. Clearly, by using the full PLS, to prove that (pa_a) simulates (pa_c), we can first find an equivalent program of (pa_a), which is exactly the one in (pa_b). Then we prove that (pa_a) per-loc simulates (pa_c) by showing that (pa_b) per-loc simulates (pa_c).

(a_dd)


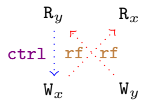
(d_dd)


(prop_a) $\begin{matrix} x :=_{\texttt{rlx}} 1 \\ b :=_{\texttt{rlx}} x \end{matrix} \wedge \begin{matrix} y :=_{\texttt{rlx}} 1 \\ a :=_{\texttt{rlx}} y \end{matrix}$

The simulation from (pa_a) to (pa_c) can also be proved by the PLS core definition in Sec. 2.1. To understand the additional proving ability that the full PLS brings us, the simulation from (b) to (a) in Fig. 1 provides a better hint. The execution diagram of (a) is shown as (a_dd) above, while the diagram of (b) is shown as (d_dd). In (d_dd), for every single thread, a control dependency (`ctrl`) exists from the read to the write. If we observe that the two reads both read `1`, we have exactly two reads-from edges (`rf`) from writes to reads. Thus, the diagram contains a cycle, which means that the execution of reading both as `1` is impossible if no optimization is applied to (b) (Fig. 1). Like the traditional simulation frameworks, PLS core is unable to prove the per-loc simulation from (b) to (a), which is the correct behavior in the sense that no optimization is applied. the desired simulation between (b) and (a) must take into account some resulting behaviors caused by optimizations. It is clear that the two `ctrl` edges in (d_dd) can be removed by some very simple optimizations, so that (b) becomes (a); and its execution diagram is the same as that of (a_dd). Then we can use the PLS core to build the simulation relation as the one in Sec. 2.1. This is the main content of the full PLS definition, which includes the optimization effects as the equivalence relation eq, then proves the per-loc similarity from an equivalence representative of (b) to (a) by using the PLS core.

(par)
```
a :=rlx y//1    || b :=rlx x//1
if (a=1)        || if (b=1)
      x :=rlx 1 ||       y :=rlx 1
                ||    else
                ||       y :=rlx 1
```

(par')
```
a :=rlx y       || b :=rlx x
if (a=1)        || y :=rlx 1
      x :=rlx 1 ||
```

(par'_dd)


If a traditional simulation framework would be parameterized with the `eq` relation, it could prove the simulation from (b) to (a) in Fig. 1, but it is inadequate for the simulation from the (par) above to (a) (Fig. 1). For that, the full power of PLS is required. To prove such a per-loc simulation, we first select an equivalence representative of program (par) to be the program (par'). Then, we prove the per-loc simulation from (par') to (a) by the strategy for proving the relation from (c) to (a) (Sec. 2.1).

We then need to answer the question: what kind of equations are allowed in `eq`? The principle is described in the Roach Model of Manson *et al.* [34], and systematically explained by Vafeiadis *et al.* [42]: the short answer is any equation that can preserve program meaning, especially, the meaning of the critical section created by the acquire (`acq`) and release (`rel`) atomic memory operations. Essentially, the acquire/release atomics are C++ memory devices that implement a weak version of the memory locking mechanism. Moving a memory operation before an acquire atomic operation or after a release atomic operation violates the Roach Model principle that states: "shared memory accesses can be moved in critical regions but not out of them" (Fig. 5b). In the paper of Vafeiadis *et al.*, several cases are mentioned of an optimization violating this principle; each of them involves the removal or addition of read/write memory operations. For simplicity in this paper, we provide the following observation about a conservative construction of `eq` to preserve the Roach Model principle. In it, $\texttt{LTS}|_{tid}$ means chopping the LTS to only execute single-threaded CFGs in the thread *tid*.

Observation 1 Assume that we have a transition system $\texttt{LTS}_{\Sigma}$, and a singleton relation set $\texttt{eq} = \{(\texttt{LTS}_{\Sigma}, \texttt{LTS}_{\Xi}, \sim)\}$. We assume that for every thread *tid*, we derive two single-threaded systems from $\texttt{LTS}_{\Sigma}$ and $\texttt{LTS}_{\Xi}$ as $\texttt{LTS}_{\Sigma}|_{tid}$ and $\texttt{LTS}_{\Xi}|_{tid}$, the $\sim$ relation ($\texttt{LTS}_{\sigma}|_{tid} \sim \texttt{LTS}_{\sigma}|_{tid}$) has the property that $\texttt{LTS}_{\sigma}|_{tid}$ is bisimilar to $\texttt{LTS}_{\sigma}|_{tid}$. Then, for any relation $\sqsubseteq$, such that $\texttt{PLS}^{\texttt{eq}}_{Loc}(\sqsubseteq)$, and any state ξ in $\texttt{LTS}_{\Xi}$ that does not transition (in $\texttt{LTS}_{\Xi}$) to a Roach-Model-violating state (Fig. 5b), if $\sigma \sqsubseteq \xi$, then σ does not transition (in $\texttt{LTS}_{\Sigma}$) to a Roach-Model-violating state.

3 Program Meaning Preservation

Morpheus is a a domain-specific language for formal specification of program transformations. In previous papers about Morpheus [31,33], it was shown how to combine a sequential memory model, the Morpheus framework, and an underlying instruction semantics for a programming language to prove the correctness of a traditional compiler optimization (PRE). This section introduces a combination of PLS, Morpheus, and the program semantics for the language in Fig. 3 (based on a weak memory model) to prove an optimization semantically preserving the program meaning. We first introduce the Morpheus specification language and examples of optimizations specified in Morpheus. Then, we introduce the program semantics of the language in Fig. 3, which combines the instruction semantics with a weak memory model. Given an optimization ζ and program μ, we rewrite μ to μ' by ζ. The proof is to build a PLS relation from a LTS, whose states have the form (μ', ω) for any environment state ω with a fixed format (Fig. 8), to another LTS, whose states have the form (μ, ω).

3.1 Morpheus and Example Optimization Specifications

The Morpheus specification language [33] is enlightened by the Trans language of Kalvala *et al.* [20]. Morpheus specifies an optimization as conditional compositions of rewrites on CFGs. Morpheus is split into three components: core graph transformations, conditions given in First Order Computation Tree Logic (FOCTL), and a strategy language for building complex transformations out of component transformations and conditions. The details of the Morpheus syntax are in the work [33] (and in Appendix 6.1). Conceptually, for an optimization specified in Morpheus, the rewrite portion expresses the local transformation to be made, the condition characterizes the situations in which the optimization should be applied, and the strategy language allows us to build a combination of transformations out of collections of local ones. Morpheus is a special-purpose language for the transformation of CFGs, and as such is parameterized by aspects of CFGs, namely node names (π in Fig. 3), node contents (program basic block B in Sec. 3), and edge labels marking control flow (l in Fig. 3). Transformation specifications may mention aspects of CFGs concretely, but more generally, they use pattern variables that will be instantiated with control flow graph components in each specific application. We will use the term "expressions" to refer to patterns built from both concrete entities and metavariables (which will be instantiated with concrete entities when the transformation is applied). We use the term **metavariable** (range: a and b) to refer to the variables in the patterns and expressions in Morpheus transformations, as opposed to the concrete programming variables and memory locations that will be found in Fig. 3.

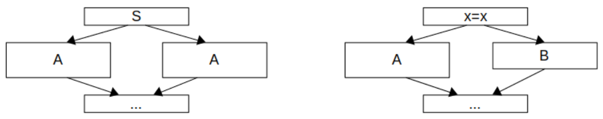


Fig. 6: Examples of Simple Code Motion Optimizations

In this paper, we use two kinds of simple code motion (SCM) optimizations as examples. The general strategies for them are shown as graphs in Fig. 6. Given a CFG C for a thread in a program μ, the left optimization in Fig. 6 locates (by a Morpheus condition expression) a basic block B of C, whose termination is a binary branching instruction and the two outgoing edges pointing to the two basic blocks B_1 and B_2 that have the same content and same outgoing edges. Then the left optimization changes the binary branching instruction in B to a non-conditional one, and also changes the edges of B to a single outgoing edge with a label `seq`. This is done by a strategy code in Morpheus with a sequence of graph transformations. Similarly, the right optimization first locates a basic block B of C whose termination is a binary branching instruction whose Boolean guard is always evaluated as `true` (by static rewriting). Then the optimization changes the binary branching instruction to an unconditional branching one `br`, and makes all of the outgoing edges of B point to the basic block indicated by the `true` branching of B.

$$\begin{aligned}
&\texttt{sameOutEdge}(a,b) \triangleq \texttt{stmt}(a) = \texttt{stmt}(b) \land \texttt{sameEdges}(a,b)\\
&\qquad\qquad\qquad\quad \lor\texttt{stmt}(a) = \texttt{stmt}(b) \land \neg\texttt{sameEdges}(a,b) \land \texttt{sameOutEdge}(\texttt{next}(a),\texttt{next}(b))\\[4pt]
&\texttt{leftOpt}(\pi) \triangleq \texttt{EXISTS}\ \pi_1\ \pi_2.\texttt{SATISFIED\_AT}\ \pi.\texttt{sameOutEdge}(\texttt{next}(\texttt{yes},\pi),\texttt{next}(\texttt{no},\pi))\\
&\qquad\qquad\quad ;\texttt{relabel\_node}(\pi,(\texttt{insts}(\pi),\texttt{br}));\texttt{move\_edge}((\pi,\texttt{no},\pi_2),(\texttt{seq},\pi_1))\\
&\qquad\qquad\quad ;\texttt{move\_edge}((\pi,\texttt{yes},\pi_1),(\texttt{seq},\pi_1))\\[4pt]
&\texttt{rightOpt}(\pi) \triangleq \texttt{EXISTS}\ a\ \pi_1\ \pi_2.\texttt{SATISFIED\_AT}\ \pi.\texttt{tem\_inst}(\pi) = a \land \texttt{eval}(a) = \texttt{true}\\
&\qquad\qquad\qquad ;\texttt{relabel\_node}(\pi,(\texttt{insts}(\pi),\texttt{br}));\texttt{move\_edge}((\pi,\texttt{no},\pi_2),(\texttt{seq},\pi_1))\\
&\qquad\qquad\qquad ;\texttt{move\_edge}((\pi,\texttt{yes},\pi_1),(\texttt{seq},\pi_1))
\end{aligned}$$

Fig. 7: Simple Code Motion Transformations in Morpheus

Figure 7 contains the Morpheus formulas `leftOpt` and `rightOpt` defining the left and right compiler optimizations from Fig. 6. The `sameOutEdge` formula defines the predicate for checking if two statements are the same; and their children have the same outgoing edges or statements. a and b are two metavariables representing two nodes; the `stmt`(π) function gets the basic block represented by node π, and the `sameEdges` predicate checks if a and b have the same out going edges. `leftOpt` represents the left optimization in Fig. 6. It first searches a node π that has a binary branching instruction with two out going edges (defined by the `SATISFIED_AT` Morpheus strategy operation). The `next` function gets the outgoing node of π with a fixed edge label (`yes` or `no` in `leftOpt`). It does three actions: first, it replaces the termination of π with `br` (by the Morpheus `relabel_node` action); second, it changes the `no` edge of π to π_1 with the label `seq` (by the Morpheus `move_edge` action), and finally it exchanges the `yes` edge of π with the label `seq` (also by the Morpheus `move_edge` action). The `insts` function gets the instruction list in the basic block of π. The `rightOpt` formula implements the right optimization in Fig. 6. It is similar to `leftOpt`. The only difference is that it checks if the binary branching instruction in the basic block of node π has a Boolean guard that is always evaluated as `true` (by the `eval` function). The termination is retrieved by the `tem_inst` function, and the metavariable a represents the termination of π.

The semantics of Morpheus [31,33] is basically the implementation of a graph rewrite algorithm over the FOCTL style conditions. Given an optimization formula (like Fig. 7) and a program μ, for every CFG C for a thread in μ, the algorithm generates a set of new CFGs. It first locates a basic block node satisfying the condition φ defined in a `SATISFIED_AT` strategy operation; and then it does a series of actions that change the structure of the CFG based on the node, as with the `relabel_node` and `move_edge` actions in `leftOpt`.

Here, we have briefly introduced Morpheus and given examples of optimizations defined therein. We will introduce program semantics in the next section.

3.2 Example Language Semantics under a Weak Memory Model

Domain

Time Points: $s, t \in T \subseteq \mathbb{N}$ Registers: $\varphi \subseteq (\mathit{Var} \rightarrow \mathit{Val})$ Heap Snapshots: $\gamma \subseteq (\mathit{Loc} \rightarrow (T \times \mathit{Val}))$

Thread-IDs: $tid \in Tid \subseteq \mathcal{Tid}$ Dyanmic Block Number: $\mathcal{Bn} \ni \overline{\pi} \triangleq (N \times N)$

Dyanmic Block Family: $\overline{\Pi} \subseteq Tid \rightarrow (N \times N)$ Action-IDs: $\mathcal{Aid} \ni d \triangleq (\mathcal{Bn} \times N)$

Semantic Function Types

Expression Semantics: $\texttt{eval} \subseteq (\mathit{Exp} \times \varphi) \rightarrow \mathit{Val}$ Inst Semantics: $\psi \subseteq (\mathit{Inst} \times \varphi \times \gamma) \rightarrow (\varphi \times \mathit{Act})$

Termination Semantics: $\eta \subseteq (\mathit{CInst} \times (\mathit{Var} \rightarrow \mathit{Val})) \rightarrow \mathcal{L}$

Example Instruction Semantics

$\psi(a \texttt{ := } e, \varphi, \gamma) \triangleq (\varphi[a \leftarrow \texttt{eval}(\varphi, e)], \tau)$ Assignment Semantics

$\psi(a \texttt{ :=}_{o_r}\ x, \varphi, \gamma) \triangleq (\varphi[a \leftarrow \texttt{snd}(\gamma(x))], \texttt{R}^{x}_{\texttt{snd}(\gamma(x)), o_r})$ Read Semantics

$\psi(x \texttt{ :=}_{o_w}\ a, \varphi, \gamma) \triangleq (\varphi, \texttt{W}^{x}_{\texttt{eval}(\varphi, a), o_w})$ Write Semantics

$\eta(\texttt{if } e \texttt{ then } \pi_1 \texttt{ else } \pi_2, \varphi) \triangleq \texttt{IF } \eta(\varphi, e) = 0 \texttt{ THEN yes ELSE no}$ Binary Branching Semantics

Single-Threaded Memory Concurrency Model

Events: $\mathcal{Ev} \ni ev \triangleq Tid \times \mathcal{Aid} \times \mathcal{Act}$ Execution Map: $\rho \subseteq T \rightarrow \mathcal{Ev}$

Data Dependency: $\texttt{dd} \subseteq T \times T$ Data Dependency Family: $\texttt{dds} \subseteq Tid \rightarrow (T \times T)$

Sequenced-Before Relation: $\texttt{sb} \subseteq T \times T$ Sequenced-Before Family: $\texttt{sbs} \subseteq Tid \rightarrow (T \times T)$

Acquire Dependency: $\texttt{acq}(T, \rho, \texttt{sb}) \triangleq \{(s, t) \in T^2 | (s, t) \in \texttt{sb} \wedge \texttt{is\_read}(\rho(s)) \wedge \texttt{is\_acq}(\rho(s))\}$

Release Dependency: $\texttt{rel}(T, \rho, \texttt{sb}) \triangleq \{(s, t) \in T^2 | (s, t) \in \texttt{sb} \wedge \texttt{is\_rel}(\rho(t)) \wedge \texttt{is\_write}(\rho(t))\}$

True Program Order: $\texttt{po}(\texttt{T}, \rho, \texttt{sb}, \texttt{dd}) \triangleq \texttt{dd} \cup \texttt{acq}(\texttt{T}, \rho, \texttt{sb}) \cup \texttt{rel}(\texttt{T}, \rho, \texttt{sb})$

$\texttt{single\_prop}(Tid, T, \rho, \texttt{sbs}, \texttt{dds}, \texttt{rf}) \triangleq \forall (s, t) \in (\texttt{rf} \cup \bigcup_{tid \in Tid} \texttt{po}(T, \rho, \texttt{sbs}(tid), \texttt{dds}(tid))).\ s < t$

$\texttt{sat}(Tid, T, \rho, \texttt{sbs}, \texttt{dds}, \texttt{rf}) \triangleq \texttt{single\_prop}(Tid, T, \rho, \texttt{sbs}, \texttt{dds}, \texttt{rf}) \wedge \ldots$

Operational Program Semantics

Program Order Family: $\texttt{pos} \subseteq Tid \rightarrow \texttt{po}$ Current Program Pointer: $\theta \subseteq \mathit{In}\ \mathcal{Set} \times \mathit{In}\ \mathcal{Set}$

Registers Family: $\Phi \subseteq Tid \rightarrow \varphi$ Heap Family: $\Gamma \subseteq Tid \rightarrow \gamma$

Program Pointer Family: $\Theta \subseteq Tid \rightarrow \theta$

Single Step Transition Function:

$\texttt{trans} \subseteq (C, Tid, \overline{\pi}, \texttt{po}, \texttt{sb}, \texttt{dd}, T, \rho, \varphi, \Gamma, \theta) \rightarrow (\overline{\pi}, \texttt{po}, \texttt{sb}, \texttt{dd}, T, \rho, \varphi, \Gamma, \theta, \texttt{rf})$

State Environment: $\omega \triangleq (Tid\ \mathcal{Set}, \texttt{pos}, \texttt{sbs}, \texttt{dds}, \overline{\Pi}, \Phi, \Gamma, \Theta, T, \rho, \texttt{rf})$

State: (μ, ω) Transition System: $(\mu, \omega) \xrightarrow{ev} (\mu, \omega)$

One Example Transition Rule:

$$\frac{\begin{array}{l} tid \in Tid \wedge \texttt{pos}' = \texttt{pos}[tid \mapsto \texttt{po}'] \wedge \texttt{sbs}' = \texttt{sbs}[tid \mapsto \texttt{sb}'] \wedge \texttt{dds}' = \texttt{dds}[tid \mapsto \texttt{dd}'] \\ \wedge \overline{\Pi}' = \overline{\Pi}[tid \mapsto \overline{\pi}] \wedge \Phi' = \Phi[tid \mapsto \varphi'] \wedge \Theta' = \Theta[tid \mapsto \theta'] \wedge \texttt{sat}(Tid, T', \rho, \texttt{sbs}', \texttt{dds}', \texttt{rf} \cup \texttt{rf'}) \\ \wedge \texttt{trans}(\mu(tid), tid, \overline{\Pi}(tid), \texttt{pos}(tid), \texttt{sbs}(tid), \texttt{dds}(tid), T, \rho, \Phi(tid), \Gamma, \Theta(tid)) \\ \quad = (\overline{\pi}', \texttt{po}', \texttt{sb}', \texttt{dd}', T', \rho', \varphi', \Gamma', \theta', \texttt{rf'}) \end{array}}{\begin{array}{l} (\mu, Tid, \texttt{pos}, \texttt{sbs}, \texttt{dds}, \overline{\Pi}, \Phi, \Theta, \Gamma, T, \rho, \texttt{rf}) \\ \qquad \xrightarrow{\rho'(\texttt{max}(T'))} (\mu, Tid, \texttt{pos}', \texttt{sbs}', \texttt{dds}', \overline{\Pi}', \Phi', \Theta', \Gamma', T', \rho', \texttt{rf} \cup \texttt{rf'}) \end{array}}$$

Fig. 8: Language Semantics with a Weak Memory Model

Here we discuss the operational program semantics for the language in Fig. 3 to support the proof in this section. The semantics is a bridge connecting single instruction semantics and a multi-threaded weak memory model. Fig. 8 provides a taste of the instruction semantics, memory concurrency model, and operational semantics based on the language in Fig. 3. In Fig. 8, T is a downward closed natural number set of **time points** without `0`, each of whose elements represent a "time" when an instruction executes in a program. We implement a heap snapshot (γ) as a function from a location to a pair: the pair is the time point of the most recent write to the location and the value in the location. In Fig. 3, we introduced the concept of basic blocks, nodes are numbers identifying basic blocks. We use a pair of natural numbers as a **dynamic basic block number** ($\mathcal{Bn}$); the pair uniquely identify an executing basic block in a thread during an execution. For a program $\mu : Tid \rightarrow \mathcal{CFG}$, we have a family of dynamic basic block numbers ($\overline{\Pi}$), one for each thread. In Sec. 2.2, we introduced an instruction number for each instruction in a basic block; it is represented by a natural number. Here, we name an **action-ID** as the combination of a dynamic block number $\overline{\pi}$ and an instruction number in the basic block indexed by the second argument of $\overline{\pi}$. Hence, it is clear that an **action-ID** can uniquely define an executing instruction in a thread.

At the instruction level, there are three semantic functions. The `eval` function (Fig. 8) is for evaluating an expression ($\mathcal{Exp}$) in Fig. 3. It is a straight evaluation of each term of the expression, so we omit the detailed implementation here. The function ψ implements the semantics of an instruction ($\mathcal{Inst}$ in Fig. 3). It takes an instruction, a register map (φ), and a heap snapshot (γ), and produces a resulted register map and a memory action ($\mathcal{Act}$) indicating the type of memory communication the instruction could bring. We show three cases for ψ in Fig. 8: the case when a normal assignment happens and ψ returns a τ action, the case when a read happens and ψ returns a read action, and the case when a write happens and ψ returns a write action. The function η implements the semantics of terminations. It takes a termination and registers, and returns an edge label. In Fig. 8, we show the semantics of a binary branching instruction.

The memory concurrency model is in the format of an axiomatic candidate execution model [1]. Here, we use a subset of the ATRCM model [27], which has been proved to be sound with respect to the C/C++ memory model defined by Lahav *et al.* [24] and the IMM model [40]. The basis of the model is an execution with a pair of time points T and a function ρ mapping T to memory events. By defining a set of binary relations and predicates on top of the pair, the model selects a valid set of memory executions from a set of candidate executions. If there is a pair (s, t) in one of the relations, the memory event $\rho(s)$ must happen before the event $\rho(t)$ in the execution. In Fig. 3, we also introduced actions ($\mathcal{Act}$). A memory instruction produces a read/write action, while other instructions/terminations produce a τ action. Here we combine an action, thread-ID and action-ID, making a memory event ($\mathcal{Ev}$). In the model in Fig. 8, for an execution, we assume that a family (`sbs`) of sequenced-before relations (`sb`) is given, one `sb` for each thread; and a family (`dds`) of data dependency relations (`dd`) is also given, one `dd` for each thread. A straight-forward algorithm for generating a sequenced-before relation for executing a CFG can be taken from the program text order of the CFG; also, a data dependency relation for a CFG execution can be produced by the traditional data-flow, alias, and control dependency analysis algorithms. Here we omit the details of these algorithms. In Fig. 8, we mainly introduce the single-threaded relations and predicates for the model. More details are in the Appendix 6.2. In a single thread, we assume that instructions can be executed out-of-order. We define a single-threaded program order relation (`po`) for each thread to restrict the out-of-order execution, to respect the program meaning. `po` is the union of the single-threaded data dependency (`dd`) relation, `acq` relation for acquire (`acq`) reads, and `rel` relation for release (`rel`) writes. In the definition of the relation for the acquire read instruction, we require no instruction sequenced-after the acquire read can be executed before the read; while in the relation for a release write instruction, no instruction sequenced-before the release write can be executed after it. These two relations can be better understood by the Roach Model in Fig. 5b. The `single_prop` predicate represents all single-threaded behaviors that an execution must satisfy. The predicate requires that any single-threaded `po` relation and the reads-from relation (`rf`) in an execution must not have a pair of time points (s, t), for which t happens before s. The `sat` predicate is the collection of all predicates that a valid execution must satisfy. It is detailed in the Appendix 6.2, and includes the `single_prop` predicate.

The operational transition semantics in Fig. 8 is a combination of the instruction level semantics and memory concurrency model. It is represented as a labeled transition system whose states are pairs of programs (μ) and the state environment (ω), and whose labels are memory events. A state environment is a long tuple of a set of thread-IDs (Tid), a program order family (`pos`, one for each thread), a sequenced-before relation family (`sbs`), a data dependency family (`dds`), a current dynamic block number family $(\overline{\Pi})$, a registers family (Φ), a heap snapshot family (Γ) representing different views of the threads of the main memory, a program pointer family (Θ) representing the current executing instruction of each thread, a time point set (T), a ρ mapping, and a reads-from relation (`rf`). We show the top-most rule of the transition system in Fig. 8. This rule selects a thread tid, applies the one-step transition function `trans` to the state environment of tid, checks the result of the one-step transition to see if the accumulated result satisfies the predicate of the memory model (`sat`), and then moves forward to a new step via the memory event label $\rho'(\texttt{max}(T'))$. The function `max` produces the maximum number in T'. We can retrieve the memory event by the `max` function because the `trans` function always creates a map entry in ρ from the maximum time point plus `1` to the current memory event. The detailed implementation of the `trans` function is found in the Appendix 6.3. It needs to finish several tasks as a one step evaluation for a thread tid with a CFG C. First, if its program pointer $\Theta(tid)$ points to the end of a basic block (no instructions left for execution), it selects a new basic block according to the edge information in C (applying function η to it with registers $(\Phi(tid))$ to get the edge label), and assigns a new dynamic basic block number with a new program pointer pointing to the top of the new block. In this case, `trans` also adds new relations of program order, sequenced-before, and data dependency to the existing relation sets inside the new basic block. Second, if $\Theta(tid)$ indicates that there are instructions in the basic block waiting for execution, an instruction is non-deterministically selected for execution (applying function ψ to it with registers ($\Phi(tid)$) and heap snapshot ($\Gamma(tid)$)) if the instruction satisfies the program order relation on the basic block. Third, for a step, `trans` also picks a new time point (the maximum number of the time point set T plus `1`) to add to the set T, and assigns the new time point to a new memory event. The creation of the event is to combine the thread-ID tid, a newly generated action-ID (the action-ID is calculated by combining the dynamic block number with the instruction number), and a memory action calculated from the function ψ (if the instruction is a termination, we assume that the action is τ). Fourth, `trans` also generates a new `rf` pair if the action is a read, and modifies the memory snapshot by inserting the current time point and write value if the action is a write.

3.3 The PLS Proof over Morpheus Optimizations

We utilize the optimizations and the program semantics defined in the last two sections to prove the correctness of a simple code motion optimization (SCM) as a utility of PLS. We want to show that any compiler-optimized (by SCM) program in the language (Fig. 3) per-loc simulates its original unoptimized program.

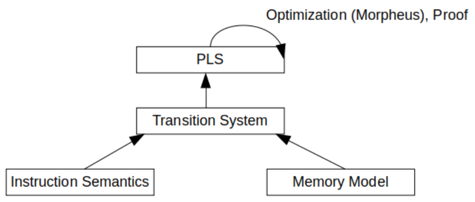


Fig. 9: Optimization Proof with PLS

Fig. 9 provides the structure of the optimization proof. In Sec. 2.1, we described how the PLS framework is parameterized by transition systems. Here we instantiate these systems with the same program transition system in Sec. 3.2. We then instantiate the states (ξ, σ and υ in Fig. 4) as the form (μ, ω) (Fig. 8). For any two states in a LTS ($\mathtt{LTS}_{\Xi}$, $\mathtt{LTS}_{\Sigma}$, or $\mathtt{LTS}_{\Upsilon}$), they have the same program μ. We also map the labels (α, β, and κ) to memory events (Ev). Given a label event ev, the property `val` is implemented as getting the value of the action in ev only if the action is a read or write; if it is a τ event, then the `val` answers $\perp$. `type` is implemented as a read for a read action in the event, as a write for a write action, and as τ for a τ action. `loc` is implemented as getting the memory location in the action of the event (if it is a τ event, then `loc` answers $\perp$). We keep the relation set `eq` the same as the one in Fig. 5a. Assume that a program μ is given, by applying the Morpheus optimization algorithm of SCM, we can rewrite μ as an optimized program μ'. For a fixed initial state ω, the PLS proof is to show that the LTS ($\mathtt{LTS}_{\Xi}$) with the initial state (μ', ω) per-loc simulates the LTS ($\mathtt{LTS}_{\Sigma}$) with the initial state (μ, ω), where there exists a per-loc simulation relation $\sqsubseteq$ for a finite set of locations Loc, such that $(\mu', \omega) \sqsubseteq (\mu, \omega)$. We formalize this result as Theorem 2, and the proof is done in Isabelle. The approach of the proof is first to prove a lemma with a similar structure but for only a single-threaded program with one CFG, and then prove Theorem 2 by using induction on the number of threads in the domain of the program.

Theorem 2. Let $(\mu, \omega)(x)$ ($\xi(x)$ or $\sigma(x)$) be the value of location x at the ω's heap snapshot (Γ in Fig. 8) that belongs to the thread tid such that $\rho(\mathtt{max}(T)) = (tid, aid, ac)$ (ρ and T are the elements in ω in Fig. 8). For any program μ in the language in Fig. 3 with a finite domain (Tid has size n), for any π and any $tid \in Tid$, let $\mu'(tid) \in \mathtt{leftOpt}(\pi)(\mu(tid))$ (or $\mu'(tid) \in \mathtt{rightOpt}(\pi)(\mu(tid))$).

Given a non-empty finite set of memory locations Loc and a given state environment ω, there exists a per-loc simulation $\sqsubseteq$ that satisfies $\mathsf{PLS}^{\mathsf{eq}}_{Loc}(\sqsubseteq)$ and $(\mu', \omega) \sqsubseteq_x (\mu, \omega)$ for all location x, and for all ξ and σ such that $\xi \sqsubseteq_x \sigma$ for a location x, $\xi(x) = \sigma(x)$.

Isabelle Formalization. The PLS framework, the combination of PLS and Morpheus, and the proof of the semantic preservation of a particular optimization on a specific language are achieved through an elegant combination of different **locale** structures [2] in Isabelle. An Isabelle locale structure is a polymorphic theorem structure that is parameterized by a list of Isabelle terms with proper types and a list of assumptions for these terms. Through a locale structure, a collection of theorems can be defined for a list of polymorphic terms, provided that the terms satisfies the assumptions defined for the terms. Users can later instantiate the locale structure to a specific instance of terms by proving the assumptions.

In the Isabelle Morpheus definition, we first define the syntax for the Morpheus specification language. We then define a polymorphic CFG locale structure, named Flow_graph, with all necessary elements in a CFG, such as a set of nodes, a set of edges, the start node and the exit node for the CFG with several assumptions on the CFG well-formedness. The Morpheus specification language semantics is also defined as a locale structure, named Morpheus_sem, which is built on top of the Flow_graph locale. Based on the CFG structure and assumptions provided by flow_graph, Morpheus_sem defines an inductive relation capturing the graph rewriting semantics of Morpheus based on the polymorphic CFG structure (flow_graph).

Before we define PLS in Isabelle, we define an LTS locale for a polymorphic labeled transition system (LTS) with four properties with some well-formedness assumptions. Three of them are listed in Fig 2. The other one is the program text of the LTS described in Sec. 2.3. PLS is defined as a locale, named PLS, with two LTSs and an equation set eq as the input terms. The two LTSs are based on the LTS locale. In the PLS locale, we define a predicate as the one in Fig. 4, which defines the full PLS. When using PLS to prove language properties, one might be more interested in finding a PLS relation. To do that, we combine the Morpheus_sem and the PLS locales, as a new locale Morpheus_com, to build a

PLS relation on top of the Morpheus semantics. In Morpheus_com, we build a new predicate $\texttt{sim}_x^{\texttt{eq}}$ μ μ' $\texttt{steprel}$ n, where μ and μ' are two programs (with the same thread domain), and μ' is the transformed program of μ through a specific optimization defined as an input term of Morpheus_com. steprel is a polymorphic function (defined as a term in a locale) to produce an LTS based on a program by omitting the implementation details of the LTS but only producing the four properties above. It takes in a program μ and a state ω, and outputs a label ev and a new state ω' transitioned from ω. The $\texttt{sim}_x^{\texttt{eq}}$ predicate is valid if and only if the LTS with the program text μ, and an initial state ω (μ, ω), per-loc simulates (with the equation set eq) the LTS with the program text μ, and an initial state ω (μ, ω) in n steps.

As an example (in Isabelle) of defining the optimization in Fig. 7 and proving Theorem 2 on a language in Fig. 3, we first define its instruction and CFG syntax with a definition capturing the instruction level semantics of the language. We also define a memory model as a locale structure capturing the relaxed concurrency behaviors described in Fig. 8 (and the Appendix 6.3). We then define the program semantics as the LTS ($\rightarrow$) in Fig. 8 by instantiating the memory model locale with the language (Fig. 3) and adding more structures (like the program pointer family). Now, we instantiate the Morpheus_sem locale by the CFG syntax in the language, the Steprel locale (for instantiating the function steprel) by the LTS ($\rightarrow$), and the eq set as the one in Fig. 5a, and the compiler optimization term in Morpheus_sem as the one in Fig. 7. The proof of Theorem 2 is then turned to show that the predicate $\texttt{sim}_x^{\texttt{eq}}$ μ μ' $\texttt{steprel}$ n is valid for arbitrary x in a location set *Loc*. We first show the case when $n = 1$ by proving for any one-step transition defined by the LTS ($\rightarrow$) for arbitrary x; then, we lift the proof inductively to arbitrary n step based on the one-step proof result.

4 Related Work

The PLS framework is a combination of three pieces of work: a simulation framework, a compiler-verification framework, and a weak memory model. Simulation/bisimulation were first introduced by Park [38]. Subsequently, much work was published that defined and proved properties about simulations [46,11,6,12,13]. Verifying compilers is one of the top problems in computer science since the work of McCarthy and Painter [35]. A good survey can be found in Dave's work [17]. Recently, one of the most significant achievements in verifying large-language compilers is Leroy's CompCert compiler [7,26]. Chlipala built verified compilers in Coq from λ-calculus to an idealized machine language [15] and from a small functional language to the language [16]. Lochbihler verified a whole-program compiler for multi-threaded Java [30]. Sevcik *et al.* built CompCertTSO [47], which adapted CompCert's correctness proofs to x86TSO to consider the compilation of racy C code. Our domain-specific language for specifying compiler optimizations in Sec. 3 is from Mansky and Gunter's [32,33].

Defining weaker memory models for multi-threaded programs than sequential consistency started with Lamport [25]. Batty *et al.* formalized the C11 model in the axiomatic candidate-execution model format, which was described in depth by Alglave *et al.* [1]. Vafeiadis *et al.* [43] found many other problems in Batty *et al.*'s model and proposed fixes. Lahav *et al.* [22,24] defined a comprehensive C++ model (RC11) based on all previous models, with extra fixes on Batty *et al.*'s model. Many previous papers [39,19,21] also proposed solutions for out-of-thin-air problems, while Podkopaev *et al.* [40] proposed a weaker model than C/C++ concurrency model including these solutions for out-of-thin-air problems. As we mentioned in Sec. 1, the distinguishing of out-of-thin-air behaviors in a program is essentially discovering syntactic similarities between programs that are semantically different. The solutions for out-of-thin-air problems can and should be attained through a well-engineered simulation framework like PLS because merging syntactically similar programs into a semantic memory model building results in incompleteness. Moreover, these models provide correct compilation schemes for a regular and small memory action and language set. It is non-trivial to extend their work to prove a compiler optimization in a large language with a weak concurrency model. Sometimes, extending their work to a little more features is problematic, such as the failing in proving examples about the promising memory model [21] and IMM memory model [40]; not to mention that some of the work has been found to contain problems in their model [40].

The first framework to combine a sequential consistency memory model, compiler proof framework, and bisimulation was CompCert [26]. There are two kinds of simulations in CompCert: the forward and backward simulations, which together describe the simulation relations between a program and its compiled/optimized program. It assumes that any execution of a program is classified into either an execution reaching an error state or a normal execution, and defines simulation relations differently for these two cases. In a normal execution, every element of the execution sequence is either a memory event or a program finishing point. In the framework, an execution sequence is neither split into different sub-traces nor distinguished based on memory location information. Moreover, CompCert's simulation framework does not recognize syntactic dependency on programs just like what PLS does with the `eq` set (Sec. 2.3). CompCertTSO [47] inherited CompCert's bisimulation framework. With minor extensions, they use the framework to prove program semantic preservations in a total store order memory model. Several studies proposed fixes to the bisimulation framework on different topics, such as divergence preservation [29] and creating a program-logic bisimulation framework for the termination-preserving refinement of concurrent programs [28]. All these works enlighten our development of PLS. PLS introduced in this paper is assumed to deal with the normal executions described in CompCert's simulation framework, and the error-state-reaching executions are assumed to be dealt with by the same mechanism as CompCert's framework. The main development and advantage of PLS in proving program semantic preservation properties in weak memory model has been introduced in Sec. 1 and 2.

5 Conclusion and Future Work

In this paper, we propose a new per-location simulation (PLS) relation that is simple and suitable for proving a compiled program preserves its original program semantics under a CFG-based programming language with a real-world, C/C++ like, weak memory model. PLS can be divided into two parts (Sec. 2.1 and 2.3). Based on the small language in Fig. 3, the concurrency model (a subset of weak memory model from C/C++ [22,24]) and program semantics in Sec. 3.2, we have shown the utility of PLS by proving that program semantics is preserved for a simple code motion optimization (defined in Sec. 3.1) for all possible programs. In the future we will use the PLS framework to prove program semantic preservation for complicated compiler optimizations with a complete weak memory model and a large real-world programming language like C/C++/LLVM.

References

1. Alglave, J., Maranget, L., Tautschnig, M.: Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. ACM Trans. Program. Lang. Syst. **36**(2), 7:1–7:74 (Jul 2014). https://doi.org/10.1145/2627752, http://doi.acm.org/10.1145/2627752
2. Ballarin, C.: Tutorial to locales and locale interpretation. Contribuciones científicas en honor de Mirian Andrés Gómez, 2010-01-01, ISBN 978-84-96487-50-5, pags. 123-140 (01 2010)
3. Batty, M., Donaldson, A.F., Wickerson, J.: Overhauling SC Atomics in C11 and OpenCL. SIGPLAN Not. **51**(1), 634–648 (Jan 2016)
4. Batty, M., Memarian, K., Nienhuis, K., Pichon-Pharabod, J., Sewell, P.: The Problem of Programming Language Concurrency Semantics. In: Vitek, J. (ed.) Programming Languages and Systems. pp. 283–307. Springer Berlin Heidelberg, Berlin, Heidelberg (2015)
5. Batty, M., Owens, S., Sarkar, S., Sewell, P., Weber, T.: Mathematizing c++ concurrency. SIGPLAN Not. **46**(1), 55–66 (Jan 2011). https://doi.org/10.1145/1925844.1926394, http://doi.acm.org/10.1145/1925844.1926394
6. van Benthem, J.: Exploring Logical Dynamics. Center for the Study of Language and Information, Stanford, CA, USA (1997)
7. Blazy, S., Leroy, X.: Mechanized Semantics for the Clight Subset of the C Language. Journal of Automated Reasoning **43**(3), 263–288 (2009). https://doi.org/10.1007/s10817-009-9148-3, http://dx.doi.org/10.1007/s10817-009-9148-3
8. Boehm, H.J.: Memory Model Rationales. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2176.html, accessed: 2007-03-09
9. Boehm, H.J., Adve, S.V.: Foundations of the C++ Concurrency Memory Model. SIGPLAN Not. **43**(6), 68–78 (Jun 2008)
10. Boehm, H.J., Demsky, B.: Outlawing Ghosts: Avoiding Out-of-thin-air Results. In: Proceedings of the Workshop on Memory Systems Performance and Correctness. pp. 7:1–7:6. MSPC '14, ACM, New York, NY, USA (2014)
11. Burkart, O., Caucal, D., Steffen, B.: Bisimulation collapse and the process taxonomy. In: Montanari, U., Sassone, V. (eds.) CONCUR '96: Concurrency Theory. pp. 247–262. Springer Berlin Heidelberg, Berlin, Heidelberg (1996)

12. Caucal, D.: On the regular structure of prefix rewriting. In: Arnold, A. (ed.) CAAP '90. pp. 87–102. Springer Berlin Heidelberg, Berlin, Heidelberg (1990)
13. Caucal, D.: On infinite transition graphs having a decidable monadic theory. In: Meyer, F., Monien, B. (eds.) Automata, Languages and Programming. pp. 194–205. Springer Berlin Heidelberg, Berlin, Heidelberg (1996)
14. Chakraborty, S., Vafeiadis, V.: Formalizing the Concurrency Semantics of an LLVM Fragment. In: Proceedings of the 2017 International Symposium on Code Generation and Optimization. pp. 100–110. CGO '17, IEEE Press, Piscataway, NJ, USA (2017), `http://dl.acm.org/citation.cfm?id=3049832.3049844`
15. Chlipala, A.: A Certified Type-preserving Compiler from Lambda Calculus to Assembly Language. SIGPLAN Not. **42**(6), 54–65 (Jun 2007)
16. Chlipala, A.: A Verified Compiler for an Impure Functional Language. SIGPLAN Not. **45**(1), 93–106 (Jan 2010)
17. Dave, M.A.: Compiler Verification: A Bibliography. SIGSOFT Softw. Eng. Notes **28**(6), 2–2 (Nov 2003)
18. Dodds, M., Batty, M., Gotsman, A.: C/C++ Causal Cycles Confound Compositionality. TinyToCS **2** (2013), `http://tinytocs.org/vol2/papers/tinytocs2-dodds.pdf`
19. Jeffrey, A., Riely, J.: On Thin Air Reads Towards an Event Structures Model of Relaxed Memory. In: Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science. pp. 759–767. LICS '16, ACM, New York, NY, USA (2016). https://doi.org/10.1145/2933575.2934536, `http://doi.acm.org/10.1145/2933575.2934536`
20. Kalvala, S., Warburton, R., Lacey, D.: Program transformations using temporal logic side conditions. ACM Trans. Program. Lang. Syst. **31** (05 2009). https://doi.org/10.1145/1516507.1516509
21. Kang, J., Hur, C.K., Lahav, O., Vafeiadis, V., Dreyer, D.: A Promising Semantics for Relaxed-memory Concurrency. SIGPLAN Not. **52**(1), 175–189 (Jan 2017). https://doi.org/10.1145/3093333.3009850, `http://doi.acm.org/10.1145/3093333.3009850`
22. Lahav, O., Giannarakis, N., Vafeiadis, V.: Taming release-acquire consistency. SIGPLAN Not. **51**(1), 649–662 (Jan 2016). https://doi.org/10.1145/2914770.2837643, `http://doi.acm.org/10.1145/2914770.2837643`
23. Lahav, O., Vafeiadis, V.: Owicki-gries reasoning for weak memory models. In: Proceedings, Part II, of the 42Nd International Colloquium on Automata, Languages, and Programming - Volume 9135. pp. 311–323. ICALP 2015, Springer-Verlag, Berlin, Heidelberg (2015)
24. Lahav, O., Vafeiadis, V., Kang, J., Hur, C.K., Dreyer, D.: Repairing sequential consistency in c/c++11. SIGPLAN Not. **52**(6), 618–632 (Jun 2017). https://doi.org/10.1145/3140587.3062352, `http://doi.acm.org/10.1145/3140587.3062352`
25. Lamport, L.: How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. IEEE Trans. Comput. **28**(9), 690–691 (Sep 1979)
26. Leroy, X.: A Formally Verified Compiler Back-end. J. Autom. Reason. **43**(4), 363–446 (Dec 2009). https://doi.org/10.1007/s10817-009-9155-4, `http://dx.doi.org/10.1007/s10817-009-9155-4`
27. Li, L., Gunter, E.: The axiomatic timed relaxed memory model (2019), `https://github.com/liyili2/timed-relaxed-memory-model`
28. Liang, H., Feng, X., Shao, Z.: Compositional Verification of Termination-preserving Refinement of Concurrent Programs. In: Proceedings of the Joint Meeting of

the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS). pp. 65:1–65:10. CSL-LICS '14, ACM, New York, NY, USA (2014). https://doi.org/10.1145/2603088.2603123, `http://doi.acm.org/10.1145/2603088.2603123`

29. Liu, X., Yu, T., Zhang, W.: Analyzing Divergence in Bisimulation Semantics. SIGPLAN Not. **52**(1), 735–747 (Jan 2017). https://doi.org/10.1145/3093333.3009870, `http://doi.acm.org/10.1145/3093333.3009870`
30. Lochbihler, A.: Mechanising a Type-Safe Model of Multithreaded Java with a Verified Compiler. Journal of Automated Reasoning **61**(1), 243–332 (Jun 2018)
31. Mansky, W., Garbuzov, D., Zdancewic, S.: An Axiomatic Specification for Sequential Memory Models. In: Kroening, D., Păsăreanu, C.S. (eds.) Computer Aided Verification. pp. 413–428. Springer International Publishing, Cham (2015)
32. Mansky, W., Gunter, E.: A framework for formal verification of compiler optimizations. In: Kaufmann, M., Paulson, L.C. (eds.) Interactive Theorem Proving. pp. 371–386. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
33. Mansky, W., Gunter, E.L., Griffith, D., Adams, M.D.: Specifying and executing optimizations for generalized control flow graphs. Science of Computer Programming **130**, 2–23 (Nov 2016). https://doi.org/10.1016/j.scico.2016.06.003
34. Manson, J., Pugh, W., Adve, S.V.: The java memory model. SIGPLAN Not. **40**(1), 378–391 (Jan 2005). https://doi.org/10.1145/1047659.1040336, `http://doi.acm.org/10.1145/1047659.1040336`
35. Mccarthy, J., Painter, J.: Correctness of a compiler for arithmetic expressions. pp. 33–41. American Mathematical Society (1967)
36. Meshman, Y., Rinetzky, N., Yahav, E.: Pattern-based Synthesis of Synchronization for the C++ Memory Model. In: Proceedings of the 15th Conference on Formal Methods in Computer-Aided Design. pp. 120–127. FMCAD '15, FMCAD Inc, Austin, TX (2015)
37. Norris, B., Demsky, B.: Cdschecker: Checking concurrent data structures written with c/c++ atomics. SIGPLAN Not. **48**(10), 131–150 (Oct 2013)
38. Park, D.: Concurrency and automata on infinite sequences. In: Deussen, P. (ed.) Theoretical Computer Science. pp. 167–183. Springer Berlin Heidelberg, Berlin, Heidelberg (1981)
39. Pichon-Pharabod, J., Sewell, P.: A Concurrency Semantics for Relaxed Atomics That Permits Optimisation and Avoids Thin-air Executions. SIGPLAN Not. **51**(1), 622–633 (Jan 2016)
40. Podkopaev, A., Lahav, O., Vafeiadis, V.: Bridging the Gap Between Programming Languages and Hardware Weak Memory Models. Proc. ACM Program. Lang. **3**(POPL), 69:1–69:31 (Jan 2019). https://doi.org/10.1145/3290382, `http://doi.acm.org/10.1145/3290382`
41. Vafeiadis, V., Balabonski, T., Chakraborty, S., Morisset, R., Nardelli, F.Z.: Common compiler optimisations are invalid in the C11 memory model and what we can do about it. In: Rajamani, S.K., Walker, D. (eds.) Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015. pp. 209–220. ACM (2015). https://doi.org/10.1145/2676726.2676995, `https://doi.org/10.1145/2676726.2676995`
42. Vafeiadis, V., Balabonski, T., Chakraborty, S., Morisset, R., Zappa Nardelli, F.: Common compiler optimisations are invalid in the c11 memory model and what we can do about it. SIGPLAN Not. **50**(1), 209–220 (Jan

2015). https://doi.org/10.1145/2775051.2676995, http://doi.acm.org/10.1145/2775051.2676995
43. Vafeiadis, V., Balabonski, T., Chakraborty, S., Morisset, R., Zappa Nardelli, F.: Common Compiler Optimisations Are Invalid in the C11 Memory Model and What We Can Do About It. SIGPLAN Not. **50**(1), 209–220 (Jan 2015)
44. Vafeiadis, V., Narayan, C.: Relaxed Separation Logic: A Program Logic for C11 Concurrency. SIGPLAN Not. **48**(10), 867–884 (Oct 2013)
45. Vafeiadis, V., Narayan, C.: Relaxed Separation Logic: A Program Logic for C11 Concurrency. SIGPLAN Not. **48**(10), 867–884 (Oct 2013). https://doi.org/10.1145/2544173.2509532, http://doi.acm.org/10.1145/2544173.2509532
46. Van Benthem, J.: Correspondence Theory, pp. 167–247. Springer Netherlands, Dordrecht (1984), https://doi.org/10.1007/978-94-009-6259-0_4
47. Ševčík, J., Vafeiadis, V., Zappa Nardelli, F., Jagannathan, S., Sewell, P.: CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency. J. ACM **60**(3), 22:1–22:50 (Jun 2013). https://doi.org/10.1145/2487241.2487248, http://doi.acm.org/10.1145/2487241.2487248

6 Appendix

6.1 Morpheus Syntax

Here we introduce the syntax of Morpheus. More details can be found at the work of Mansky *et al.* [33]. The basic approach of the Morpheus specification language is modeled after the TRANS language of Kalvala *et al.* [20]. Optimizations are specified as conditional compositions of rewrites on a generalized control flow graph (GCFG) containing the program's code. The language is partitioned into three largely independent components: core graph transformations (H below), conditions given in a variant of Computation Tree Logic (CTL) (φ below), and a strategy language (T below) for building complex transformations out of component transformations and conditions.

Intuitively, the rewrite portion of an optimization expresses the local transformation to be made, the condition characterizes the situations in which the optimization should be applied, and the strategy language allows us to build whole-system transformations out of collections of local ones. Morpheus is a special-purpose language for the transformation of GCFGs, and as such is parametrized by aspects of GCFGs, namely node names, node labels (program instructions), and edge labels (marking control flow). Transformation specifications may mention aspects of GCFGs concretely, but more generally, they use pattern variables that will be instantiated with control flow graph components in each specific application. We will use the term "expressions" to refer to patterns built from both concrete entities and metavariables (which will be instantiated with concrete entities when the transformation is applied). We use the term metavariable (a) to refer to the variables in the patterns and expressions in Morpheus transformations, as opposed to the concrete programming variables that will be found in instructions.

$$\begin{aligned}
H &\triangleq \texttt{add\_node(}\pi\texttt{,}B\texttt{,(}l_1\texttt{,}\pi_1\texttt{),...,(}l_n\texttt{,}\pi_n\texttt{))} \mid \texttt{add\_node(}\pi\texttt{)} \\
&\quad \mid \texttt{relabel\_node(}\pi\texttt{,}B\texttt{)} \mid \texttt{move\_edge((}\pi\texttt{,}l\texttt{,}\pi_1\texttt{),}\pi_2\texttt{)} \\
\varphi &\triangleq \texttt{true} \mid p(\overrightarrow{x}) \mid \varphi \wedge \varphi \mid \neg\varphi \mid \exists\, a.\, \varphi \mid \mathcal{AX}\ \varphi \mid \mathcal{AY}\ \varphi \mid \mathcal{EX}\ \varphi \mid \mathcal{EY}\ \varphi \\
&\quad \mid \mathcal{A}\ \varphi\ \mathcal{U}\ \varphi \mid \mathcal{E}\ \varphi\ \mathcal{U}\ \varphi \mid \mathcal{A}\ \varphi\ \mathcal{S}\ \varphi \mid \mathcal{E}\ \varphi\ \mathcal{S}\ \varphi \\
T &\triangleq H \mid \texttt{SATISFIED\_AT}\ \pi\ \varphi \mid \texttt{NOT}\ T \mid T \backslash T \mid \texttt{EXISTS}\ a.\ T \mid T\ \texttt{+}\ T \mid T\ \texttt{;}\ T \mid T\ \texttt{*}
\end{aligned}$$

The syntax of Morpheus consists of actions (H), conditions (φ), and transformations (T). The atomic actions H begin with `add_node` and `remove_node`, which add and remove nodes that have no incoming edges. In the case of `add_node`, the addition only takes place if the node description is well-formed as a CFG node (i.e., it has the right number and kind of outgoing edges for its instruction label). The `relabel_node` action relabels an existing node with a new instruction, as long as that new instruction is compatible with the existing edge structure. The only action that operates directly on edges (rather than nodes) is `move_edge`, which moves the destination of an edge from one node in the graph to another. The conditions φ of Morpheus are based on First-Order CTL (FOCTL). Starting from a set of atomic predicates p, they include all of the usual propositional and temporal operators. The $\mathcal{S}$ ("since") and $\mathcal{Y}$ ("yesterday") operators are the past-time counterparts to the $\mathcal{U}$ ("until") and $\mathcal{X}$ ("next") operators respectively; for instance, $\mathcal{E}\ \varphi_1\ \mathcal{S}\ \varphi_2$ holds when there exists some path backwards through the graph such that φ_1 holds until a previous point at which φ_2 holds. The existential quantifier $\exists$ is used to quantify over metavariables in a formula: these metavariables may then appear in the atomic predicates of a formula, enhancing the expressive power of the conditions. At the top level, a transformation T combines conditions and rewrites using strategies. Strategies are inherently non-deterministic, as is reflected by their returning a set of possible transformed graphs. The simplest strategy is just to perform an action H. The strategy `SATISFIED_AT` $\pi\ \varphi$ acts as the identity transformation if φ holds of the GCFG at the node π, and returns the empty set if φ fails to hold on π. Thus `SATISFIED_AT` $\pi\ \varphi$ acts as filter, allowing through only those GCFGs and nodes π that satisfy φ. On the other hand, `NOT` T and $T_1 \backslash T_2$ allow us to deselect graphs by the ability to perform a transformation. The transformation `NOT` T selects those graphs that T cannot transform, i.e., those not in the domain of T, and deselects those that it can transform. The transformation $T_1 \backslash T_2$ restricts the output of T_1 to those graphs that could not be produced by T_2, i.e., those not in the image of T_2. Note the difference between these two filters: `NOT` T filters based on the complement of the domain of a transformation, while $T_1 \backslash T_2$ filters based on the complement of the range of a transformation. The strategy `EXISTS` $a.\ T$ binds a in T, limiting its scope to the free occurrences of a in the conditions and actions of T. Finally, the constructs `+` and `;` allow for choice between and sequencing of two transformations respectively, and the iteration operator `*` allows for the repeated application of a transformation any number of times.

For a simple example of a Morpheus transformation, assume we have a language of instructions that supports assign- ments and binary arithmetic expressions. In this setting, if we have a variable assigned the result of applying an arithmetic operation to two constants, we might want to replace the operation with its result. This can be done by the following transformation:

```
simple_constant_folding(π) ≜ EXISTS x a b c oper.
SATISFIED_AT π stmt( x = oper( a,b)) ∧ is_const(a) ∧ is_const(b) ∧ is_const(c)
∧eval(oper(a,b),c) ; relabel_node(π,x = c)
```

This is an existentially quantified sequence of a condition and an action. (Note that *oper* is a metavariable that will be bound to an arithmetic operator appearing in the program syntax, and `eval(`e`,`c`)` is a predicate asserting that the expression e evaluates to the constant c.) We may apply `simple_constant_folding` to a program with a node labeled `diff = 10 - 2` , and the transformation will match π to (the name of) this node, x to `diff` , a to `10`, b to `2`, *oper* to `( op - )`, and c to `8` (because of the clause is `(`c`,`*oper*`(`a`,`b`))` , and relabel the node to `diff = 8`.

6.2 Memory Model

Here we introduce the memory model described in Sec. 3.2. The model is supposed to be in the format of an axiomatic candidate execution model [1]. By defining a set of binary relations and predicates, a candidate execution model selects a valid set of memory executions from a set of candidate executions. Here, we use a subset of a model from the ATRCM model [27], which has been proved to be sound with respect to the C/C++ memory model defined by Lahav *et al.* [24] and the IMM model [40].

Domain

Time Points: $T \subseteq \mathbb{N}$ Action-IDs: $\mathscr{Aid} \triangleq \mathscr{Name}$ Thread-IDs: $tid \in Tid \subseteq \mathscr{Tid}$

Memory Concurrency Model

Execution Map: $\rho \subseteq T \to (Tid \times \mathscr{Aid} \times \mathscr{Act})$ Data Dependency: $\mathtt{dd} \subseteq T \times T$

Data Dependency Family: $\mathtt{dds} \subseteq Tid \to (T \times T)$ Sequenced-Before Relation: $\mathtt{sb} \subseteq T \times T$

Sequenced-Before Family: $\mathtt{sbs} \subseteq Tid \to (T \times T)$

$\mathtt{acq}(T, \rho, \mathtt{sb}) \triangleq \{(s,t) \in T^2 | (s,t) \in \mathtt{sb} \wedge \mathtt{is\_read}(\rho(s)) \wedge \mathtt{is\_acq}\ (\rho(s))\}$

$\mathtt{rel}(T, \rho, \mathtt{sb}) \triangleq \{(s,t) \in T^2 | (s,t) \in \mathtt{sb} \wedge \mathtt{is\_rel}\ (\rho(t)) \wedge \mathtt{is\_write}(\rho(t))\}$

$\mathtt{po}(T, \rho, \mathtt{sb}, \mathtt{dd}) \equiv \mathtt{dd} \cup \mathtt{acq}(T, \rho, \mathtt{sb}) \cup \mathtt{rel}(T, \rho, \mathtt{sb})$

$\mathtt{pos}(Tid, T, \rho, \mathtt{sbs}, \mathtt{dds}) \equiv \bigcup_{tid \in Tid} \mathtt{po}(T, \rho, \mathtt{sbs}(tid), \mathtt{dds}(tid))$

$\mathtt{co\_cw}(T, \rho, \mathtt{rf}) \triangleq$
$\{(s,t) | (s,t) \in \mathtt{rf} \wedge (\exists (r, r') \in \mathtt{rf}. r < s \wedge t < r' \wedge \mathtt{same\_loc}(\rho(s), \rho(r)) \wedge \mathtt{same\_thread}(\rho(r'), \rho(t)))\}$
$\cup \{(s,t) | \exists r\ r'. (s,r) \in \mathtt{rf} \wedge (t, r') \in \mathtt{rf} \wedge \mathtt{same\_thread}(\rho(s), \rho(t))$
$\wedge \mathtt{same\_thread}(\rho(r), \rho(r')) \wedge ((s < t \wedge r > r') \vee (s > t \wedge r < r'))\}$

$\mathtt{single\_prop}(Tid, T, \rho, \mathtt{sbs}, \mathtt{dds}, \mathtt{rf}) \triangleq \forall (s,t) \in (\mathtt{rf} \cup \mathtt{pos}(Tid, T, \rho, \mathtt{sbs}, \mathtt{dds})). s < t$

$\mathtt{at\_co}(T, \rho, \mathtt{rf}) \triangleq \mathtt{co\_cw}(T, \rho, \mathtt{rf}) = \emptyset$

$\mathtt{sat}(Tid, T, \rho, \mathtt{sbs}, \mathtt{dds}, \mathtt{rf}) \triangleq \mathtt{single\_prop}(Tid, T, \rho, \mathtt{sbs}, \mathtt{dds}, \mathtt{rf}) \wedge \mathtt{at\_co}(T, \rho, \mathtt{rf})$

$\mathtt{mo}_x(T, \rho) \triangleq \{(s,t) \in T^2 | a < b \wedge \mathtt{same\_loc}(\rho(s), \rho(t)) \wedge \mathtt{is\_write}(\rho(s)) \wedge \mathtt{is\_write}(\rho(s))\}$

$\mathtt{mo} \triangleq \bigcup_{x \in Loc} \mathtt{mo}_x$

Fig. 10: Parts of the Definition of the Memory Model

We first show a set of useful conventions. $R^?$, R^+ and R^* represent the reflexive, transitive, and reflexive-transitive closures of relation R. A memory execution (memory trace) in this paper is viewed as a sequence of **memory events** ($\mathscr{Ev}$ in Fig. 8) representing interactions between program executions and the main memory. The structure of a memory event is a triple of a thread-ID (type Tid), a unique memory **action-ID** (type $\mathscr{Aid}$ described in Fig. 8), and a memory action (type l). Each non-τ memory event (`W` and `R`) comes from a memory instruction (store and load instructions) in a program execution produced by the program semantics. A memory execution is defined as a triple of $(T, \rho, \mathtt{rf})$, where T is a downward closed natural number set excluding `0` representing time points, ρ is a partial function from natural numbers to memory events and a reads from relation ($\mathtt{rf}$, type $T \times T$) determining the write-read pairs in the execution. To determine if a memory execution is valid, we need other entities: a set of threads (Tid, type TID set), a family ($\mathtt{sbs}$, type $TID \to T \times T$) of sequenced-before relations $\mathtt{sb}$, each of which is based on the information from a CFG, and a family ($\mathtt{dds}$, type $TID \to T \times T$) of data dependency relations $\mathtt{dd}$. For a CFG G, $\mathtt{sb}$ describes the relations on memory instructions in the CFG that simulate the instruction evaluation order relations from an in-order execution machine to execute G, while dd describes the data, control, and alias dependencies that can be generated by traditional data flow, control flow, and alias analysis algorithms.

In Fig. 10, we describe how we judge if a memory execution is valid. We first define the program order relation po for a thread as the union of all single-threaded instruction dependency relations in the thread. It is a function taking in a time point set T, a ρ function and sb for the thread, and it outputs a relation set defining the instruction dependency. For simplicity, throughout this paper, we use refer to po as the relation set generated by the $\texttt{po}(T, \rho, \texttt{sb}, \texttt{dd})$. The po relation for a set of memory executions represents the exact order of evaluation based on the program meaning. The sequenced-before relation (sb) represents an approximation of the order of evaluation in a memory execution, but it does not mean that an instruction must happen before another instruction only because the programmer writes them in that order. For example, program (a) in Fig. 1 has the read from y sequenced-before the write to x in the left thread, but the program allows the write to happen before the read. In this paper, po is defined as a union of the dd, acq, and rel relations. dd is the data dependence relation for a memory execution including all dependency generated by data flow analysis, control flow analysis, alias analysis and dynamic or static methods that determine the data dependency of instructions in a CFG. acq and rel relations are defined in Fig. 10. Here, is_something is a predicate checks if a memory event has the property described as something. acq defines the binary relations between all acq reads and all memory operations (reads and writes) after it in a thread; rel defines the relations between a rel write in a thread and all operations prior to it. We also define pos as the union of all po in each thread. A valid C/C++ memory model has the property that there is a global total order (modification order) on observations of writes for each location. We define modification order for each location ($\texttt{mo}_x$) in Fig. 10, and mo is the union of all per-location modification orders. To guarantee the valid multi-threaded behaviors, the predicate at_co (Fig. 10) ensures that bad behaviors defined by the co_cw relation do not happen. The co_cw relation collects bad relations in an execution that violates two execution orders: first, every two read instructions in the same thread (by the same_thread function) at the same location (by the same_loc function) read from writes in a timely order. Second, the sends (writes) and receives (reads) between two threads are in FIFO order. Finally, the predicate $\texttt{sat}(Tid, T, \rho, \texttt{sbs}, \texttt{rf})$ checks if an execution is valid by checking an execution satisfy the at_co predicate, as well as checking if all edges in the union of program order relations and the reads-from relation on a program are from an early time to a later time.

We have introduced the memory model briefly. The model is a C/C++ like weak memory model because we have proved that the model is sound with respect to the RC11 model [22]. More specifically, the model is SRA-consistent as the proof below. The details of the proof are in the technical report [27].

Lemma 1. For any valid execution $\texttt{sat}(T, Tid, \rho, \texttt{sbs}, \texttt{dds}, \texttt{rf})$ in ATRCM, and a location set (Loc) given as collecting all locations used in ρ, then $\texttt{pos} \cup \texttt{mo} \cup \texttt{rf}$ is acyclic.

6.3 Program Transition Semantics

Here, we define the semantics for a program based on the program syntax in Fig. 3 and instruction semantics in Sec. 3.2. The program semantics is building an abstract machine executing single threaded instructions through a family of conceptual CPUs (one for each thread), and multi-threaded instructions through a conceptual memory machine. We assume that a CPU execute a basic block of instructions at a time, and any one of executing basic blocks in an program execution, named **dynamic basic block**, can be identified as a unique number $m \in \mathbb{N}$ in a program execution. We use a pair of a unique number and a basic block number of a block $(m, \pi) \in \mathbb{N} \times \mathbb{N}$ to refer to the **dynamic basic block number** (as $\overline{\pi}$) for a dynamic block whose content is the basic block whose node is π in a CFG. Then, a pair of dynamic block number and an instruction number ($\mathbb{N} \times \mathbb{N} \times \mathbb{N}$), named action-ID (as d having type $A = \mathbb{N} \times \mathbb{N} \times \mathbb{N}$), can uniquely identify an executing instruction in a thread in a program execution. In Sec. 6.2, we have described a $\texttt{po}$ relation (or a family $\texttt{pos}$), we extend the idea to a relation $\overline{\texttt{po}}$ (or a family $\overline{\texttt{pos}}$, one for each thread) describing similar program order relations as $\texttt{po}$. $\overline{\texttt{po}}$ for a thread is a relation on $A \times A$, and it describes the program order relations on different instructions instead of memory events in $\texttt{po}$. Every $\overline{\texttt{po}}$ in $\overline{\texttt{pos}}$ is generated along with program semantics transitions.

$$\texttt{form\_D}(\overline{\pi}, \beta) \equiv \{d \mid \exists e\ i.d = (\overline{\pi}, i) \wedge e = \texttt{ins}(\beta, d)\} \quad \texttt{gen}(\overline{\texttt{po}}, \texttt{D}, W) \equiv \{d \in \texttt{D} \mid (\neg \exists d' \in W.(d', d) \in \overline{\texttt{po}})\}$$

$$\text{(a)}\ \frac{\begin{array}{l}(N, \pi_0, \lambda, E) = C \wedge \pi = \texttt{snd}(\overline{\pi}) \wedge \texttt{ins}(\beta, d) = e \wedge \texttt{is\_CInst}(\beta, d) \wedge \lambda(\pi) = \beta \\ \wedge l = \eta(e, \phi) \wedge (\pi, l, \pi') \in E \wedge \lambda(\pi') = \beta' \wedge \overline{\pi}' = (\texttt{fst}(\overline{\pi}) + 1, \pi') \wedge D = \texttt{form\_D}(\overline{\pi}', \beta') \\ \wedge \rho' = \rho[\texttt{max}(T) + 1 \mapsto (tid, d, \tau)] \wedge \overline{\texttt{po}}' = \texttt{gen\_}\overline{\texttt{po}}(G, \overline{\pi}', \overline{\texttt{po}}, \rho', D, \beta', \varphi) \wedge W = \texttt{gen}(\overline{\texttt{po}}, D, D)\end{array}}{\begin{array}{l}\texttt{trans}\big(C, tid, \overline{\pi}, \overline{\texttt{po}}, \texttt{sb}, \texttt{dd}, T, \rho, \varphi, \varGamma, (\{d\}, S)\big) \\ \quad = \big(\overline{\pi}', \overline{\texttt{po}}', \texttt{sb}, \texttt{dd}, T \cup \{\texttt{max}(T) + 1\}, \rho', \varphi, \varGamma, (W, \emptyset), \emptyset\big)\end{array}}$$

$$\text{(b)}\ \frac{\begin{array}{l}(N, \pi_0, \lambda, E) = C \wedge \lambda(\texttt{snd}(\overline{\pi})) = \beta \wedge \texttt{ins}(\beta, d) = e \wedge \neg\texttt{is\_CInst}(\beta, d) \wedge \varGamma(tid) = \gamma \\ \wedge \varGamma(tid') = \gamma' \wedge \gamma'(x) = (t, v) \wedge (\varphi', \texttt{R}^x_{v,o}) = \psi(e, \varphi, \gamma[x \mapsto (t, v)]) \\ \wedge W' = \texttt{gen}(\overline{\texttt{po}}, D - (S \cup \{d\}), W) \wedge \texttt{rf} = \{\texttt{fst}(\gamma'(x)), \texttt{max}(T) + 1)\} \\ \wedge \texttt{sb}' = \texttt{gen\_sb}(\texttt{sb}, \overline{\texttt{po}}, d) \wedge \texttt{dd}' = \texttt{gen\_dd}(\texttt{dd}, \overline{\texttt{po}}, d)\end{array}}{\begin{array}{l}\texttt{trans}\big(C, tid, \overline{\pi}, \overline{\texttt{po}}, \texttt{sb}, \texttt{dd}, T, \rho, \varphi, \varGamma, (W \cup \{d\}, S)\big) \\ \quad = \big(\overline{\pi}, \overline{\texttt{po}}, \texttt{sb}', \texttt{dd}', T \cup \{\texttt{max}(T) + 1\}, \rho[\texttt{max}(T) + 1 \mapsto (tid, d, \texttt{R}^x_{v,o})], \varphi', \varGamma, (W', S \cup \{d\}), \texttt{rf}\big)\end{array}}$$

$$\text{(c)}\ \frac{\begin{array}{l}tid \in Tid \wedge \overline{\texttt{pos}}' = \overline{\texttt{pos}}[tid \mapsto \overline{\texttt{po}}'] \wedge \texttt{sbs}' = \texttt{sbs}[tid \mapsto \texttt{sb}'] \wedge \texttt{dds}' = \texttt{dds}[tid \mapsto \texttt{dd}'] \\ \wedge \overline{\varPi}' = \overline{\varPi}[tid \mapsto \overline{\pi}] \wedge \varPhi' = \varPhi[tid \mapsto \varphi'] \wedge \varTheta' = \varTheta[tid \mapsto \theta'] \wedge \texttt{sat}(Tid, T', \rho, \texttt{sbs}', \texttt{dds}', \texttt{rf} \cup \texttt{rf'}) \\ \wedge \texttt{trans}(\mu(tid), tid, \overline{\varPi}(tid), \overline{\texttt{pos}}(tid), \texttt{sbs}(tid), \texttt{dds}(tid), T, \rho, \varPhi(tid), \varGamma, \varTheta(tid)) \\ \quad = (\overline{\pi}', \overline{\texttt{po}}', \texttt{sb}', \texttt{dd}', T', \rho', \varphi', \varGamma', \theta', \texttt{rf'})\end{array}}{\begin{array}{l}\big(\mu, Tid, \overline{\texttt{pos}}, \texttt{sbs}, \texttt{dds}, \overline{\varPi}, \varPhi, \varTheta, \varGamma, T, \rho, \texttt{rf}\big) \\ \quad \xrightarrow{\rho'(\texttt{max}(T'))} \big(\mu, Tid, \overline{\texttt{pos}}', \texttt{sbs}', \texttt{dds}', \overline{\varPi}', \varPhi', \varTheta', \varGamma', T', \rho', \texttt{rf} \cup \texttt{rf'}\big)\end{array}}$$

Fig. 11: The Program Semantics

The operational transition semantics in Fig. 11 is combination of the instruction level semantics and memory concurrency model. It is represented as a labeled transition system whose states are pairs of programs (μ) and the state environment (ω), and whose labels are memory events. A state environment is a long tuple of a set of thread-IDs (Tid), a program order family ($\texttt{pos}$, one for each thread), a sequenced-before relation family ($\texttt{sbs}$), a data dependency family ($\texttt{dds}$), a current dynamic block number family ($\overline{\Pi}$), a registers family (Φ), a heap snapshot family (Γ) representing different views of the threads of the main memory, a program pointer family (Θ) representing the current executing instruction of each thread, a time point set (T), a ρ mapping, and a reads-from relation ($\texttt{rf}$). We show the rule (c) as the top-most rule of the transition system in Fig. 11. This rule selects a thread tid, applies the one-step transition function $\texttt{trans}$ to the state environment of tid, checks the result of the one-step transition to see if the accumulated result satisfies the predicate of the memory model ($\texttt{sat}$), and then moves forward to a new step via the memory event label $\rho'(\texttt{max}(T'))$. The function $\texttt{max}$ produces the maximum number in T'. We can retrieve the memory event by the $\texttt{max}$ function because the $\texttt{trans}$ function always creates a map entry in ρ from the maximum time point plus $\texttt{1}$ to the current memory event.

We show two rules ((a) and (b)) of $\texttt{trans}$ in Fig. 11. It needs to finish several tasks as a one step evaluation for a thread tid with a CFG C. First, if its program pointer $\Theta(tid)$ points to the end of a basic block (no instructions left for execution), it selects a new basic block according to the edge information in C (applying function η to it with registers ($\Phi(tid)$) to get the edge label), and assigns a new dynamic basic block number with a new program pointer pointing to the top of the new block. In this case, $\texttt{trans}$ also adds new relations of program order, sequenced-before, and data dependency to the existing relation sets inside the new basic block. Second, if $\Theta(tid)$ indicates that there are instructions in the basic block waiting for execution, an instruction is randomly selected for execution (applying function ψ to it with registers ($\Phi(tid)$) and heap snapshot ($\Gamma(tid)$)) if the instruction satisfies the program order relation on the basic block. Third, for a step, $\texttt{trans}$ also picks a new time point (the maximum number of the time point set T plus $\texttt{1}$) to add to the set T, and assigns the new time point to a new memory event. The creation of the event is to combine the thread-ID tid, a newly generated action-ID (the action-ID is calculated by combining the dynamic block number with the instruction number), and a memory action calculated from the function ψ (if the instruction is a termination, we assume that the action is τ). Fourth, $\texttt{trans}$ also generates a new $\texttt{rf}$ pair if the action is a read, and modifies the memory snapshot by inserting the current time point and write value if the action is a write.

Rules (a) and (b) (Fig. 11) are sample rules of the $\texttt{trans}$ function that connects between the transition function ($\rightarrow$) and the single-instruction semantics defined in Sec. 8. $\texttt{trans}$ transitions from a tuple of a CFG C, a thread-ID tid, a current dynamic block number $\overline{\pi}$, a program order $\overline{\texttt{po}}$, a sequenced-before relation $\texttt{sb}$, a data dependency relation $\texttt{dd}$, a time point n, a mapping ρ, a register

ϕ, a family of memory snapshots $\mathit{\Gamma}$, and a program counter θ; to another tuple of a possible new dynamic block number $\overline{\pi}'$, an updated program order $\overline{\texttt{po}}'$, an updated sequenced-before relation $\texttt{sb}'$, an updated data dependency relation $\texttt{dd}'$, a new time point n', a new mapping ρ', a possible new register ϕ', an updated family of memory snapshots $\mathit{\Gamma}'$, an updated program counter θ, and a set of reads-from relations `rf` containing a possible write-read pair generated by a load instruction. In a thread *tid*, `trans` selects one of the possible next instructions in θ to execute. The (a) rule deals with the transition after executing a branching state at the end of a basic block, while rule (b) deals with the case of a load instruction. In these rules, `ins` is a function producing the instruction expression from a basic block and an action-ID. $\gamma|_v$ means to form a new mapping by getting rid of the time point in the value pair $T \times val$. The function $\texttt{gen\_}\overline{\texttt{po}}$ takes the existing $\overline{\texttt{po}}$ and a basic block and generates a new $\overline{\texttt{po}}'$ containing all relations in the `po` relation, all program order relations between instructions in the basic block, and the program order relations between the old-instructions in $\overline{\texttt{po}}$ and the instruction in the new basic block. $\texttt{gen\_}\overline{\texttt{po}}$ happens once when a new dynamic basic block is generated. The function `gen_sb` generates an updated `sb` relation based on the information in the updated $\overline{\texttt{po}}$ relation, while `gen_dd` generates an updated `dd` relation based on the information in the updated $\overline{\texttt{po}}$ relation.