Axiomatic Timed Relaxed Concurrency Model

Author: Please provide author information

Abstract

We introduce a new concurrency model, named the *axiomatic timed relaxed concurrency model* (ATRCM), that combines conceptual time with the relaxed memory model [49], building on top of the axiomatic candidate execution model [3]. ATRCM was designed to be similar to a conceptual computer for executing programs in an imperative language, and for neatly handling the behaviors of both single-threaded out-of-order executions and multi-threaded memory-bus scheduling. We have defined two different scheduling orders: coherence and FIFO. The coherence-ordered version is weaker than the previous C++ axiomatic memory models [23, 21, 5, 42], while the FIFO-ordered one is stronger. Both kinds of scheduling-ordered ATRCM models have been proved sound in the theorem prover Isabelle [40, 37] with respect to the previous models [21, 23, 42], while the coherence-ordered model has been proved complete with them. Furthermore, we corrected several mistakes in previous memory models. The main objective for ATRCM is to be used, as a weak concurrency model, in a large C-like, CFG-based, imperative language to prove that any compiled program semantically preserves its original program's meaning. To accomplish this, we created a *Per-Location Simulation framework* (PLS) to perform the proof for a fixed set of compiler optimizations. We combine ATRCM and PLS with a set of equational rules capturing the syntactic dependency of a fixed set of compiler optimizations to prove that any compiled program semantically preserves its original program's meaning in a large language. To the best of our knowledge, ATRCM with PLS is the first framework enabling the proving of the semantic-preservation property in a large language.

2012 ACM Subject Classification ACM 2012

Keywords and phrases Concurrency Model, Relaxed C/C++/LLVM memory model, Compiler Verification

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2020.

1 Introduction

A memory model describes the semantics of concurrency in a programming language, particularly one with imperative features, delimiting the allowed execution sequences, particularly regarding the values passed between different portions of program executions via variables whose values are stored in the memory. Weak concurrency models for real-world imperative programming languages (C/C++/LLVM/Java) have been studied broadly [3, 7, 5, 17, 46, 38, 8, 4, 36, 22, 21, 23, 41, 19, 20, 12, 42]. Among these studies, the common trend in the formal concurrency-model definitions is to define a concurrency model based on a set of memory operations, or on a very small set of program pieces used to communicate with the memory, then provide correctly verified schemes for compiling from a high-level language to machine code to highlight the key aspects of the compiler transformations.

1.1 Examples of Program Pieces, Memory Executions and Diagrams

Here we provide some example program pieces and other useful graph visual structures that are used in the paper to highlight important aspects of ATRCM. Figure 1 provides some examples of **program pieces**, **execution structures**, **memory executions**, and **execution diagrams**. (a) - (g) in the figure are all program pieces. Some of their syntax differs from the syntax used later in the paper (see definition in Fig. 3), but they are simple syntactic sugars for programs that can be defined as the syntax in Fig. 3. They are just shown here in an easily understandable way. For example, the `do-while` loop in (g) is just

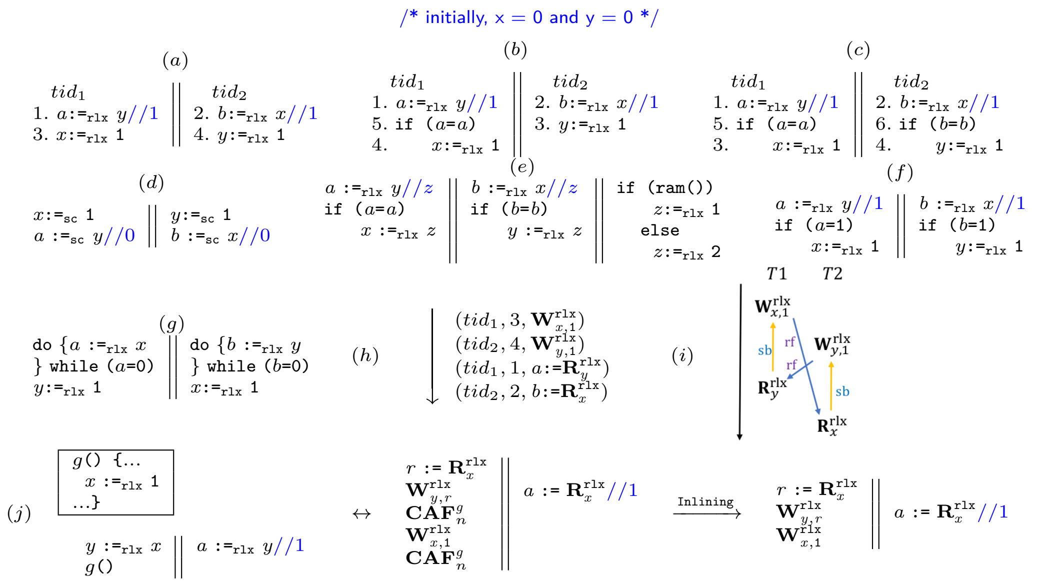


Figure 1 Programs, Executions, Execution Structures, and Diagrams

a syntactic sugar for an if-statement with a loop structure. Throughout this paper, all examples are in this syntactic sugar format. In a program piece and an execution structure, "||" operators that show the parallelism among different threads, and the value after the blue "//" operator in a read instruction puts a restriction on the executions generated by the program piece/execution structure by requiring the value for the read. Sometimes, we put thread-ID names (tid_1 and tid_2) to label threads in a program piece (like (a)) or execution diagram (like (i)), and we label the instructions in a program piece, each of which has a unique number acting as the action-ID of the instance of the executing instruction.

Example (a) in Fig. 1 is a typical example for explaining single-threaded out-of-order executions, since there is no data dependency and memory consistency ordering relations among each thread, and the order of evaluation in each thread in a weak memory model is not fixed. This is why we can observe that the two read atomics both reads 1 in (a). Since its simple structure, it acts as the target program to predict the supposedly allowed behaviors of unoptimized programs. For example, both (b) and (c) can be easily compiled to (a), so their concurrency behavior in C/C++ should be the same as the behaviors of (a). On the other hand, if we observe that the two read atomics in program piece (f) both read 1, or that example (g) terminates, these are not acceptable behaviors because doing compilations from (f) and (g) to (a) does not preserves the program meanings of the two.

An **execution structure** (e.g. the middle and right of (j)) is a syntactic sugar that represents a set of memory executions. It does so by placing memory actions, representing memory events, in a program fashion with "||" operators that show the parallelism among the different threads. The value after the blue "//" operator is the value the read reads in a line, and it essentially puts a restriction on the execution structure by requiring the value for a read in the structure. (j) shows how an execution structure is generated from a program. The left side is a program with a function call g(), The function g contains several instructions with a single write instruction. We generate the middle execution structure by placing a pair of **CAF** fences around the executions of function g, especially around the write event inside g(). The right side is another execution structure achieved after applying an inline expansion optimization to the left program piece.

Execution structures are just a way of presentation. ATMRC actually uses memory executions, either candidate or valid ones, like the ones in (h) (Fig. 1). In an execution, the "↓" represents the flow of time. A memory execution is a sequence of memory events. Often, the linear listing of events in an execution like (h) does not tell the whole story. We then use an ATRCM execution diagram to discuss the relationships among the different events. (i) is an event diagram representing the second execution in (h). An execution diagram also has a time flow (↓), and it lists all threads and their events at the time points when they actually happen. In addition, we also list all restrictions by using pointed edges, with names indicating what kinds of restrictions they are.

1.2 Extending Existing Models

When we try to extend existing models to prove the correctness of a real-world compiler step, especially to prove that a real-world compiler optimization semantically preserves all programs in a language (meaning that all programs optimized in a concurrency model in a language preserve the meaning of their original programs), problems arise. The first problem is confusion in the semantic scope of the language. Historically, the semantics of a real-world imperative language (C/C++/LLVM/Java) have 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 program piece (c) in Fig. 1 (Sec. 1.1), variables a and b can both read 1 if we consider the fact that a simple optimization removes the Boolean guards in (d), making it like program piece (a). Previous researchers either defined their concurrency models without considering optimization effects [5, 21, 12, 23], or they tried to include some optimization effects in their model definitions [41, 19, 20, 42]. The former ignored an important consideration, while the latter were incomplete, just as with incompleteness in first-order logic. There are uncountably infinite (not recursively enumerable) optimization algorithms, but the set of all proofs generated by a computable axiom set from a concurrency model is recursively enumerable. The second problem in proving preservation of meaning is a consequence of the incompleteness, specifically, that directly extending some of the previous models to prove optimizations in a large language results in problematic or error states. The famous out-of-thin-air behaviors are a prime example. The IMM model [42] was designed to include the behaviors of the optimization from (c) to (a) in Fig. 1. However, it accidentally enables both reads in the (f) program to be 1. The promising memory model [20] s able to handle the optimization from (c) to (a), disallowing the behavior of (f), but but it is not able to prove that the two reads in (e) (Fig. 1) can be any value from location z. It also allows program (g) to terminate, a specific out-of-thin-air behavior disallowed in Manson et al.'s work [34].

In this paper, we present the axiomatic time relaxed concurrency model (ATRCM) in the axiomatic candidate-execution model style [3], with a Per-Location Simulation (PLS) framework. This is the first framework to solve the above problems and enable the proof of the semantic preservation of a specific compiler optimization on a large language (Fig. 3) under a weak concurrency model. Here are some features of ATRCM:

Similarity to Real World Computer Concepts. ATRCM splits the model of computation into two pieces. In a memory execution, single-threaded consistency provides guidance for single-threaded out-of-order executions, similar to single-threaded CPU behavior. ATRCM's second piece is the predicate for describing the memory-bus scheduling strategy. In this model, two predicates are provided to describe two kinds of scheduling strategy: FIFO and coherence. FIFO-scheduled ATRCM is the stronger of the two, and we have shown that FIFO ATRCM is sound with respect to previous works [23, 21, 5, 42]. Coherence ATRCM is weaker, but it is still SRA-consistent [21], and also satisfies the SC-Per-Location property [42], which a weak machine-level concurrency model needs to satisfy. The reason to have this separation is that we want to use ATRCM and PLS to prove facts about semantic preservation under compiler optimization based on real-world language semantics, like C/C++/LLVM. Typically, these languages are designed or formalized based on an abstract machine reflecting the real-world computer components. By making ATRCM reflect these designs, we can relate our proofs more precisely to the compiler optimizations with respect to real-world implementations. Details are in Sec. 3.3.

Taking Care of (Out-Of-)Thin-Air Problems. From a compiler verification point of view, thin-air behaviors are those that should not happen when executing an optimized program in in a concurrency setting. For example, the transformation from (c) to (a) (Fig. 1) enables new behaviors, but they are acceptable; but the extra behaviors created by the transformation from (e) to (a) are not acceptable, and are classified as thin-air behaviors. However, if we assume that a conceptual machine executed the language without an optimization, the new behaviors of the transformation from (c) to (a) would not be acceptable, either. Thus, the determination of a thin-air behavior depends on a fixed set of compiler optimizations. Different sets cause different thin-air behaviors. In ATRCM, we developed the PLS framework to prove, for a fixed set of compiler optimizations implemented as a set of equational rules capturing syntactic dependency caused by the optimizations, that optimized programs semantically preserve their original program's meaning. The details are in Sec. 4.

Creating a New Fence for Function Calls. ATRCM is the first work to consider fences on function calls and returns, named call fences. (j) (Fig. 1) provides an example for using call fences. In the left thread in the program piece, there is a function call g`()`, so two call fences (`CAFs`) are generated for the life of the call (as shown in the middle structure). Without the `CAF` fences, the write to x can be executed before the read, so that it is possible for the right thread to read `1`, which is not an acceptable behavior. Call fences enable ATRCM to define and verify a set of optimizations related to calls, such as inline expansion.

Substituting Program Order for Sequenced-Before. For a single-threaded execution in a program, the previous work has viewed the program order relation (`po`) and sequenced-before relation (`sb`) as the same. Here, we split these two concepts. `sb` is basically the program text order and evaluation order when executing a single-threaded program in an in order execution machine. On the other hand, `po` is the union of single-threaded dependencies, including data dependency, control dependency, memory ordering dependency, etc. It represents the true order of evaluation in the programmer's mind. For example, program (a) (Fig. 1) has reads before writes in both threads, but they are labeled with relaxed (`rlx`) ordering, so the programmer did not expect `sb` order in each thread of the program. A lot of the time, these two are the same, but by substituting `po` for `sb`, we are able to complete some previously partial consistencies. For example, SRA-consistency [21] is used to describe the behaviors of `sc` and `acq`/`rel` atomics excluding `rlx` atomics. By substituting `po` for `sb`, we are able to extend SRA-consistency to include the `rlx` atomics behaviors.

Strengthening the Control Dependency Definition in the IMM Model. In the IMM model [42], the control-dependency definition suggests that all local variables and memory reads on some locations in the Boolean guard of a binary operation are dependent on all later writes and uses of the variables and locations. This definition is too weak, since it enables the thin-air behaviors of example (e) – having both reads reading `1`, and allowing example (g) to terminate. To correct this, we strengthen the definition by creating a control fence (`CF`).

A `CF` prevents all write memory events to be executed before or after it. Read events are able to execute before the `CF`, but they need to respect the data dependency in an execution. The IMM model is defined to compile the promising memory model [20], and the weak control dependency is its primary way of handling the compilation. Again, through the PLS framework, with a set of equational rules capturing the optimization's syntactic dependency, we are able to show for a particular program all of its correct (as in the case of IMM) as well as unacceptable behaviors. The details are in Sec. 3.1 and 4.4.

2 Background and Examples

Here we provide an introduction of the key aspects of ATRCM with examples, following the standard axiomatic approach of defining memory consistency models [3]. We distinguish programs, program instruction semantics, program semantics, and memory events in ATRCM. The story of the different entities involved in ATRCM and PLS is in Figure 2.

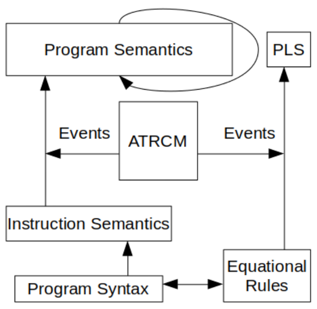


Figure 2 ATRCM and PLS Layouts

We have a set of programming language syntax at the bottom, and the instruction semantics is defined on the syntax. The connection between the whole program semantics and instruction semantics is the concurrency model handled by ATRCM through a set of memory events that are different from the programming language syntax. On the right of Fig. 2, it indicates the strategy of using PLS to prove program semantic preservation based on the program semantics and a set of equational rules capturing the syntactic dependency defined by a fixed of optimizations. PLS works on top of the set of memory events. Here, we introduce different components, but ATRCM and PLS will be introduced in Sec. 3 and 4.

Some useful relations based on a relation (R) in this paper are the reflexive $(R^?)$, transitive (R^+) and reflexive-transitive closures (R^*). $R|_x$ selects all relations in R that access the memory location x, $R|_{\texttt{loc}}$ selects all relations in R both of whose sides access the same memory location, and $R|_{\neq\texttt{loc}}$ is defined as $R \backslash R|_{\texttt{loc}}$. We denote by $R_1; R_2$ the left composition of two relations R_1 and R_2, and assume that ; binds tighter than $\cup$ and $\backslash$. $R|_{\texttt{imm}}$ defines the set of all immediate R edges: $R|_{\texttt{imm}} \equiv R \backslash R; R$, while $[A]$ is the identity relation on a set A. In particular, $[A]; R; [B] \equiv R \cap (A \times B)$. $A \subset_{\texttt{fin}} B$ means that A is a finite subset of B. `dom` is to get the domain of a function while `ran` is to get the co-domain. $\sqcup T$ gets the maximum of T. Some ranging conventions in the paper are in Fig. 3. For example, a, b, and r ranges over the local registers in a program, and x, y, and z ranges over memory locations. We provide a conceptual understanding of **program executions** and **memory executions** here. A program execution is a sequence of states containing executed instructions and their environment states in a conceptual computer, while a memory execution is a sequence of memory events corresponding to states in a program execution that is used by conceptual CPUs in the computer to communicate with the memory machine.

2.1 Programming Language and Memory Event Syntax

Domains

$\pi \in \Pi \triangleq \mathbb{N}$	Basic Block Numbers	$\overline{\pi} \in \mathbb{N}$	Dynamic Block Numbers
$i \in \mathbb{N}$	Instruction Position Numbers	$x, y, z \in \mathsf{Loc} \triangleq \mathbb{N}$	Locations
$n \in \mathbb{Z}$	Integer Values	$v \in \mathsf{Val} \triangleq (\mathsf{Loc}\|\mathbb{Z})$	Values
$a, b, r \in \mathsf{Reg}$	Registers	$tid \in \mathsf{Tid}$	Thread Identifiers
$d \in \mathsf{Aid} \triangleq \mathbb{N} \times \Pi \times \mathbb{N}$	action-ID	$s, t \in \mathsf{Times} \triangleq \mathbb{N}$	Time Points
$g \in \mathsf{Name}$	Function Names	$in \in \mathsf{I} \triangleq (\mathsf{S}\|\mathsf{C})$	Instructions
$ar \in \mathsf{AR} \triangleq \mathbb{N}$	Function Arites		

Orderings

o_r ::= `rlx` | `acq`| `sc`	Read Orderings	o_f ::= `rlx` | `acq`| `acqrel`| `sc`	Fence Orderings
o_w ::= `rlx` | `rel`| `sc`	Write Orderings	o_{rmw} ::= `rlx` | `acq`| `rel`| `acqrel`| `sc`	RMW Orderings

$\sqsupseteq \triangleq \{(\texttt{rlx},\texttt{acq}),\ (\texttt{rlx},\texttt{rel}),\ (\texttt{acq},\ \texttt{acqrel}),\ (\texttt{rel},\texttt{acqrel}),\ (\texttt{acqrel},\ \texttt{sc})\}^*$

Instructions

$\mathsf{Type} \ni ty ::= \texttt{int} \mid ty\,\texttt{*} \quad \mathsf{Exp} \ni e ::= n \mid r \mid e + e \mid e * e \mid e = e \mid e < e \mid e\ op\ e \mid \ldots$

$\mathsf{S} \ni in ::= \texttt{skip} \mid r := ty\ e \mid r := \texttt{phi}\ ty\ ((e, \pi)\ \mathsf{list}) \mid (r|x) := (ty)(r|x) \mid r :=_{o_r} (\texttt{vol})^?\ ty\ x \mid x :=_{o_w} (\texttt{vol})^?\ ty\ e$
$\quad \mid r :=_{o_{rmw}} (\texttt{vol})^?\ ty\ \texttt{fadd}(x, e) \mid \texttt{fence}_{o_f} \mid r := g\ (\texttt{inline})^?\ (e\ \mathsf{list})$

$\mathsf{C} \ni in ::= \texttt{if}\ e\ \texttt{then}\ \pi_1\ \texttt{else}\ \pi_2 \mid \texttt{br}\ \pi \mid \texttt{return}\ e \qquad \mathsf{L} \ni cl ::= \texttt{seq} \mid \texttt{yes} \mid \texttt{no}$

Programs

$N \subset_{\texttt{fin}} \Pi \quad \pi_0 \in N \quad \lambda \in N \rightarrow (\mathsf{S}\ \mathsf{list} \times (\mathsf{C} \cup \{\texttt{Exit}\})) \quad E \subseteq N \times \mathsf{L} \times N \quad \mu \subseteq \mathsf{Tid} \rightarrow \mathsf{CFG}$

$\mathsf{CFG} \ni G ::= (N, \pi_0, \lambda, E)$	Control Flow Graphs (CFG)
$\mathsf{SProg} \ni p ::= (ar, r\ \mathsf{list}, G)$	Functions
$\mho ::= \mathsf{Name} \rightarrow \mathsf{SProg}$	Function Database
$\mathsf{Prog} \ni P ::= (\mathsf{Tid}, \mu)$	Programs

Memory Actions & Events

$\mathsf{A} \ni ac ::= \texttt{ARead}\ \mathsf{bool}\ x\ o_r\ [\texttt{R}^{o_r}_{(x,\ldots)}] \mid \texttt{AWrite}\ \mathsf{bool}\ v\ x\ o_w\ [\texttt{W}^{o_w}_{(x,v,\ldots)}] \mid \texttt{CallFence}\ g\ n\ [\texttt{CAF}^{g}_{n}]$
$\quad \mid \texttt{RMW}\ \mathsf{bool}\ v\ x\ o_{rmw}\ [\texttt{RMW}^{o_{rmw}}_{(x,v,\ldots)}] \mid \texttt{Fence}\ o_f\ [\texttt{F}^{o_f}] \mid \texttt{ControlFence}\ [\texttt{CF}] \mid \tau$

$\mathsf{Event} \ni l ::= (tid, d, ac)$

Figure 3 Domains and Example Language Syntax

ATRCM is independent of a particular programming language. However, a concurrency model is essentially a guide to the concurrency behavior of a specific programming language. ATRCM was created with heavy attention given to the features of a CFG-based imperative language and a C/C++ memory model. For proving the semantic preservation of compiler optimizations, we would like to define a minimal set of instructions in Fig. 3. Expressions (e) are constructed from registers (local variables) and integers, and represent values and locations. Instructions include assignments, branching statements, pointer-integer casting instructions, function calls with a possible `inline` flag, function return statements, and memory instructions. The instruction `if` e `then` π_1 `else` π_2 jumps to the block numbered π_1 if the statement e evaluates as `true`, otherwise, it jumps to block π_2. $r :=_{o_r} (\texttt{vol})^?\ ty\ x$ is a load instruction reading the value from the location x, casting it to ty, and storing it in register r. $x :=_{o_w} (\texttt{vol})^?\ ty\ e$ is an instruction that stores the value of e, whose type is ty, to the location x. $r :=_{o_{rmw}} (\texttt{vol})^?\ ty\ \texttt{fadd}(x, e)$ atomically increments the value in location x by the value of e and loads the old value as type ty in register r. The optional `vol` flag forces the given memory instruction to respect the volatile memory access model, which here refers to the LLVM volatile model [28]. An action-ID (`Aid`) is a triple of a dynamic block number, a basic block number, and an instruction number. It can uniquely identify an executing instruction in a thread of a program execution, so a pair of a thread-ID and action-ID can uniquely identify an executing instruction in a program execution.

A **basic block** is a list of S instructions following by a termination instruction (C or `Exit`). In a basic block, we assign each instruction a **position number** (i), where the sequential instructions are assigned their position in the list (starting from 0), and the terminal instruction (conditional or `Exit`) is assigned the length of the list of sequential instructions. Thus, its position number is one greater than that of the last instruction in the list. In a CFG, the node set N contains the basic block numbers of the CFG $\lambda : N \to (S\ \mathsf{list} \times (C \cup \{\texttt{Exit}\}))$ is a labeling of each node, with a basic block comprising the list of sequential instructions terminated by a branch or `Exit`, and $E \subseteq N \times L \times N$ is a set of edges labeled `seq`, `yes` or `no`, such that, if $\mathsf{snd}(\lambda(n)) = \texttt{br}\ \pi$, then there is a unique out-edge of n, labeled `seq`; if $\mathsf{snd}(\lambda(n)) = \texttt{Exit}$ or $\texttt{return}\ e$, then n has no out-edges; otherwise, there are exactly two out-edges, one labeled `yes` and one labeled `no`. For simplicity, we assume all CFGs in the language (Fig. 2.1) are in the static single assignment format (SSA), and every local variable satisfies the variable dominance property in a CFG. For every variable in a basic block π that has more than one source from two different in-coming blocks of π, there is a `phi` instruction in π to merge the different sources and the `phi` instruction mentions all incoming edges of the block exactly once. A function (p) is defined as a triple of an arity determining the number of arguments, the argument list (registers), and a CFG. A program is defined as a mapping from a set of thread-IDs to CFGs (for simplicity). In this paper, we assume any program to be executed is typed correctly, and the Boolean values `true` and `false` are just syntactic sugars for the integers `1` and `0`. Notice that we have a `skip` instruction, so `if` e $\{e_1\}$ in all examples in this paper just means that `if` e `then` $\{e_1\}$ `else skip`.

We also have a set of **memory events** built on top of a set of **memory actions**. The memory action (l) definition is in Fig. 3. A memory event is either an initial event ($\perp$) or a triple of a thread-ID, action-ID and memory action. Memory events are used by the conceptual CPU when it is executing a memory instruction to communicate with the memory machine. The initial event ($\perp$) represents the initial state of all memory locations. Here, we assume every location has an initial value of `0`. Each executed instruction has a corresponding memory event in the corresponding memory execution mapped from the program execution. Each executed memory, function call, and binary branching instruction has a corresponding non-τ memory event with the same thread-ID and action-ID pair as the memory instruction in the memory execution (a non-atomic memory instruction might have more than one corresponding memory event, but it is excluded in the paper, please refer to this case in the appendix). Other executed instructions have a corresponding τ event. `ARead` (`R`), `AWrite` (`W`) and `RMW` are memory actions corresponding to an atomic load, an atomic write, and a `fadd` instruction, respectively. The bool values in them indicate that the given actions respect the volatile memory access model. The `ControlFence` (`CF`) represents the control dependency from an executed binary branching instruction with respect to all of its executed descendant instructions. `CallFence` (`CAF`) is a fence to force the non-deterministic execution of instructions not crossing function calls. A function call usually generates a pair of `CAF`s, one for the call and the other for its return. We assume the program semantics has a global counter to assign a unique natural number to a function call in a program execution, so a pair of `CAF`s generated as above have the same n number. The brackets of each action in Fig. 3 contain the abbreviation of the action, which is used as a syntactic sugar of a memory event in an execution structure (see Sec. 1.1). The ... in an abbreviation means that we might include additional information for the event containing the action in some examples. For example, execution (e) (Fig. 5) has events with additional sets of time points stating the extra data dependency for the event.

We have briefly described the basic language and memory event syntax. We will introduce the semantics and executions next.

2.2 Instruction Semantics, Axiomatic Program Semantics and Executions

A **basic memory execution** is defined as $(Tid, Loc, T, \rho, \texttt{rf})$, where $Tid \subseteq \mathsf{Tid}$ is a set of thread-IDs, $Loc \subseteq \mathsf{Loc}$ is the set of locations used in the execution, $T \subseteq \mathsf{Times}$ is the set of time points, ρ $(T \to \mathsf{Event})$ is a bijective function from time points to memory events, and $\texttt{rf}$ $(T \times T)$ is a write-read relation set defining the source write event that a read event reads from in the execution. We use bold and teletype font variable names to mean functions to get the target terms in an entities. For example, $\texttt{tid}(l)$ gets the thread-ID of an event, and $\boldsymbol{\overline{\pi}}(l)$ gets the dynamic block number. $\texttt{W}(T, \rho)$ collects all of the write event time points in the execution defined by (T, ρ), and $\texttt{R}(T, \rho)$ collects all of the read event time points. $\texttt{W}_x(T, \rho)/\texttt{R}_x(T, \rho)$ are sets of write/read event time points accessing only location x. The $\texttt{rf}$ above is a subset of $\bigcup_{x \in Loc} \texttt{W}_x(T, \rho) \times \texttt{R}_x(T, \rho)$. For every $(s, t) \in \texttt{rf}$, $\texttt{v}(s) = \texttt{v}(t)$, $\texttt{tid}(s) \neq \texttt{tid}(v)$, $s < t$; and for every $(s_1, t) \in \texttt{rf}$, $s_1 = s$.

A **memory execution** is defined as $\zeta = (Tid, Loc, T, \rho, \texttt{rf}, \texttt{sbs}, \texttt{dds})$, which is a basic memory execution with two additional families of relations $\texttt{sbs}$ and $\texttt{dds}$, one for each thread in Tid. $\texttt{sbs}$ is a family of sequenced-before relations ($\texttt{sb}$ for each thread) in different single-threaded programs, while $\texttt{dds}$ is a family of data dependency relations in different single-threaded programs ($\texttt{dd}$ for each thread), including traditional data dependency, address dependency, output dependency, and pointer aliasing dependency. Both $\texttt{sbs}$ and $\texttt{dds}$ can be generated through the corresponding program execution of the memory execution based on the program instruction semantics. The instruction semantics in Fig. 4 is built in a mechanism that generates the $\texttt{sb}$ and $\texttt{dd}$ for each thread.

$$\texttt{sbs}(T, \rho, \overline{\texttt{sbs}}) \triangleq \texttt{fun}\ tid \to \{(s,t) | \wedge d_1 = \texttt{aid}(\rho(s)) \wedge d_2 = \texttt{aid}(\rho(t)) \wedge (d_1, d_2) \in \overline{\texttt{sbs}}(tid)\}$$
$$\texttt{dds}(T, \rho, \overline{\texttt{dds}}) \triangleq \texttt{fun}\ tid \to \{(s,t) | \wedge d_1 = \texttt{aid}(\rho(s)) \wedge d_2 = \texttt{aid}(\rho(t)) \wedge (d_1, d_2) \in \overline{\texttt{dds}}(tid)\}$$
$$\texttt{bind}(as, es, \varphi) \triangleq \{(a \mapsto v) | \exists i. a = \texttt{nth}(as, i) \wedge v = \texttt{v}(\texttt{ev}(\varphi, \texttt{nth}(es, i)))\}$$

syntax: $\mathsf{S} \cup \mathsf{C}$	semantics: $\psi_{\Omega}^{tid}(g, n, in, d, \varphi, \delta, \gamma, R)$
r := $ty\ e$	$(\varphi[r \leftarrow (ty, \texttt{ev}(\varphi, e))], \delta, \tau)$
r := (int)x	$(\varphi[r \leftarrow (\texttt{int}, x)], \delta, \tau)$
r_x := $(ty$*$)r$	$(\varphi[r_x \leftarrow ((ty*), \texttt{ev}(\varphi, r))], \delta, \tau)$
if e then π_1 else π_2	IF $\texttt{ev}(\varphi, e) = 0$ THEN (no, CF) ELSE (yes, CF)
r := $g(es)$	$\texttt{re}(\mho(g), R) = (ar, as, G) \wedge \lvert as \rvert = \lvert es \rvert$ $\Rightarrow (\texttt{bind}(as, es, \varphi), G, n+1, (g, r, n, d, \varphi)\#\delta, \texttt{CAF}_n^g, R \cup \texttt{def}(ar, as, G))$
return e	let $(g, r, n', (\overline{\pi}', \pi', i'), \varphi')\#\delta' = \delta$ in $(\varphi[r \mapsto \texttt{ev}(\varphi, e)], \delta', \texttt{CAF}_n^g)$
r :=$_{o_r}$ $ty\ x$	$(\varphi[r \leftarrow (ty, \gamma(x))], \delta, \texttt{R}_{\gamma(x), o_r}^x)$
x :=$_{o_w}$ $ty\ r$	$(\varphi, \delta, \texttt{W}_{\texttt{ev}(\varphi, r), o_w}^x)$

Figure 4 Part of the Instruction Level Semantics, and Generation of $\texttt{sbs}$ and $\texttt{dds}$

For a CFG G, $\texttt{sb}$ describes the relations for the memory instructions that simulate the instruction evaluation order relations for an in-order execution machine ($\mathcal{M}$) executing G. What we actually generate through $\mathcal{M}$ on every single-threaded program is actually $\overline{\texttt{sbs}}$ and $\overline{\texttt{dds}}$, which are similar in meanings to $\texttt{sbs}$ and $\texttt{dds}$, and are relations on action-IDs. By giving (T, ρ), $\overline{\texttt{sbs}}$, and $\overline{\texttt{dds}}$, the $\texttt{sbs}$ and $\texttt{dds}$ are defined in Fig. 4. Obviously, for a given program, depending on the values of memory locations each thread observed, different $\texttt{sbs}$ and $\texttt{dds}$ can be generated by $\mathcal{M}$, the memory model is defined based on any one of pairs of $\texttt{sbs}$ and $\texttt{dds}$. In this paper, program semantics is assumed to execute based on a basic block, and it generates a dynamic block number $\overline{\pi}$ every time it takes a new block. The $\overline{\pi}$ values in action-IDs can be viewed as the program counter of the in-order execution machine, and the $\texttt{sb}$ in a thread can be defined as: $(s,t) \in \texttt{sb}(T,\rho) \triangleq \overline{\pi}(\rho(s)) < \overline{\pi}(\rho(t)) \vee (\overline{\pi}(\rho(s)) = \overline{\pi}(\rho(t)) \wedge \texttt{i}(\rho(s)) < \texttt{i}(\rho(t)))$. Thus, the generation of the sb in each thread is just the step-by-step transitions of the program counter in $\mathcal{M}$.

In Fig. 4, we describe a function ψ, which is the sequential instruction level semantics used in $\mathcal{M}$ and program semantics. We use the renaming function $\texttt{re}(p,R)$ to remain all of the register variables, including the argument variables, in a function p to not have any occurrence of variables in R; and ev is a function for evaluating an expression. ψ is parameterized with a thread-ID tid and a function database Ω, and its input has the form $(g,n,in,d,\phi,\delta,\gamma,R)$, where g is the current function name, n is a global counter for distinguishing different pairs of CAF fences, φ : $\mathsf{Reg} \rightarrow \mathsf{Val}$ is the register state, δ is the function stack, $\gamma : Loc \rightarrow (\mathsf{Times} \times \mathsf{Val})$ is a memory snapshot mapping the memory locations to values and the write event action-IDs that represent the latest update of the location in the snapshot, and R is the set for renaming the register variables when calling a new function. Depending on different kinds of instructions, ψ has four kinds of outputs as examples shown in Fig. 4. Other than binary branching instructions, function calls and returns, ψ returns a triple of a new φ, new δ and memory action generated in a single step.

Another function κ is used by $\mathcal{M}$ to generate the $\overline{sbs}$ and $\overline{dds}$ relations for a single thread, as well as a mapping Υ $(Tid \times \texttt{Aid} \rightarrow \texttt{Name} \times \mathbb{N} \times (\texttt{I}))$ from pairs of thread-IDs and action-IDs to instructions with additional information. In generating $\overline{sbs}$ and $\overline{dds}$ relations, we implements the the def-use chain algorithm to compute the data dependency for a CFG in κ. Υ acts as a database for locating a specific instruction (and the function name the instruction resides and the global counter number when the instruction happens) at the point when a pair of a thread-ID and action-ID is generated. It is used in the program semantics definition.

In Def. 1, we define a part of an axiomatic program semantics to generate valid program executions and omit some rules on specificity about particular instructions due to the space limitation. Each state in an execution ξ is (l,σ), where l is the memory event, and σ is the transition state $\sigma = (\Phi,\Delta,\Gamma,T,\rho,\texttt{rf},\Box)$. Φ is the family of registers, Δ is the family of stacks, and Γ is the family of memory snapshots representing the views of memory of each thread. T, ρ,and rf are accumulated components in the corresponding memory execution of ξ. $\Box$ is some additional components we might need when we make the program semantics definition to be operational. $\sigma|_{tid}$ means to create a tuple by getting state information for a single thread tid as $(\Phi(tid),\Delta(tid),\Gamma(tid),T,\rho,\texttt{rf},\Box(tid))$.

▶ **Definition 1.** Given a function database Ω, and a program (Tid,μ), assume that $\overline{\texttt{sbs}}$, $\overline{\texttt{dds}}$, and Υ have been generated through $\mathcal{M}$ on Ω and (Tid,μ), and $\texttt{sat}(Tid,Loc,T,\rho,\texttt{sbs},\texttt{dds},\texttt{rf})$ defines the memory consistency model, which checks if a memory execution is valid. Given an initial state $\sigma_0 = (\emptyset,\emptyset,\emptyset,\emptyset,\emptyset,\Box)$, a transition system $(\nu,\sigma) \Longrightarrow_l \sigma'$ produces a set of program executions Ξ, such that for every **program execution** $\xi \in \Xi$, which is listed as a sequence of $((l_0,\sigma_0),(l_1,\sigma_1),...,(l_n,\sigma_n),...$, and for every adjacent states σ_i and σ_{i+1}, they satisfy that $\exists tid \in Tid\ d.\Upsilon(tid,d) = (g,n,in) \wedge ((g,n,in),\sigma_i) \Longrightarrow_{l_i} \sigma_{i+1} \wedge l_i = (tid,d,\texttt{ac}(\psi^{tid}_{\Omega}(in,\sigma_i|_{tid})))$.

Let each state of ξ to be $(l_i,\sigma_i) = ((tid_i,ac_i),(\Phi_i,\Delta_i,\Gamma_i,T_i,\rho_i,\texttt{rf}_i))$, and let $\Upsilon(tid_i,d_i) = (g,n,in)$ and $((g,n,in),\sigma_i) \Longrightarrow_{l_i} \sigma_j$, such that σ_i and σ_j are adjacent states, and $\sigma_j = (\Phi_j,\Delta_j,\Gamma_j,T_j,\rho_j,\texttt{rf}_j)$. ξ is a **valid program execution**, if any its state σ_i satisfies:

- For every $(tid_i,d_i) \in \texttt{dom}(\Upsilon)$, it appears at most once in the l_i memory event of any state of ξ.
- For the transition from σ_i to σ_j, let $\psi(in,\sigma_i|_{tid}) = (\varphi',\delta',ac)$, then $\Phi_j(tid) = \varphi'$, $\Delta_j(tid) = \delta'$, and if ac is a write or RMW with location x, value v, and $\sqcup(T_j) = t$, then $\Gamma_j(tid)(x) = (d_i,v)$, and another state pieces in σ_i and σ_j are the same.

- For the transition from σ_i to σ_j, $T_j \backslash T_i = \sqcup(T_j)$, $\rho_j(\sqcup(T_j)) = l_i$, $\rho_j \backslash \rho_i = \{\sqcup(T_j) \mapsto l_i\}$, and f l_i is not a read event or RMW event, then $\texttt{rf}_i = \texttt{rf}_j$, and f l_i is a read event or RMW event, then $\texttt{rf}_j \backslash \texttt{rf}_i = \{(t, \sqcup(T_j))\}$, and $\rho(t)$ is a write event whose value is the same as the read/RMW event in l_i, and $\texttt{sat}(Tid, Loc, T_j, \rho_j, \texttt{sbs}, \texttt{dds}, \texttt{rf}_j)$, where $\texttt{sbs} = \texttt{sbs}(T, \rho, \overline{\texttt{sbs}})$ and $\texttt{dds} = \texttt{dds}(T, \rho, \overline{\texttt{dds}})$.
- Other conditions on each different instruction.

For any valid program execution ξ, assume that (l_f, σ_f) is the final state of ξ, and $\sigma_f = (\Phi_f, \Delta_f, \Gamma_f, T_f, \rho_f, \texttt{rf}_f)$, with the given Tid, Loc, $\texttt{sbs}$ and $\texttt{dds}$ components, the corresponding memory execution of ξ is $(Tid, Loc, T_f, \rho_f, \texttt{sbs}, \texttt{dds}, \texttt{rf}_f)$. The Def. 1 is too abstract, for proving any useful compiler optimization semantic preservation property, we need the operational program semantics defined in Sec. 4.2.

3 The Axiomatic Model

In this section, we introduce different components of ATRCM, and then the whole system and theorems. For a program execution ξ, we can generate its corresponding memory execution as a tuple $(Tid, Loc, T, \rho, \texttt{sbs}, \texttt{dds}, \texttt{rf})$, named a candidate memory execution, where we define predicates to check if it is valid. In ATRCM, the process is divided into three parts. First, there are assumptions for the elements of the candidate execution to fulfill as described in Sec. 2.1 and 2.2. This is handled by the `well_formed` predicate here (Fig. 8). Second, for each individual action type feature, in Fig. 3, we define an inductive set (restriction set) parameterized by an execution for that feature, to give **restrictions** about which memory events should follow others in terms of time points. We call these sets **always-predicates**. One of the conditions that a valid execution should satisfy is that the union of all always-predicates (including `rf` but not `sbs`) has no edges from a later time point pointing to an earlier time point. Third, some features cannot be described easily by always-predicates alone. In such cases, we define other predicates, named **never-predicates**, to rule out bad executions. They are satisfied if the behavior they describe never happens. Throughout this section, we use the predicate `is_something` to test if a memory event has an action defined as `something`. For example `is_read` means that an event has a read action (or `RMW`), `is_acq` means that an event is an `acq` atomics, and `is_mem_op` means that an event is a write, read or `RMW`.

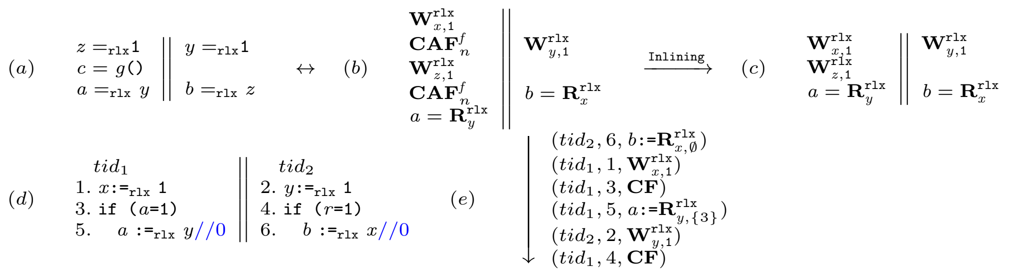
3.1 The `ControlFence` and `CallFence` Actions

Here, we focus on the first set of memory actions that did not commonly appear in the previous axiomatic models. The support for having `ControlFence` and `CallFence` events is found in Sec. 1. The always-predicates for these two fences are in Fig. 5.

In compiling a program to CPU assembly code, a lot of fences will be generated to prevent CPU reordering instructions. In all current compiler implementations (C, C++, LLVM, etc), without specific flags (e.g. enabling function inlining optimization), function calls are always surrounded by fences to prevent later instructions from being executed earlier than the content in the function calls. The `CallFence` event (`CAF`) puts these fences around the function calls. For a functional call, a pair of `CAF`s are inserted at the places of the functional call and the return statement. A `CAF` has two arguments: the function name that it surrounds, and a unique identifier for a pair of `CAF`s that surround the function call. This identifier gives the power to define some aggressive compiler optimizations. For example, the definition of an inline expansion optimization on a function call in a program can be viewed

$$\texttt{control\_dep}(T,\rho,\texttt{sb}) \triangleq \{(s,t) \mid (s,t) \in T^2 \cap \texttt{sb} \wedge ((\texttt{is\_control\_fence}(\rho(s)) \wedge \neg\texttt{is\_read}(\rho(t))) \vee (\texttt{is\_control\_fence}(\rho(t)) \wedge \neg\texttt{is\_read}(\rho(s))))\}$$

$$\texttt{call\_dep}(T,\rho,\texttt{sb}) \triangleq \{(s,t) \mid (s,t) \in T^2 \cap \texttt{sb} \wedge (\texttt{is\_call\_fence}(\rho(s)) \vee \texttt{is\_call\_fence}(\rho(t)))\}$$



```
      tid1                  ||      tid2
1. x:=rlx 1                 || 2. y:=rlx 1
3. if (a=1)                 || 4. if (r=1)
5.    a :=rlx y//0          || 6.    b :=rlx x//0
```

Figure 5 The Properties of Control and Call Fences and Examples

in ATRCM as the removal of a pair of CAFs. One example of generating executions on a program that an optimization has been applied is in Fig. 5 (a), (b) and (c). The left side is a program piece with a function call g in one thread (g is defined in Fig. 1 (j)). The original program generates an execution structure like the one in (b) (Fig. 5), with CAFs surrounding the events inside function g. After we apply inline expansion, we remove the CAFs so that the execution structure becomes (c). In this structure, the write to the z event is free to execute before the write to the x event. This is acceptable in this particular program. Generally speaking, though, inline expansion is not a concurrency safe optimization, meaning that there are times when removing the CAFs damages the program meaning, such as the inline expansion in (j) in Fig. 1. Once the CAFs are removed, the read in the right thread can observe an extra value of 1.

A ControlFence (**CF**) is generated when we have a binary branching instruction at the corresponding time point in the corresponding program execution of a memory execution. It represents the control dependency in the program. We have seen previously in the IMM model that the control-dependency definition is too weak (Sec. 1). Here we make a version that is stronger by not allowing any memory write event to **move across** a CF, meaning that if there is an sb relation on a CF and a write event, then the write event must execute at a time point that does not violate the sb relation. However, a read event can move across a CF provided that there is no data dependency between the CF and the read. Thus, ATRCM allows speculative read executions. For example, the behavior indicated by example (f) in Fig. 1 is not acceptable because we cannot move the two writes across the two CFs even if there is no data dependency between the writes and the CFs. In program piece (d) in Fig. 5, moving the read from x across the CF in thread tid_2 is acceptable, while moving the read from y is not since there is a data dependency between the read and the CF in thread tid_1. (e) is one of the valid execution of (d). In (e), the two additional sets on the two reads are the data dependency the reads need to respect.

3.2 Atomic Memory Operations and Fences

In this section, we discuss the action behaviors of the atomic memory operations (ARead, AWrite and RMW) and memory fences (Fence). As we discussed in Sec. 1, it is best to describe concurrency behavior in terms of single-threaded out-of-order execution behaviors and multi-threaded memory bus scheduling behaviors. Here, we describe a group of always-predicates (`ff_dep`, `vol_dep`, `acqr`, `acqf`, `seqr`, and `seqw`) for describing some single-threaded behaviors and two never predicates (`co_cw` and *co_cw_fifo*) for describing two multi-threaded scheduling behaviors (**FIFO** and **coherence** schedulings). The combination of these behaviors is presented in Sec. 3.3. The division is based on the following observation on memory consistency orderings (o_f, o_r, o_w, and o_{rmw} in Fig. 3): except for `sc` ordering, any previous compiler scheme or machine-level implementation of memory orderings, their behaviors can be described well by the single-threaded, out-of-order execution model. Part of the `sc` atomics can also be described by the model so we can separate the behaviors of `sc` atomics into one part similar to other orderings (such as the `seqr` and `seqw` predicates), and a multi-threaded part (by the `sc_co_cw` never-predicate). There are also multi-threaded effects in the whole system. What we discover is that they can be interpreted by very simple memory bus scheduling properties (FIFO and coherence) based on the single-threaded out-of-order execution model.

$$
\begin{aligned}
&\texttt{ff\_dep}(T,\rho,\texttt{sb}) \triangleq \{(s,t)\in T^2\cap \texttt{sb}\ |\neg\texttt{is\_mem\_op}(\rho(s))\wedge\neg\texttt{is\_mem\_op}(\rho(t))\}\\
&\texttt{vol\_dep}(T,\rho,\texttt{sb}) \triangleq \{(s,t)\in T^2\cap \texttt{sb}\ |\texttt{is\_volatile}(\rho(s))\wedge\texttt{is\_volatile}(\rho(t))\}\\
&\texttt{acqr}(T,\rho,\texttt{sb}) \triangleq \{(s,t)\in T^2\cap \texttt{sb}\ |\texttt{is\_read}(\rho(s))\wedge\texttt{is\_acq}(\rho(s))\wedge\texttt{is\_mem\_op}(\rho(t))\}\\
&\texttt{acqf}(T,\rho,\texttt{sb}) \triangleq \{(s,t)\in T^2\cap \texttt{sb}\ |\texttt{is\_read}(\rho(s))\wedge\texttt{is\_acq}(\rho(t))\wedge\texttt{is\_fence}(\rho(t))\}\\
&\qquad\qquad\qquad\quad \cup\{(s,t)\in T^2\cap \texttt{sb}\ |\texttt{is\_acq}(\rho(s))\wedge\texttt{is\_fence}(\rho(s))\wedge\texttt{is\_mem\_op}(\rho(t))\}\\
&\texttt{seqr}(T,\rho,\texttt{sb}) \triangleq \{(s,t)\in T^2\cap \texttt{sb}\ |\texttt{is\_read}(\rho(s))\wedge\texttt{is\_sc}(\rho(s))\wedge\texttt{is\_mem\_op}(\rho(t))\}\\
&\qquad\qquad\qquad\quad \cup\{(s,t)\in T^2\cap \texttt{sb}\ |\texttt{is\_write}(\rho(s))\wedge\texttt{is\_sc}(\rho(t))\wedge\texttt{is\_read}(\rho(t))\}\\
&\texttt{seqw}(T,\rho,\texttt{sb}) \triangleq \{(s,t)\in T^2\cap \texttt{sb}\ |\texttt{is\_mem\_op}(\rho(s))\wedge\texttt{is\_sc}\ (\rho(t))\wedge\texttt{is\_write}(\rho(t))\}\\
&\qquad\qquad\qquad\quad \cup\{(s,t)\in T^2\cap \texttt{sb}\ |\texttt{is\_write}(\rho(s))\wedge\texttt{is\_sc}(\rho(s))\wedge\texttt{is\_write}(\rho(t))\}
\end{aligned}
$$

$$
\begin{aligned}
&\texttt{sc\_fence\_in\_mid}(T,\rho,s,t) \triangleq (\exists t'.s<t'<t\wedge\{s,t,t'\}\subseteq T\wedge\texttt{is\_sc\_fence}(\rho(t')))\\
&\texttt{write\_in\_mid}(T,\rho,s,t,x) \triangleq (\exists t'.s<t'<t\wedge\{s,t,t'\}\subseteq T\wedge\texttt{is\_write}(\rho(t'))\wedge\texttt{has\_loc}(\rho(t'),x))\\
&\texttt{sr}(T,\rho,\texttt{rf}) \triangleq\\
&\qquad (s,t)\in T^2\wedge s<t\wedge\texttt{is\_write}(\rho(s))\wedge\texttt{is\_sc\_fence}(\rho(t))\wedge x=\texttt{get\_loc}(\rho(s))\\
&\qquad \wedge\neg\texttt{sc\_fence\_in\_mid}(T,\rho,s,t)\wedge\neg\texttt{write\_in\_mid}(T,\rho,s,t,x)\wedge\neg\texttt{sc\_read\_in\_mid}(T,\rho,s,t,x)\\
&\qquad\qquad\qquad \Rightarrow (s,x,t)\in\texttt{sr}(T,\rho,\texttt{rf})\ (*\ \text{scBase}\ *)\\
&\qquad |\ (s,x,t')\in\texttt{sr}(T,\rho,\texttt{rf})\wedge t'<t\wedge t\in T\wedge\texttt{is\_sc\_fence}(\rho(t))\wedge x=\texttt{get\_loc}(\rho(s))\\
&\qquad \wedge\neg\texttt{sc\_fence\_in\_mid}(T,\rho,t',t)\wedge\neg\texttt{write\_in\_mid}(T,\rho,t',t,x)\wedge\neg\texttt{sc\_read\_in\_mid}(T,\rho,t',t,x)\\
&\qquad\qquad\qquad \Rightarrow (s,x,t)\in\texttt{sr}(T,\rho,\texttt{rf})\ (*\ \text{scInduct}\ *)\\
&\qquad \ldots\\
&\texttt{cw}(T,\rho,\texttt{rf}) \triangleq \{(s,x,t)|(s,t)\in\texttt{rf}\wedge x=\texttt{get\_loc}(\rho(s))\}\cup\texttt{sr}(T,\rho,\texttt{rf})\\
&\texttt{sc\_co\_cw}(T,\rho,\texttt{rf}) \triangleq\\
&\quad \{(s,t)|(s,t)\in\texttt{rf}\wedge(\exists t'\in T.\texttt{is\_sc\_write}(\rho(t'))\wedge\texttt{same\_loc}(\rho(s),\rho(t'))\wedge s<t'<t)\}\\
&\quad \cup\{(s,t)|(s,t)\in\texttt{rf}\wedge(\exists t'\in T.\texttt{is\_sc\_fence}(\rho(t'))\wedge s<t'<t\wedge(s,\texttt{get\_loc}(\rho(s)),t')\notin\texttt{cw}(T,\rho,\texttt{rf}))\}\\
&\quad \cup\{(s,t)|(s,t)\in\texttt{rf}\wedge(\exists t'\in T.\texttt{is\_sc\_read}(\rho(t'))\wedge s<t'<t\wedge(s,\texttt{get\_loc}(\rho(s)),t')\notin\texttt{cw}(T,\rho,\texttt{rf}))\}\\
&\quad \cup\{(s,t)|(s,t)\in\texttt{rf}\wedge(\exists t'\in T.s\neq t'\wedge(t',\texttt{get\_loc}(\rho(s)),t)\in\texttt{cw}(T,\rho,\texttt{rf}))\}\\
&\texttt{non\_fifo}(T,\rho,\texttt{rf}) \triangleq \{(s,t)|(s,t)\in\texttt{rf}\wedge\\
&\qquad\qquad (\exists(t_1,t_2)\in\texttt{rf}.\ (s>t_1\wedge t_2>t)\vee(s<t_1\wedge t_2<t))\}\\
&\texttt{non\_at\_co}(T,\rho,\texttt{rf}) \triangleq\\
&\quad \{(s,t)|(s,t)\in\texttt{rf}\wedge(\exists(t_1,t_2)\in\texttt{rf}.\ t_1<s\wedge t<t_2\\
&\qquad\qquad\qquad \wedge\texttt{same\_loc}(\rho(s),\rho(t_1))\wedge\texttt{same\_thread}(\rho(t_2),\rho(t)))\}\qquad (*\ \alpha\ *)\\
&\quad \cup\{(s,t)|\exists t_1\ t_2.\ (s,t_1)\in\texttt{rf}\wedge(t,t_2)\in\texttt{rf}\wedge\texttt{same\_thread}(\rho(s),\rho(t))\\
&\qquad\qquad \wedge\texttt{same\_thread}(\rho(t_1),\rho(t_2))\wedge((s<t\wedge t_1>t_2)\vee(s>t\wedge t_1<t_2))\}\qquad (*\ \beta\ *)\\
&\texttt{co\_cw}(T,\rho,\texttt{rf}) \triangleq \texttt{sc\_co\_cw}(T,\rho,\texttt{rf})\cup\texttt{non\_at\_co}(T,\rho,\texttt{rf})\\
&\texttt{co\_cw\_fifo}(T,\rho,\texttt{rf}) \triangleq \texttt{sc\_co\_cw}(T,\rho,\texttt{rf})\cup\texttt{non\_fifo}(T,\rho,\texttt{rf})
\end{aligned}
$$

Figure 6 Atomic Memory Operations and Fence Behaviors

We first discuss the single-threaded behaviors for all of these actions including the `sc` atomics but not for their multi-threaded effects. To do so, we define always-predicates with three elements provided: a set of time points (T), an `sb` relation, and a function ρ (Sec.2.2). The always-predicates declare restrictions stating when a memory event must follow another one. Figure 6 provides the definitions for the always-predicates for a few cases. The first always-predicate `ff_dep` requires that the relative execution order for different fences is the same as the one given by `sb`, while `vol_dep` restricts the behaviors of the volatile memory operations. As we mentioned in Sec. 2.1, the volatility of a memory action is given by the *bool* value from the flag in its corresponding instruction. Our model adopts the volatility concept from LLVM, i.e. two volatile memory operations cannot move across each other.

In ATRCM, we have defined an always-predicate for each combination of a memory order (excluding `rlx`, which itself has no restrictions) and a memory operation or fence (no `acq` write or `rel` read). Here, we only show the `acqr`, `acqf`, `seqr` and `seqw` sets to put restrictions on `acq` reads, `acq` fences, `sc` reads, and `sc` writes. The `acqr` set prevent any sequenced-after memory operation event of an `acq` read to execute before it in time order. The `acqf` set prevent any sequenced-after memory operation event of an `acq` fence from executing in time order before any read operation before the fence. On top of the `acqr` set, the `seqr` set for an `sc` read additionally requires all writes sequenced-before the `sc` read to not execute after it in time order. The `seqw` set for the `sc` write first requires that the memory operations sequenced-before the write do not move across it (similar to the restrictions for `rel` write), and second requires that writes sequenced-after the `sc` write do not move across it. The other atomics have always-predicates similar to the ones here.

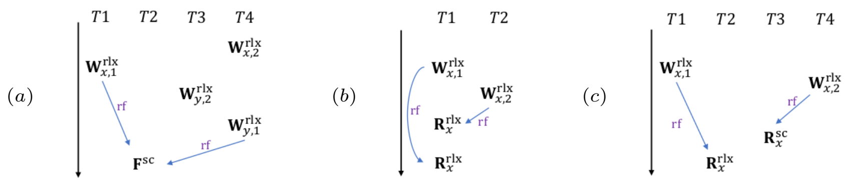


Figure 7 Diagrams for `cw` and `co_cw`

Building on the single-threaded behaviors, we now define the multi-threaded effects of `sc` atomics through the `sc_co_cw` never-predicate. The intuition comes from the memory cache behaviors. We consider that the execution of non-`sc` atomics is only to update the local cache and the time of the value available observation by other threads is unknown; while the `sc` atomics maintain a global consistency of values at a location as soon as they are executed. To achieve this, we first define an inductive relation `sr` to calculate the supposed write-read relations for the `sc` fence/read and its most recent write. We show only two rules for `sc` fences in the definition of `sr` in Fig. 7, and there are other unlisted rules for handle `sc` reads. The execution of an `sc` fence maintains a global consistency of values at all locations, so we view an `sc` fence as a read that observes and unifies all values from different caches for each location in the execution; the value stored at each location is from the latest write, as we show in Fig. 7 (a). In the diagram, we build write-read pairs from the latest writes to x and y with the fence in $T2$. We do this for both `sc` fences and reads (also $RMWs$) in defining the `sr` always-predicate in Fig. 6, which is defined inductively. In these inductive rules, `sc_fence_in_mid` checks if an `sc` fence happens between the two time points in its argument; *write_in_mid* checks if a write to a certain location (x) happens between the two time points in the argument; and `sc_read_in_mid` checks if an `sc` read from a certain location (x) happens between the two time points in the argument. Now, based on the information in `sr`, we can define the never-predicate `sc_co_cw` to collect all write-pairs (s, t) in the given `rf`, such that (1) an `sc` write to the same location happens in the middle of s and t; (2) there is an `sc` fence in the middle of s and t that does not read from the write in s according to `sr`; (3) there is an `sc` read in the middle of s and t that does not read from the write in s according to `sr`; (4) there is another write in `sr` different from the write in s, that

t is reading from.

The multi-threaded scheduling properties are defined by two possible never-predicates: `non_at_co` and `non_fifo`, which ensure two different schedulings: coherence and FIFO. Examples of violating the schedulings are in Fig. 7 (b) and (c), respectively. The coherence scheduling ensures (1) the modification order for each location in each thread, such that if a thread reads two writes from possibly two different threads in order, then the two writes must also execute in time order; as well as (2) single-threaded total order where if a list of writes happens in a thread in time order, then their reads in different threads are also in the time order. The `non_at_co` predicate is for the purpose of collecting the violating pairs of `rf` relations from these two coherence scheduling properties. The FIFO scheduling ensures that for a write-read pair (s, t) in `rf`, there does not exist another write-read pair such that the write executes earlier than s and the read executes later than t. The `non_fifo` predicate collects all pairs of `rf` relations that violate the FIFO scheduling property.

We have stated all of the important single-threaded and multi-threaded memory operation and fence behaviors of ATRCM here. In the following section, we will put these pieces together and prove some theorem to relate ATRCM to previous works.

3.3 Putting it All Together

We have defined predicates to capture the behaviors of individual memory events. Here we merge them together in a single predicate and investigate the equivalence between ATRCM and other models. Mainly, the proofs of equivalence are based on the SRA model [21], RC11 model [23, 21] and the IMM model [42]. To keep uniformity, we use the basic actions and other elemental syntax defined by Batty et al. [5].

We first connect all pieces of ATRCM from the previous sections together in Fig. 8. We define the **program order** of an execution to be all single-threaded restrictions as `po` in the figure. We define **single-threaded consistency** as the predicate `always_prop`. This means for each `po` set for each thread at a memory location, no edge from a later time point pointing to an earlier time point. The definition of **coherence consistency** is that the set `co_cw` is empty. Hence, an **ATRCM consistency** (by the `sat` predicate) is both single-threaded consistent and coherence consistent. Similarly, we define **FIFO consistency** to be a never predicate that makes `co_cw_fifo` empty. Hence, an **ATRCM FIFO consistency** (by the `sat_fifo` predicate) is both single-threaded and FIFO consistent. A small observation on the ATRCM consistent and ATRCM FIFO consistent is that if an execution is ATRCM FIFO consistent, then it is also ATRCM consistent, since if an execution has FIFO scheduling in its multi-threaded behavior, it also has coherence scheduling. Thus, if ATRCM consistency is proved to be sound to a model, then so does ATRCM FIFO consistent.

$$
\begin{aligned}
\texttt{single\_order}(T,\rho,\texttt{sb}) &\triangleq \texttt{acqr}(T,\rho,\texttt{sb}) \cup \texttt{acqf}(T,\rho,\texttt{sb}) \cup \texttt{seqr}(T,\rho,\texttt{sb}) \cup \texttt{seqw}(T,\rho,\texttt{sb}) \cup \ldots \\
\texttt{po}'(T,\rho,\texttt{sb},\texttt{dd}) &\triangleq \texttt{dd} \cup \texttt{control\_dep}(T,\rho,\texttt{sb}) \cup \texttt{call\_dep}(T,\rho,\texttt{sb}) \cup \texttt{single\_order}(T,\rho,\texttt{sb}) \cup \texttt{vol\_dep}(T,\rho,\texttt{sb}) \\
\texttt{po}(Tid,T,\rho,\texttt{sbs},\texttt{dds}) &\triangleq \bigcup_{tid\in\texttt{Tid}} \texttt{po}'(T,\rho,\texttt{sbs}(tid),\texttt{dds}(tid)) \\
\texttt{always\_prop}(Tid,T,\rho,\texttt{sbs},\texttt{dds},\texttt{rf}) &\triangleq \forall(s,t)\in(\texttt{rf}\cup \bigcup_{tid\in\texttt{Tid}} \texttt{po}'(T,\rho,\texttt{sbs}(tid))).s<t \quad \textbf{(single-threaded consistency)} \\
\texttt{sat}(Tid,Loc,T,\rho,\texttt{sbs},\texttt{dds},\texttt{rf}) &\triangleq \texttt{well\_formed}(Tid,Loc,T,\rho,\texttt{sbs},\texttt{dds},\texttt{rf}) \wedge \texttt{always\_prop}(Tid,T,\rho,\texttt{sbs},\texttt{dds},\texttt{rf}) \\
&\quad \wedge \texttt{co\_cw}(T,\rho,\texttt{rf}) \neq \emptyset \quad \textbf{(coherence consistency)} \\
\texttt{sat\_fifo}(Tid,Loc,T,\rho,\texttt{sbs},\texttt{dds},\texttt{rf}) &\triangleq \texttt{well\_formed}(Tid,Loc,T,\rho,\texttt{sbs},\texttt{dds},\texttt{rf}) \wedge \texttt{always\_prop}(Tid,T,\rho,\texttt{sbs},\texttt{dds},\texttt{rf}) \\
&\quad \wedge \texttt{co\_cw\_fifo}(T,\rho,\texttt{rf}) \neq \emptyset \quad \textbf{(FIFO consistency)}
\end{aligned}
$$

Figure 8 ATRCM Consistency and ATRCM FIFO Consistency

Now we have discussed all aspects of ATRCM. We have constructed all of these aspects of ATRCM in Isabelle/HOL, and we also constructed the RC11, SRA and IMM models based on the memory action syntax from Batty **et al.**'s models in Isabelle, and proved the equivalence of our ATRCM model with previous models by proving the soundness between ATRCM and RC11/SRA/IMM and the completeness between ATRCM and IMM (so it includes RC11). The soundness proofs are based on establishing that ATRCM satisfy consistency properties in these models, while the completeness proof is done by constructing a transformation from the memory actions of Batty **et al.**'s model to those of ATRCM, and proving that what is a valid execution in RC11/IMM can also be observed in ATRCM.

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

$$\mathtt{mo} \triangleq \bigcup_{x \in \mathtt{Locs}} \mathtt{mo}_x \quad \mathtt{rs} \triangleq \mathtt{[W]};\mathtt{sb}|^{?}_{\mathtt{loc}};\mathtt{[W^{\sqsupseteq rlx}]};(\mathtt{rf};\mathtt{[RMW]})^{*} \quad \mathtt{sw} \triangleq \mathtt{[Q^{\sqsupseteq rel}]};(\mathtt{[F]};\mathtt{sb})^{?};\mathtt{rs};\mathtt{rf};\mathtt{[R^{\sqsupseteq rlx}]};(\mathtt{sb};\mathtt{[F]})^{?};\mathtt{[Q^{\sqsupseteq acq}]}$$

$$\mathtt{rb} \triangleq \mathtt{rf}^{-1};\mathtt{mo} \quad \mathtt{eco} \triangleq (\mathtt{rf} \cup \mathtt{mo} \cup \mathtt{rb})^{+} \quad \mathtt{hb} \triangleq (\mathtt{po} \cup \mathtt{sw})^{+}$$

$$\mathtt{scb} \triangleq \mathtt{sb} \cup \mathtt{sb}|_{\neq\mathtt{loc}};\mathtt{hb};\mathtt{sb}|_{\neq\mathtt{loc}} \cup \mathtt{hb}|_{\mathtt{loc}} \cup \mathtt{mo} \cup \mathtt{rb} \quad \mathtt{psc}_{\mathtt{base}} \triangleq (\mathtt{[E]}^{\mathtt{sc}} \cup \mathtt{[F]}^{\mathtt{sc}};\mathtt{hb});\mathtt{scb};(\mathtt{[E]}^{\mathtt{sc}} \cup \mathtt{hb};\mathtt{[F]}^{\mathtt{sc}})$$

$$\mathtt{psc}_{\mathtt{F}} \triangleq \mathtt{[F]}^{\mathtt{sc}};(\mathtt{hb} \cup \mathtt{hb};\mathtt{eco};\mathtt{hb});\mathtt{[F]}^{\mathtt{sc}} \quad \mathtt{psc} \triangleq \mathtt{psc}_{\mathtt{base}} \cup \mathtt{psc}_{\mathtt{F}} \quad \mathtt{detour} \triangleq ((\mathtt{co} \setminus \mathtt{sb});(\mathtt{rf} \setminus \mathtt{sb})) \cap \mathtt{po}$$

Figure 9 Parts of the Relations in Batty's Model and SRA/RC11/IMM

We first define the modification order at a specific location x as $\mathtt{mo}_x$ in Figure 9, which respects the modification order definitions in RC11/SRA/IMM. Based on `mo`, the first lemma we want to prove is that ATRCM satisfies the modification order printed in the C++11 memory model documentation [15]. We show only the one under the coherence scheduling. The FIFO scheduled executions satisfy this lemma trivially. The proof of lemma 2 is a direct consequence of the requirement for coherence order in the memory machine defined by the α part of the `cross_co_cw` never-predicate, and it has been formalized and proved in Isabelle.

▶ **Lemma 2.** For any valid execution $\mathtt{sat}(Tid, Loc, T, \rho, \mathtt{sbs}, \mathtt{dds}, \mathtt{rf})$ in ATRCM, for any location $x \in Loc$, $\{s,t,s_1,t_1\} \subseteq T$, $(s,t) \in \mathtt{rf}|_x$, $(s_1,t_1) \in \mathtt{rf}|_x$, t and t_1 having events from the same thread, $t < t_1$, then $s = s_1$ or $(s,t_1) \in \mathtt{mo}_x(T,\rho)$.

The second theorem builds the relationship between ATRCM and SRA. Essentially, the SRA model defines the relationships of the `acq` and `rel` atomics. To show the soundness of ATRCM with respect to SRA, we prove that ATRCM satisfies the main property of SRA (the SRA-coherence property in [21]). Before getting into the details of the proof, we first introduce several always-predicates/relations, mostly from SRA/RC11/IMM, then we use them in our proofs in Fig. 9. Except for $\mathtt{mo}_x$, we do not list the arguments for each set, for simplicity. To properly use these relations, proper arguments for each one are necessary. In RC11/SRA/IMM, the relations are through memory events, since they do not have the concept of time points. Here, the sets are built through time points mapped to memory events. In these relation definitions, `[X]` means to create an identity relation over the set `X`, and the ; operator is the left function composition. In an execution, `W` refers to the set of all writes, `R` to the set of all reads, `RMW` to the set of all read-modify-writes, `F` means the set of all memory fences, and `Q` is the sum of the above. $\sqsupseteq O$ means that the atomics have at least the order of O (order is defined in Fig. 3).

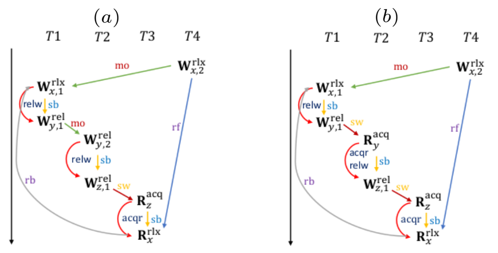
We introduce a key item that distinguishes ATRCM from the previous axiomatic models – the definition of a program order relation (`po`). In the previous models [23, 21, 5, 42], this was a critical matter in different theorems. However, they either did not specify what a program order [5, 21] exactly was, or claimed that `po` was just the `sb` relation [3, 23, 42].

Since ATRCM is used as a model for proving compiler correctness, we cannot just leave the program order without an answer. The latter definition is too restrictive and rules out too many good cases. For example, the (a) program piece (Fig. 1) can never happen according to RC11's NO-THIN-AIR property because the definition uses `sb` as its key part. The essential usage of `po` is to take into account all kinds of program dependencies. In a traditional memory model setting, the memory actions are those reads and writes that directly interact with the memory, so other than assuming that program instructions relate to each other line by line (`sb`), there is no good way to enforce the program dependencies. In ATRCM, since the memory actions lifted from many program dependencies are designed explicitly, we are able to compute the actual program dependencies by combining different kinds of dependencies together, like the `po` definition in Figure 8.

With the `po` relation in mind, we prove the SRA-coherence property [21] for ATRCM consistent executions, listed in Theorems 3 and 4. The first version (Theorem 3) uses the `sb` relation, as it is part of the precise SRA-coherence property listed in SRA. This version only works if there are neither non-atomics nor `rlx` atomics in an execution. (i) in Fig. 1 is the diagram of (a) and is an example that does not satisfy SRA-coherence (having the cycle $\mathbf{W}_y \rightarrow \mathbf{R}_y \rightarrow \mathbf{W}_x \rightarrow \mathbf{R}_x \rightarrow \mathbf{W}_y$) but current compilers like GCC and LLVM allow this behavior. The second version (Theorem 3) uses `po` in ATRCM to substitute for `sb`, and we do not need to rule out non-atomics or `rlx` atomics in this case, because some `sb` edges do not occur in `po` (e.g. the `sb` edges in (d)). With a precise definition of program dependencies instead of the blurry `sb` relation in SRA, SRA-coherence can be used to describe the behaviors of (a) in Fig. 1, and (a) can be distinguished from (c) if the two reads in (c) both read value `1` (not-allowed in SRA). This is because there are edges between the reads and the writes in (c) in its `po`, but there are no such edges in (a). In late proofs and some definitions (like `hb`) in Fig. 9, we use `po` in ATRCM instead of sequenced-before relations.

▶ **Theorem 3.** For any valid execution $\texttt{sat}(Tid, Loc, T, \rho, \texttt{sbs}, \texttt{dds}, \texttt{rf})$ in ATRCM, and every memory operation action in ρ has at least `acq` and `rel` order, then $\bigcup_{tid \in \texttt{Tid}} \texttt{sbs}(tid) \cup \texttt{mo} \cup \texttt{rf}$ is acyclic.

▶ **Theorem 4.** For any valid execution $\texttt{sat}(Tid, Loc, T, \rho, \texttt{sbs}, \texttt{dds}, \texttt{rf})$ in ATRCM, then $\texttt{po}(Tid, T, \rho, \texttt{sbs}, \texttt{dds}) \cup \texttt{mo} \cup \texttt{rf}$ is acyclic.



(c)

tid_1	tid_2	tid_3
$\mathbf{W}^{\texttt{rlx}}_{x,1}$	$a = \mathbf{R}^{\texttt{acq}}_{z}$ //1	$\mathbf{W}^{\texttt{rlx}}_{y,1}$
$\mathbf{W}^{\texttt{rel}}_{z,1}$	$\mathbf{F}^{\texttt{sc}}$	$\mathbf{F}^{\texttt{sc}}$
	$b = \mathbf{R}^{\texttt{rlx}}_{y}$ //0	$c = \mathbf{R}^{\texttt{rlx}}_{x}$ //0

Figure 10 ATRCM Example Diagrams

The next set of theorem proofs shows that ATRCM satisfies the properties of RC11. There are four main ones for RC11-consistency: COHERENCE, ATOMICITY, SC and NO-THIN-AIR. In these properties, we do not need to show ATOMICITY, because the `RMW` atomics in ATRCM are assumed to be atomic. ATRCM consistent executions do not satisfy the COHERENCE property in RC11/IMM, because they do not allow the behavior in (b) of Fig. 10. the executions in Fig. 10 (a) and (b) are borderline examples, meaning that they are not specifically disallowed in the hardware memory models (ARM/POWER models [43, 30]), but no current compilers have enabled them. Allowing or disallowing them together is not a problem. The problem is that RC11 allows (a) and disallows the other on the basis of the difference between a synchronized-with (`sw`) relation and a modification order relation (`mo`). An `sw` edge creates a happens-before relation between two events, but a `mo` edge just indicates the order of two writes for the same location. However, this is not the case here. There is an additional `rel` order on the write to y in $T1$ (a). A `rel` ordered write ensures that any other write does not go across it. In this case, the `rel` write ensures that the write to x must happen before that to y, which creates a relation between the `mo` edges of x and y. According to the compilation scheme from their models [23, 42] to POWER, cases (a) and (b) both generate light weight fences between the two elements in $T1$, $T2$ and $T3$. Essentially, from their own perspective, the two executions have no difference in terms of constraints when they are translated to POWER; so there is no reason for a model to accept one and disallow the other. In ATRCM, the FIFO ordered model disallows both, while the coherence ordered model allows them.

COHERENCE in RC11 is defined as $\texttt{hb};\texttt{eco}^?$ (`eco` and `hb` in Fig.9). The extended-coherence order (`eco`) is the transitive closure of the set of `rf`, `mo` and reads-before (`rb` in Fig.9) restrictions, while the happens-before (`hb`) is the transitive closure of the `sb` and synchronized-with (`sw`) sets. In (e), even if we substitute `po` for `sb`, there is still a reflexive edge between the write to x at thread $T1$ and the read from x in thread $T3$. This shows that ATRCM executions with coherence scheduling are weaker than RC11/IMM ones. We proved that ATRCM ensures that programs with only one shared location are sequentially consistent (the SC-per-location property from IMM), as required in machine level concurrency models [43, 30]. We have shown that in Isabelle the ATRCM consistent executions satisfy SC-per-location in Theorem 5, and that ATRCM FIFO consistent executions satisfy the RC11/IMM COHERENCE property in Theorem 6.

▶ **Theorem 5.** (SC-Per-Location for ATRCM Consistency). For a valid execution $\texttt{sat}(\mathit{Tid}, \mathit{Loc}, T, \rho, \texttt{sbs}, \texttt{dds}, \texttt{rf})$ in ATRCM, $(\texttt{po}(\mathit{Tid}, T, \rho, \texttt{sbs}, \texttt{dds}))|_{\texttt{loc}} \cup \texttt{rf} \cup \texttt{mo} \cup \texttt{rb}$ is acyclic.

▶ **Theorem 6.** (RC11 COHERENCE property for ATRCM FIFO Consistency). For a valid execution $\texttt{sat\_fifo}(\mathit{Tid}, \mathit{Loc}, T, \rho, \texttt{sbs}, \texttt{dds}, \texttt{rf})$ in ATRCM, then $\texttt{hb};\texttt{eco}^?$ is irreflexive.

The SC consistency property in RC11/IMM ensures the `sc` atomics perform properly in a valid execution. RC11/IMM defines a complicated relation `psc` to be acyclic, and the definition of `psc` is in Fig. 9, which restricts the `sc` atomics and fence behavior. One counterexample is the (c) in Fig. 10 where the two reads after the `sc` fence in threads tid_2 and tid_3 must execute before each other in time, a execution time conflict occurs. The last property is NO-THIN-AIR in RC11/IMM, which prevents out-of-thin-air behaviors. In RC11, the property is defined by the predicate: $\texttt{sb} \cup \texttt{rf}$ is acyclic, so RC11 actually disallows the behavior in Fig. 1 (a). This program is allowed in a lot of hardware and compiler implementations. IMM makes a weaker version by replacing $\texttt{sb} \cup \texttt{rf}$ with a relation, named `ar`, defining the unions of all program restrictions. As we mentioned in Sec. 1, the control dependency definition in IMM is too weak, so by substituting ATRCM `CF` dependency for their control dependency, the `ar` is equal to $\texttt{psc} \cup \texttt{po} \cup \texttt{rf} \cup \texttt{detour}$. The acyclicity of the `detour` restriction set is saying that a thread (tid_1) cannot read from a value that will be sent by a write in a later event in tid_1 to another thread (tid_2) that sends the read value of tid_1. This property is guaranteed by the coherence scheduling and the assumption of no write-read pairs in `rf` are from a later write to a early read in time. Hence, the acyclicity of `ar` is approximately the combination of the SC consistency and NO-THIN-AIR property if we substitute `po` for `sb`. So we have formalized and proved the Theorem 7 to capture the acyclicity of `ar` as the soundness proof of ATRCM consistency with respect to the RC11 SC consistency, RC11 NO-THIN-AIR property and IMM NO-THIN-AIR property.

▶ **Theorem 7.** Any valid execution $\texttt{sat}(Tid, Loc, T, \rho, \texttt{sbs}, \texttt{dds}, \texttt{rf})$ in ATRCM satisfies that $\texttt{psc} \cup (\texttt{po}(Tid, T, \rho, \texttt{sbs}, \texttt{dds})) \cup \texttt{rf} \cup \texttt{detour}$ is acyclic.

The final theorem in this section is the relative completeness theorem of ATRCM with respect to IMM. It is relative because we need some modifications to IMM to be successful. The first modification is to use the memory events in Batty et al.'s model [5], because IMM splits an `RMW` action into two different atomics. Since we do not support this feature, we use Batty et al.'s set of events. Second, a candidate execution in IMM based on Batty et al.'s event set syntax is a tuple as ($\texttt{Acts}_r$, Tid_r, Loc_r, $\texttt{sbs}_r$, $\texttt{dds}_r$, $\texttt{rf}_r$, $\texttt{mo}_r$, $\texttt{sc}_r$). Tid_r, Loc_r, $\texttt{sbs}_r$, $\texttt{dds}_r$, $\texttt{rf}_r$ and $\texttt{mo}_r$ are similar to the entities without subscript r. $\texttt{Acts}_r$ is a set of memory events in IMM, and $\texttt{sc}_r$ is a relation defining the `sc` atomics with respect to other events happening in an execution. Since RC11/SRA/IMM does not have the concept of time points, its candidate executions need these relations to describe the execution behaviors. The problem is that the definition of $\texttt{mo}_r$, $\texttt{sc}_r$ and $\texttt{dds}_r$ can be absolutely anything. Even though the implementation of RC11 has well-formed checks, it is not enough. For example, there is no rule to prevent $\texttt{mo}_r$ from being defined as an empty set in a candidate execution in RC11, while the execution can still be RC11-consistent. To prevent this, we require all events in $\texttt{Acts}_r$ to appear once in $\texttt{sbs}_r$, all write events in $\texttt{Acts}_r$ to appear once in $\texttt{mo}_r$, all `seq` events to appear once in $\texttt{sc}_r$, and the translation of $\texttt{dds}_r$ in ATRCM is the `dds` relation generated from our program execution model in Sec. 2.2. We describe the above properties as well-formedness in Theorem 8. The `trans` function translates an IMM execution to a set of ATRCM ones. For any execution in IMM, `trans` generates a set of executions in ATRCM, since IMM is descriptive and every valid execution defined in IMM describes a group of valid executions. It is hard to make an execution in IMM strictly unique. We show Theorem 8 below, as we have proved the theorem.

▶ **Theorem 8.** For a valid and well-formed execution ($\texttt{Acts}_r$, Tid_r, Loc_r, $\texttt{sbs}_r$, $\texttt{dds}_r$, $\texttt{rf}_r$, $\texttt{mo}_r$, $\texttt{sc}_r$) that is IMM-consistent and well-formed, and (Tid, Loc, T, ρ, `sbs`, `dds`, `rf`) $\in$ $\texttt{trans}$($\texttt{Acts}_r$, Tid_r, Loc_r, $\texttt{sbs}_r$, $\texttt{dds}_r$, $\texttt{rf}_r$, $\texttt{mo}_r$, $\texttt{sc}_r$), then $\texttt{sat}(Tid, Loc, T, \rho, \texttt{sbs}, \texttt{dds}, \texttt{rf})$.

4 Usage of ATRCM and PLS

We show a usage of ATRCM by plugging it into the PLS framework with the optimization domain specific language from the Morpheus framework [33]. Mainly, we want to show a fixed optimization preserves program semantic meaning for all programs in a language (Fig. 3). A PLS relation is given as $\texttt{PLS}^{\texttt{eq}}(\sigma', \sigma)$ where σ' and σ are two states, `eq` is a set of equational rules capturing the syntactic dependency of a fixed set of optimizations on the language (Fig. 3). The general strategy of using PLS to prove compiler optimization semantic preservation is given in Fig. 2. To describe the PLS definition, we first need to what are the set `eq` and the operational program semantics that PLS is based on.

$$\begin{array}{ll}
\texttt{cut}(B,i) \equiv (B_1, in, B_2)\ \texttt{IF}\ B_1@[in]@B_2 = B \wedge in = \texttt{nth}(B,i) & \texttt{get\_B}(\lambda,\pi) \equiv \texttt{fst}(\lambda(\pi))@[\texttt{snd}(\lambda(\pi))] \\
\texttt{insert\_i}(B,i,in) \equiv B_1@[in]@B_2\ \texttt{IF}\ B_1@[in']@B_2 = B \wedge in' = \texttt{nth}(B,i) & \texttt{up}(B) \equiv (B\backslash\texttt{last}(B), \texttt{last}(B)) \\
\texttt{cone}(E,\pi,\pi') \equiv \{(\pi_1, cl, \pi_2) \in E | \pi_1 \neq \pi'\} \cup \{(\pi, cl, \pi_1) | (\pi', cl, \pi_1) \in E\} &
\end{array}$$

$$\begin{array}{l}
\texttt{eq} \triangleq \{(\ \equiv_e\ ) \\
\quad n = \varphi(r) \Rightarrow (r,\varphi) \equiv_e (n,\varphi),\ (n_1 \texttt{ + } n_2, \varphi) \equiv_e (n_1 \texttt{sem(+)} n_2, \varphi),\ (n_1 \texttt{ * } n_2, \varphi) \equiv_e (n_1 \texttt{sem(*)} n_2, \varphi), \\
\quad (n \texttt{ = } n, \varphi) \equiv_e (\texttt{true}, \varphi),\ n_1 \neq n_2 \Rightarrow (n_1 \texttt{ = } n_2, \varphi) \equiv_e (\texttt{false}, \varphi),\ n_1 < n_2 \Rightarrow (n_1 \texttt{ < } n_2, \varphi) \equiv_e (\texttt{true}, \varphi), \\
\quad n_1 \geq n_2 \Rightarrow (n_1 \texttt{ < } n_2, \varphi) \equiv_e (\texttt{false}, \varphi),\ (n_1\, op\ n_2, \varphi) \equiv_e (n_1 \texttt{sem}(op)\ n_2, \varphi), \\
\quad (e,\varphi) \equiv_e (e',\varphi) \Rightarrow (es; (e,\pi); es', \varphi) \equiv_e (es; (e',\pi); es', \varphi),\ \ldots \\
(\ \longrightarrow_i\ ) \\
\quad (e,\varphi) \equiv_e (e',\varphi) \Rightarrow (r \texttt{ := } e, \pi, \lambda, E, \varphi) \longrightarrow_i (r \texttt{ := } e', \pi, \lambda, E, \varphi), \\
\quad (e,\varphi) \equiv_e (n,\varphi) \Rightarrow (r \texttt{ := } e, \pi, \lambda, E, \varphi) \longrightarrow_i (r \texttt{ := } n, \pi, \lambda, E, \varphi[r \mapsto n]), \\
\quad (e,\varphi) \equiv_e (e',\varphi) \Rightarrow (x \texttt{ :=}_{ow}\ ty\ e, \pi, \lambda, E, \varphi) \longrightarrow_i (x \texttt{ :=}_{ow}\ ty\ e', \pi, \lambda, E, \varphi), \\
\quad (e,\varphi) \equiv_e (e',\varphi) \Rightarrow (r \texttt{ :=}_{ormw}\ ty\ \texttt{fadd}(x,e), \pi, \lambda, E, \varphi) \longrightarrow_i (r \texttt{ :=}_{ormw}\ ty\ \texttt{fadd}(x,e'), \pi, \lambda, E, \varphi), \\
\quad (es,\varphi) \equiv_e (es',\varphi) \Rightarrow (r \texttt{ := phi}\ ty\ es, \pi, \lambda, E, \varphi) \longrightarrow_i (r \texttt{ := phi}\ ty\ es', \pi, \lambda, E, \varphi), \\
\quad (\texttt{if}\ 0\ \texttt{then}\ \pi_1\ \texttt{else}\ \pi_2, \pi, \lambda, E \cup \{(\pi, \texttt{yes}, \pi_1), (\pi, \texttt{no}, \pi_2)\}, \varphi) \longrightarrow_i (\texttt{br}\ \pi_2, \pi, \lambda, E \cup \{(\pi_1, \texttt{seq}, \pi)\}, \varphi), \\
\quad n \neq 0 \Rightarrow (\texttt{if}\ n\ \texttt{then}\ \pi_1\ \texttt{else}\ \pi_2, \pi, \lambda, E \cup \{(\pi, \texttt{yes}, \pi_1), (\pi, \texttt{no}, \pi_2)\}, \varphi) \longrightarrow_i (\texttt{br}\ \pi_2, \pi, \lambda, E \cup \{(\pi_2, \texttt{seq}, \pi)\}, \varphi), \\
\quad \lambda(\pi_1) = \lambda(\pi_2) \Rightarrow \\
\qquad\quad (\texttt{if}\ e\ \texttt{then}\ \pi_1\ \texttt{else}\ \pi_2, \pi, \lambda, E \cup \{(\pi, \texttt{yes}, \pi_1), (\pi, \texttt{no}, \pi_2)\}, \varphi) \longrightarrow_i (\texttt{br}\ \pi_1, \pi, \lambda, E \cup \{(\pi_1, \texttt{seq}, \pi)\}, \varphi),\ \ldots \\
(\ \longrightarrow_B\ ) \\
\quad i < \texttt{size}(B) \wedge (B_1, in, B_2) = \texttt{cut}(B,i) \wedge (in, \pi, E, \varphi) \longrightarrow_i (in', \pi, E, \varphi') \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \Rightarrow (B, \pi, i, \lambda, E, \varphi) \longrightarrow_B (B_1@[in']@B_2, \pi, i+1, \lambda, E, \varphi'), \\
\quad i < \texttt{size}(B) \wedge (B_1, in, []) = \texttt{cut}(B,i) \wedge (in, \pi, E, \varphi) \longrightarrow_i (in', \pi, E', \varphi') \\
\qquad \wedge (\pi, cl, \pi') \in E' \wedge (\neg\exists \pi_1\ cl_1 . \pi' \neq \pi_1 \wedge (\pi, cl_1, \pi_1) \in E') \wedge B' = \texttt{get_B}(\lambda, \pi') \\
\qquad \wedge (r \texttt{ := phi}\ ty\ es) = \texttt{nth}(B', i') \wedge (e, \pi') \in es \wedge B'' = \texttt{insert_i}(B', i', r \texttt{ := } e) \\
\qquad\qquad\qquad\qquad\qquad\qquad \Rightarrow (B, \pi, i, \lambda, E, \varphi) \longrightarrow_B (B_1@[in'], \pi, i+1, \lambda[\pi' \mapsto \texttt{up}(B'')], E', \varphi'), \\
\quad (\pi_1, \texttt{seq}, \pi') \in E \wedge (\neg\exists \pi_2\ cl . (\pi_2, cl, \pi') \in E \wedge \pi_2 \neq \pi_1) \wedge B_1 = \texttt{get_B}(\lambda, \pi_1) \wedge B_2 = \texttt{get_B}(\lambda, \pi') \\
\qquad\qquad \Rightarrow (B, \pi, i, \lambda, E \cup \{(\pi_1, \texttt{seq}, \pi')\}, \varphi) \longrightarrow_B (B, \pi, i, \lambda[\pi_1 \mapsto \texttt{up}(B_1@B_2)], \texttt{cone}(E, \pi_1, \pi'), \varphi), \\
\quad i = \texttt{size}(B) \wedge (\pi, cl, \pi') \in E \wedge B' = \texttt{get_B}(\lambda, \pi') \Rightarrow (B, \pi, i, \lambda, E, \varphi) \longrightarrow_B (B', \pi', 0, \lambda[\pi \mapsto \texttt{up}(B)], E, \varphi), \\
(\ \equiv_{cfg}\ /\ \equiv_p\ /\ \equiv_P\ /\ \equiv_\Omega\ ) \\
\quad B' = \texttt{get_B}(\lambda, \pi_0) \wedge (B, \pi_0, 0, \lambda, E, \emptyset) \longrightarrow_B^* (B', \pi', i, \lambda', E', \varphi) \Rightarrow (N, \pi_0, \lambda, E) \equiv_{cfg} (N, \pi_0, \lambda', E'), \\
\quad G \equiv_{cfg} G' \Rightarrow (g, ar, rs, G) \equiv_p (g, ar, rs, G'),\ tid \in Tid \wedge \mu(tid) \equiv_p p' \Rightarrow (Tid, \mu) \equiv_P (Tid, \mu[tid \mapsto p']) \\
\quad g \in \texttt{dom}(()\Omega) \wedge \Omega(g) \equiv_p p' \Rightarrow \Omega \equiv_\Omega \Omega[g \mapsto p']\}
\end{array}$$

Figure 11 Some Equational Rules for the Programming Language Syntax

4.1 Equational Rules on Program Syntax

The PLS framework is parameterized by a set of equations for programs. These equations capture the syntactic dependency of the programs, so that PLS parameterized by these equations is able to prove that a particular optimization semantically preserves the program meanings. One such example is the simple code motion optimization (SCM). In Fig. 1, we transform (c) to (a) by SCM, and program piece (a) semantically preserves (c).

Here, to prove the semantic preservation of an optimization of programs in Fig. 3, we specify an equation set ($\texttt{eq}$) in Fig. 11. This set enables PLS to prove the semantic preservation of constant propagation (for registers), simple redundant elimination (for registers), and SCM optimizations for all programs in the language in Fig. 3. In the following section, we prove the semantic preservation of SCM as an example. In the $\texttt{eq}$ set, $\equiv_e$,$\equiv_{cfg}$, $\equiv_p$, $\equiv_P$, and $\equiv_\Omega$ contain equation rules for expressions (e), CFGs, functions p, programs (P), and function databases, respectively. $(e, \varphi) \equiv_e (e', \varphi)$ syntactically equates two states with the forms (e, φ) and (e', φ), where φ is a mapping from registers to integer constants. $\texttt{sem}(op)$ is a semantic interpretation of the operator op. We list only a subset of rules in $\equiv_e$. There are some transition rules saying that if two sub-expressions e and e' can be equated, then expressions containing the sub-expressions $e_1[e]$ and $e_1[e']$ are equal. $\equiv_{cfg}$, $\equiv_p$, $\equiv_P$, and $\equiv_\Omega$ syntactically equates two CFGs, two functions, and two programs, and two function databases. These equational rules partition the respective domains into equivalent classes. For example, $\equiv_P$ partitions the program domain set into a set of equivalent classes, where two programs P and P' are in the same class if $P \equiv_P P'$. To support the computation of the $\equiv_{cfg}$ definition, we also need two sets of term rewriting rules, $\longrightarrow_i$ and $\longrightarrow_B$, on instructions and basic blocks. $\longrightarrow_i$ rewrites a state $(in, \pi, \lambda, E, \varphi)$ to another state with the same components, where in is the instruction, π is the basic block number in resides in, λ and E are λ functions and edge sets for the CFG containing the instructions, and φ is the mapping described above. For any state with a fixed instruction, $\longrightarrow_i$ has only one way of rewriting. $\longrightarrow_i$ definition in Fig. 11 lacks some rules for function calls, returns, and type castings. The $\longrightarrow_B$ describes a set of rewriting rules for basic blocks. Its transition state is a tuple of $(B, \pi, i, \lambda, E, \varphi)$, where B is a basic block in the form of a list of instructions, and i is the current position pointer pointing to an instruction in the block. $\longrightarrow_B$ is confluent. The complicated second rule of $\longrightarrow_B$ means that when a rewrite from a binary branching operation to an unconditional branching operation happens, we also need to detect if it affects the `phi` instruction in a block. If a block has a unique incoming edge, then the `phi` instruction can just be a normal register assignment. The third rule in $\longrightarrow_B$ says that we can merge two basic blocks B and B' together if there is only an unconditional jump from B to B', and B' has no other incoming edges. $\longrightarrow_B$ is complicated because the equation rule set $\equiv_{cfg}$ depends on transition states $\longrightarrow_B$, so we want to make every basic block computed from a transition state of $\longrightarrow_B$ to be well-formed.

4.2 The Operational Program Semantics

Here, we define the operational semantics for a program based on the program syntax and instruction semantics in Sec. 2.1 and the axiomatic program semantics described in Sec. 2.2. The axiomatic program semantics is too abstract, and it is hard to be used by PLS to prove an optimization semantic preservation property on a large language. The operational program semantics $\Longrightarrow$ 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. For each thread, it captures the out-of-order execution behaviors in a block. Hence, the system does not allow speculative reads and writes and is stronger than ATRCM that allows speculative reads in the control fence definition (Sec. 3.1). Even though the system is stronger, we are still able to use it with PLS to prove the semantic preservation of the simple code motion optimization. Similar to the $\overline{\texttt{sbs}}$ and $\overline{\texttt{dds}}$ in Sec. 2.2, we define a $\overline{\texttt{pos}}$ as a family of $\overline{\texttt{po}}$, one for each thread, which has the same functionality as `po`, but it is defined on pairs of action-IDs. A part of the operational semantics is described in Fig. 12.

The operational semantics of a program P is defined as a labeled transition semantics as $\sigma \Longrightarrow_l \sigma'$ where σ and σ' are states and l is a memory event acting as the label in a transition. (c) in Fig. 12 is the main rule in the transition semantics. A state is defined as a tuple (having type name: $State$) of $(\Omega, Tid, Loc, \mu, \overline{\texttt{pos}}, Na, n, \overline{\Pi}, \Pi, \Phi, \Theta, R, \Upsilon, \Gamma, T, \rho, \texttt{rf})$, where Ω is a function database, Tid is a set of thread-IDs, Loc is a set of locations, μ is the μ function in Fig. 3, $\overline{\texttt{pos}}$ is a family of $\overline{\texttt{po}}$ relations described above, Na is a family of function names recording the current function name for each thread, n is a global counter for labeling `CAF` fences, $\overline{\Pi}$ is a family of dynamic block numbers, each of which acts as the dynamic block number counter for each thread, Π is a family of basic block numbers, Φ is a family of registers, Θ is a family of program counters (type $TID \to (A\ \mathsf{set}) \times (A\ \mathsf{set})$) that point out the next possible instructions to execute for a thread, and also have a set of finishing executing instructions, Υ and R are similar entities described in Sec. 2.2, Γ is a family (type $TID \to (loc \to T \times val)$) of observable memory snapshots, T is a time point set, ρ is a mapping from time points to memory events (type $\mathbb{N} \to \mathsf{Event}$), and `rf` is the accumulated reads-from relation (type $T \times T$) under construction along with the transitions. In the transition $\sigma \Longrightarrow_a \sigma'$, the transition state is (l, σ') (type $\mathsf{Event} \times State$). A program execution and its corresponding memory execution are defined similarly as the ones in Sec. 2.2.

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

$$\texttt{(a)} \quad \frac{\begin{array}{l}(N, \pi_0, \lambda, E) = G \wedge \lambda(\pi) = \beta \wedge \texttt{ins}(\beta, d) = in \wedge \texttt{is\_C}(\beta, d) \wedge (cl, ac) = \psi_{\Omega}^{tid}(g, n, in, d, \varphi, \delta, \Gamma(tid), R) \\ \wedge(\pi, l, \pi') \in E \wedge \lambda(\pi') = \beta' \wedge D = \texttt{form\_D}(\overline{\pi} + 1, \pi', \beta') \wedge \rho' = \rho[\sqcup(T)\texttt{+1} \mapsto (tid, d, ac)] \\ \wedge \overline{\texttt{po}}' = \texttt{gen\_}\overline{\texttt{po}}(G, \overline{\pi}\texttt{+1}, \pi', \overline{\texttt{po}}, D, \beta', \varphi) \wedge W = \texttt{gen}(\overline{\texttt{po}}', D, D)\end{array}}{\begin{array}{l}\Big(tid, G, \overline{\pi}, \pi, \overline{\texttt{po}}, g, n, T, \rho, \varphi, \delta, \Gamma, R, \Upsilon, (\{d\}, S)\Big) \longrightarrow_{\Omega} \\ \quad \Big(\overline{\pi}\texttt{+1}, \pi', \overline{\texttt{po}}', g, n, T \cup \{\sqcup(T)\texttt{+1}\}, \rho', \varphi, \delta, \Gamma, R, \Upsilon[(tid, d) \mapsto (g, n, in)], (W, \emptyset), \emptyset\Big)\end{array}}$$

$$\texttt{(b)} \quad \frac{\begin{array}{l}(N, \pi_0, \lambda, E) = G \wedge \lambda(\pi) = \beta \wedge \texttt{ins}(\beta, d) = in \wedge \neg\texttt{is\_C}(\beta, d) \wedge \Gamma(tid) = \gamma \wedge \Gamma(tid') = \gamma' \\ \wedge \gamma'(x) = (t, v) \wedge (\varphi', \delta, \texttt{R}_{v,o}^{x}) = \psi_{\Omega}^{tid}(g, n, in, d, \varphi, \delta, \gamma[x \mapsto (t, v)], R) \\ \wedge W' = \texttt{gen}(\overline{\texttt{po}}, D - (S \cup \{d\}), W) \wedge \texttt{rf} = \{\texttt{fst}(\Gamma(tid')(x)), \sqcup(T)\texttt{+1})\} \wedge \rho' = \rho[\sqcup(T)\texttt{+1} \mapsto (tid, d, \texttt{R}_{v,o}^{x})]\end{array}}{\begin{array}{l}\Big(tid, G, \overline{\pi}, \pi, \overline{\texttt{po}}, g, n, T, \rho, \varphi, \delta, \Gamma, R, \Upsilon, (W \cup \{d\}, S)\Big) \longrightarrow_{\Omega} \\ \quad \Big(\overline{\pi}, \pi, \overline{\texttt{po}}, g, n, T \cup \{\sqcup(T)\texttt{+1}\}, \rho', \varphi', \delta, \Gamma, R, \Upsilon[(tid, d) \mapsto (g, n, in)], (W', S \cup \{d\}), \texttt{rf}\Big)\end{array}}$$

$$\texttt{(c)} \quad \frac{\begin{array}{l}\texttt{sbs} = \texttt{gen\_sb}(\texttt{T}, \rho, P, \Upsilon, \Omega) \wedge \texttt{dds} = \texttt{gen\_dd}(\texttt{T}, \rho, P, \Upsilon, \Omega) \wedge \texttt{sat}(Tid, Loc, T, \rho, \texttt{sbs}, \texttt{dds}, \texttt{rf} \cup \texttt{rf'}) \\ \wedge tid \in \texttt{TID} \wedge \overline{\texttt{pos}}' = \overline{\texttt{pos}}[tid \mapsto \overline{\texttt{po}}'] \wedge \overline{\Pi}' = \overline{\Pi}[tid \mapsto \overline{\pi}'] \wedge \overline{\Pi}' = \overline{\Pi}[tid \mapsto \pi'] \wedge \Phi' = \Phi[tid \mapsto \varphi'] \\ \wedge \Delta' = \Delta[tid \mapsto \delta'] \wedge \Theta' = \Theta[tid \mapsto \Theta'] \\ \wedge \Big(tid, \mu(tid), \overline{\Pi}(tid), \Pi(tid), \overline{\texttt{pos}}(tid), Na(tid), n, T, \rho, \Phi(tid), \Delta, \Gamma, R, \Upsilon, \Theta(tid)\Big) \longrightarrow_{\Omega} \\ \qquad \Big(\overline{\pi}', \pi', \overline{\texttt{po}}', g', n', T', \rho', \varphi', \Gamma', R', \Upsilon', \theta', \texttt{rf'}\Big)\end{array}}{\begin{array}{l}\Big(\Omega, Tid, Loc, \mu, \overline{\texttt{pos}}, Na, n, \overline{\Pi}, \Pi, \Phi, \Delta, \Theta, \Gamma, R, \Upsilon, T, \rho, \texttt{rf}\Big) \Longrightarrow_{\rho'(\sqcup(T'))} \\ \Big(\Omega, Tid, Loc, \mu, \overline{\texttt{pos}}', Na', n', \overline{\Pi}', \Pi', \Phi', \Theta', \Gamma', R', \Upsilon', T', \rho', \texttt{rf} \cup \texttt{rf'}\Big)\end{array}}$$

Figure 12 Part of The Operational Program Semantics

Rules `(a)` and `(b)` (Fig. 12) are sample rules for the transition relation $\longrightarrow_{\Omega}$ that connects between the transition $\Longrightarrow$ and the single-instruction semantics defined in Sec. 2.1. Its input and output states are basically a single-threaded version of $\Longrightarrow_l$, except a few items are different. In a thread *tid*, $\Longrightarrow_l$ selects one of the next possible instructions in θ to execute. The `(a)` rule deals with the transition after executing a branching state at the end of a dynamic 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. The function $\texttt{gen\_}\overline{\texttt{po}}$ takes the existing $\overline{\texttt{po}}$ and a dynamic block and generates a new $\overline{\texttt{po}}'$ containing all of the relations in the $\overline{\texttt{po}}$ relation, all of the program order relations between instructions in the dynamic block, and the program order relations between the old-instructions in $\overline{\texttt{po}}$ and the instruction in the new dynamic block. $\texttt{gen\_}\overline{\texttt{po}}$ happens once when a new dynamic basic block is generated. Similarly, `gen_sb` generates new `sbs` relations based on the current executed instruction information and post history of `sbs`, and `gen_dd` generates new `dds` relations based on the current executed instruction information and post history of `dds`.

In this section, we have mentioned all of the necessary components to support the definition and usage of PLS. We will introduce PLS in the next section.

4.3 Per-Location Simulation

Based on the program semantics in Sec. 4.2, a set of program traces (Ξ : ($\mathsf{Event} \times \mathit{State}$) list) can be generated. Each transition state of a program trace $\xi \in \Xi$ has the form (l, σ') where l and σ' are from the labeled transition $\sigma \Longrightarrow_l \sigma'$. Here, l is a memory event whose action (Sec. 2.1). To generate a memory execution from ξ, we drop the σ' element in every transition state in ξ as a new sequence ξ'. To utilize PLS, we also define a `ms` function in Fig. 13 to mask all memory events in an memory execution that are not memory operations.

Now, we define PLS (Fig. 13). the `init` function initializes a state with the input of a function database with finite domain, a **non-empty finite set** of memory locations, a program μ function and a finite thread-ID set. A simple PLS relation is defined as $\texttt{PLS}(\sigma', \sigma)$, where σ' and σ are running program states for a compiled program piece and its original one, respectively.

$$
\begin{array}{l}
\texttt{ms}(l) \triangleq \texttt{IF is\_mem\_op } (l) \texttt{ THEN } l \texttt{ ELSE } \tau \\
\sigma \Longrightarrow_{\tau} \sigma' \triangleq \sigma \Longrightarrow_{l} \sigma' \wedge \texttt{is\_}\tau(\texttt{ms}(l)) \\
\sigma \Longrightarrow_{\texttt{not\_}\tau} \sigma' \triangleq \sigma \Longrightarrow_{l} \sigma' \wedge \neg\texttt{is\_}\tau(\texttt{ms}(l)) \qquad \sigma \Longrightarrow_{\texttt{not\_}x} \sigma' \triangleq \sigma \Longrightarrow_{l} \sigma' \wedge \neg\texttt{is\_}\tau(\texttt{ms}(l)) \wedge \neg\texttt{has\_loc}(l,x) \\
\Longrightarrow_{1} \triangleq \Longrightarrow_{\tau}^{*} \Longrightarrow_{\texttt{not\_}\tau} \qquad \Longrightarrow_{n} \triangleq \underbrace{\Longrightarrow_{1} \dots \Longrightarrow_{1}}_{\text{n}} \qquad \sigma \Longrightarrow_{x} \sigma' \equiv \sigma \Longrightarrow_{l} \sigma' \wedge \neg\texttt{is\_}\tau(\texttt{ms}(l)) \wedge \texttt{has\_loc}(l,x) \\
\\
\texttt{init}(\Omega, Loc, Tid, \mu) \triangleq \quad (\Omega, Tid, Loc, \mu, \emptyset, \{(tid, \top) | tid \in Tid\}, 0, \{(tid, 0) | tid \in Tid\}, \\
\qquad\qquad \{(tid, \pi_0) | \exists N\ \lambda\ E.\mu(tid) = (N, \lambda, \pi_0, E)\}, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset) \\
\texttt{PLS}(\sigma', \sigma) \triangleq \\
\quad \forall x \in \texttt{Locs} \\
\qquad \forall l'\ \sigma_1'.\sigma' \Longrightarrow_{\{x,l'\}} \sigma_1' \Rightarrow \\
\qquad\quad (\exists a\ \sigma_1.\sigma \Longrightarrow_{\texttt{not\_}x}^{*} \Longrightarrow_{\{x,l\}} \sigma_1 \wedge \texttt{same\_val}(l,l') \wedge \texttt{PLS}(\sigma_1', \sigma_1)) \\
\texttt{PLS}^{\texttt{eq}}((\Omega, Loc, Tid, \mu'), (\Omega, Loc, Tid, \mu)) \triangleq \\
\qquad \exists \mu_1\ \Omega'.(Tid, \mu) \equiv_{P} (Tid, \mu_1) \wedge \Omega \equiv_{\Omega} \Omega' \wedge \texttt{PLS}(\texttt{init}(\Omega', Loc, Tid, \mu'), \texttt{init}(\Omega', Loc, Tid, \mu_1))
\end{array}
$$

Figure 13 Per Location Simulation Definitions

There are also some syntactic sugars, based on the labeled transition system $\Longrightarrow_a$, defined in Fig. 13. In these definitions, $\texttt{is\_}\tau$ checks if a label (memory event) is a τ one, `has_loc` checks if a memory event is accessing a given memory location, and `same_val` checks if two events have the same value. $\Longrightarrow_{\{x,a\}}$ means the transition has the event a, and it also accesses the location x. $\texttt{PLS}^{\texttt{eq}}$ is the PLS definition parameterized by an equation set `eq`. One such example is in Sec. 4.1. $\texttt{PLS}^{\texttt{eq}}(\texttt{init}(\Omega, Loc, Tid, \mu'), \texttt{init}(\Omega, Loc, Tid, \mu'))$ means that program $\texttt{P}'$ preserves the meaning of program `P` in the initial environment context (`P` simulates $\texttt{P}'$).

For every memory location x, $\texttt{PLS}(\sigma, \sigma')$ guarantees the order of the memory events in every trace at a particular location. Without a equation set, the simple PLS can prove the semantic preservation of the example (a) with respect to (b) in Fig. 1. With a equation set `eq`, the PLS framework is more powerful. It enables the semantic preservation proof of (a) with respect to (c) in Fig. 1.

To show that per-location simulation really is a simulation relation, we show below that it is reflexive and transitive.

▶ **Theorem 9.** A per-location simulation builds a reflexive and transitive relation.

In the following section, we will show an example of using this form of equivalence relation to prove a compiler optimization preserving program meaning.

4.4 Example: Proving SCM Preserving Program Semantics

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). What they did was to define the instruction level semantics for a target language (a subset of LLVM), and pack the semantics with Morpheus's control flow graph semantics and a set of axioms from the sequential memory model. Then, based on their bisimulation theories on top of Morpheus, they proved the optimization correctness.

Here, we utilize the domain specific language of Morpheus to prove the semantic preservation of a simple code motion optimization as an example of using PLS. The proof is based on the small language in Fig. 3 with the memory model defined in Sec. 3. Definition and semantics of control flow graphs are found in Figs. 3 and 12. Based on this set of definitions, we want to show that a compiler-optimized program per-location-simulates its original unoptimized program for all possible programs defined in Fig. 3.

For simplicity, we only prove this for programs mentioning up to three threads, as well as the location set and the domain of the input function database are finite. For an optimization, we define the transformation in Morpheus from a program to its transformed program. With a fixed non-empty finite set of memory locations Loc and an `eq` set in Fig. 11, for any possible program $P = (Tid, \mu)$ in the language (Fig. 3), we are able to inductively prove that its compiled program $P' = (Tid, \mu')$ preserves the meaning of P by $\texttt{PLS}^{\texttt{eq}}(\texttt{init}(\Omega', Loc, Tid, \mu'), \texttt{init}(\Omega', Loc, Tid, \mu))$ (as the arrow symbol in Fig. 2). The proof is driven by the program semantics on μ. The inductive search space for the proof grows exponentially with respect to the numbers of threads. This is why we limit the number of threads to a constant number of three in this paper.

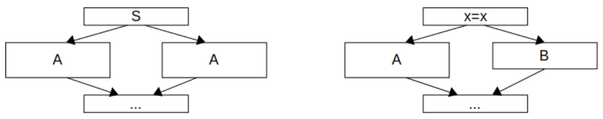


Figure 14 Simple Code Motion Optimization Conceptual Examples

In Figure 14, we show two simple code motion (SCM) optimization examples. The left one changes the conditional branching operation to a non-conditional one because the two targeted basic blocks of the conditional branching operation are the same. The right one also changes the conditional branching operation to a normal `seq` labeled instruction because the condition in the conditional branching operation is always `true`. To express this set of optimizations here, we use the Morpheus tool [33]. The details of the language can be found in the respective paper, and we only discuss some examples and advantages that Morpheus brings us here. Morpheus is used to describe program transformations as rewrites on control flow graphs with temporal logic side conditions. The greatest advantage of using the Morpheus tool on PLS is that its temporal logic side conditions have semantics in terms of first order logic formulas, and the transformation can also easily be turned into a one in the program order of a given CFG; thus, the side conditions can be merged with the weak memory model and then plunged into a PLS proof.

$$\begin{array}{ll}
\texttt{sameOutEdge}(a,b) \equiv & \texttt{stmt}(a) = \texttt{stmt}\ (b) \wedge \texttt{sameEdges}(a,b) \\
 & \vee \texttt{stmt}(a) = \texttt{stmt}\ (b) \wedge \neg\texttt{sameEdges}(a,b) \wedge \texttt{sameOutEdge}(\texttt{next}(a), \texttt{next}(b)) \\
\texttt{leftOpt}(n) \equiv & \exists x\ m_1\ m_2.\texttt{SATISFIED\_AT}\ n.\texttt{sameOutEdge}(\texttt{next}(\texttt{yes}, n), \texttt{next}(\texttt{no}, n)); \\
 & \quad \texttt{relabel\_node}(n, \texttt{skip}); \texttt{move\_edge}((n, \texttt{no}, m_2), m_1) \\
\texttt{rightOpt}(n) \equiv & \exists x\ m_1\ m_2.\texttt{SATISFIED\_AT}\ n.\texttt{stmt}(x\texttt{=}x) \\
 & \quad ; \texttt{relabel\_node}(n, \texttt{skip}); \texttt{move\_edge}((n, \texttt{no}, m_2), m_1)
\end{array}$$

Figure 15 Simple Code Motion Transformations through Morpheus

In Figure 15, the Morpheus formulas `leftOpt` and `rightOpt` define the left and right compiler optimizations from Fig. 14. The `sameOutEdge` formula defines the predicate for checking if two statements are the same and their children have the same outgoing edges or statements. The `leftOpt` and `rightOpt` formulas merge the two outgoing edges and put the new edge in the `yes` branch edge of the branching statement at a given CFG statement labeled n. Certainly, we need other sample compiler optimizations, such as `skip` elimination and dead-code elimination, to tell the whole story of the left and right compiler optimizations in Fig. 14. Interested readers can learn about them in the Morpheus framework paper [33]. We have proved the following theorems about the two optimizations in Fig. 15 in Morpheus plus PLS.

▶ **Theorem 10.** Giving a function database Ω with finite domain, a non-empty finite location set Loc, and an set `eq` in Fig. 11, for any program $P = (Tid, \mu)$ in Morpheus (with a target language in Fig. 3) with a maximum of three threads (in Tid), for any n, let $\mu' = (\lambda\ tid.\texttt{leftOpt}(n)(\mu(tid)))$ (or $\mu' = (\lambda\ tid.\texttt{rightOpt}(n)(\mu(tid))))$, then $\texttt{PLS}^{\texttt{eq}}(\texttt{init}(\Omega, Loc, Tid, \mu'), \texttt{init}(\Omega, Loc, Tid, \mu))$.

In this subsection, we have briefly described an equivalence relation defined on two set of valid executions from two programs, how we can define a compiler optimization in Morpheus, linked it to ATRCM, and used the program representation definition to prove the optimization correctness.

5 Related Work

Lamport probably was the first to define a memory model weaker than sequential consistency for multi-threaded programs [24]. Adve and Hill [1] started defining weak memory orders for memory operations. Focusing just on hardware models: Ahamad et al. [2] axiomatized causal memory and proved some important theorems. Higham et al. [18] formalized SPARC and a number of simpler memory models in both axiomatic and operational styles. Sevcik et al. created a formal verification framework for a small C-like language [49]. The same group [50] later developed the CompCertTSO to verify a compiler from CLight to X86 based on a relaxed memory model.

In the SPARC documentation [44], an axiomatic style similar to the candidate execution model was used. Alglave et al. [3] specified in great detail how to use a candidate execution model to define relaxed memory models and provided several verification tools. The C11 memory model was designed by the C++ standards committee based on a paper by Boehm and Adve [7]. Batty et al. formalized the C11 model with some improvements and proved the soundness of its compilation to X86-TSO [5]. A number of papers [17, 46, 38, 8] found that Batty et al.'s model enabled thin-air behaviors. Vafeiadis et al. [45] found many other problems in Batty et al.'s model and proposed fixes. In 2016, Batty et al. proposed a more concise model for `sc` atomics [4], but the model is stronger than C11; and the `sc` fences there are too weak. Much previous work [46, 36, 22, 21] focused on a fragment of C++ concurrency. From this corpus, we select Lahav et al.'s SRA model [21] to show the soundness of our `acq/rel` atomics. In 2017, Lahav et al. [23] defined a comprehensive C++ model (RC11) based on all previous models, with extra fixes on Batty et al.'s model. In Sec. 1, 2, and 3.3, we discussed this model multiple times. The main problems with the model is that its OUT-OF-THIN-AIR condition is too strong and rules out too many good executions ((e) in Fig. 1). Many previous papers [41, 19, 20] also proposed solutions for out-of-thin-air problems. These models were not in the axiomatic candidate execution fashion, and one of them (the promising memory model [20]), which we have compared in Sec. 1, has been proved to be represented by the IMM model [42]. Chakraborty and Vafeiadis [12] provided a concurrent abstracted memory model for LLVM IR. It provided the semantics for a fragment of LLVM IR memory operations while keeping the model stronger than Lahav et al.'s. The IMM model by Podkopaev et al. [42], based on RC11 and the promising memory model, defined its OUT-OF-THIN-AIR property with a weaker one than the one in RC11. We have shown in Sec. 1 and 4.4 that it is not suitable in handling many thin-air behaviors, and some of its control dependency is too weak so it enables some thin-air behaviors. The essential difference is that IMM is designed to provide a spiritual sample for people to understand how to compile C++ to hardware code, while ATRCM is designed to be used by a PLS to prove properties about a compiler.

The framework introduced in this paper on PLS 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 [39]. Subsequently, much work was published that defined and proved properties about simulations [47, 9, 48, 10, 11]. 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 [16]. Here, we focus on the most related work. One of the most significant achievements in verifying large-language compilers is Leroy's CompCert compiler [6, 25]. Chlipala built verified compilers in Coq from λ-calculus to an idealized machine language [13] and from a small functional language to the machine language [14]. Lochbihler verified a whole-program compiler for multi-threaded Java [29]. Sevcik et al. built CompCertTSO [50], which adapted CompCert's correctness proofs to x86TSO in order to consider the compilation of racy C code. Our domain-specific language for specifying compiler optimizations in this paper is from Mansky and Gunter's work [32, 33].

The first framework to combine a memory model, compiler proof framework, and bisimulation was CompCert [25]. Its bisimulation framework already indicated the weakness of the traditional bisimulation definitions. A bisimulation framework needs to be defined by first distinguishing between programs reaching error states and safe programs, even though CompCert assumed sequential consistency. PLS uses CompCert's simulation framework for error state handling, and focuses on dealing with safe programs. CompCertTSO [50] inherited CompCert's bisimulation framework. Several studies proposed fixes to the bisimulation framework on different topics, such as divergence preservation [27] and creating a program-logic bisimulation framework for the termination-preserving refinement of concurrent programs [26]. All these works enlighten our development of ATRCM and PLS.

6 Conclusion and Future Work

In this paper, we define the major components of ATRCM and the PLS framework, using them to prove that all programs in a large language compiled with SCM optimization semantically preserve their original program's meaning. We also prove theorems about ATRCM as a way of connecting it with previous models [23, 21, 5, 42]. In addition, through the definition of ATRCM we have corrected some mistakes in the definitions of previous models. The special feature of ATRCM is its division of predicates describing concurrency behaviors based on the concept of an abstract machine: this feature implements one predicate to describe single-threaded behaviors and another to describe multi-threaded ones. PLS relates two programs if they generate memory executions that can be simulated through every sequence at a location. To the best of our knowledge, PLS is the first framework that is weaker than a traditional simulation framework [6, 25] to prove the compiler-correctness property of semantic preservation of programs in in a large language. For future work, we plan to use ATRCM + PLS to prove that a wide range of compiler optimizations semantically preserve program meaning in real-world languages like C/C++/LLVM. To do so, we will need to include compiler optimizations like partially redundant elimination, inline expansion, thread inlining, etc. We will also need to include definitions for changed-size memory locations, and memory location creations and deletions in ATRCM.

References

1. Sarita V. Adve and Mark D. Hill. Weak Ordering – A New Definition. *SIGARCH Comput. Archit. News*, 18(2SI):2–14, May 1990.
2. Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. Causal Memory: Definitions, Implementation, and Programming. *Distributed Computing*, 9(1):37–49, Mar 1995.
3. Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Trans. Program. Lang. Syst.*, 36(2):7:1–7:74, July 2014. URL: `http://doi.acm.org/10.1145/2627752`, `doi:10.1145/2627752`.
4. Mark Batty, Alastair F. Donaldson, and John Wickerson. Overhauling SC Atomics in C11 and OpenCL. *SIGPLAN Not.*, 51(1):634–648, January 2016.
5. Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing c++ concurrency. *SIGPLAN Not.*, 46(1):55–66, January 2011. URL: `http://doi.acm.org/10.1145/1925844.1926394`, `doi:10.1145/1925844.1926394`.
6. Sandrine Blazy and Xavier Leroy. Mechanized Semantics for the Clight Subset of the C Language. *Journal of Automated Reasoning*, 43(3):263–288, 2009. URL: `http://dx.doi.org/10.1007/s10817-009-9148-3`, `doi:10.1007/s10817-009-9148-3`.
7. Hans-J. Boehm and Sarita V. Adve. Foundations of the C++ Concurrency Memory Model. *SIGPLAN Not.*, 43(6):68–78, June 2008.
8. Hans-J. Boehm and Brian Demsky. Outlawing Ghosts: Avoiding Out-of-thin-air Results. In *Proceedings of the Workshop on Memory Systems Performance and Correctness*, MSPC '14, pages 7:1–7:6, New York, NY, USA, 2014. ACM.
9. Olaf Burkart, Didier Caucal, and Bernhard Steffen. Bisimulation collapse and the process taxonomy. In Ugo Montanari and Vladimiro Sassone, editors, *CONCUR '96: Concurrency Theory*, pages 247–262, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.
10. Didier Caucal. On the regular structure of prefix rewriting. In A. Arnold, editor, *CAAP '90*, pages 87–102, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg.
11. Didier Caucal. On infinite transition graphs having a decidable monadic theory. In Friedhelm Meyer and Burkhard Monien, editors, *Automata, Languages and Programming*, pages 194–205, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.
12. Soham Chakraborty and Viktor Vafeiadis. Formalizing the Concurrency Semantics of an LLVM Fragment. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO '17, pages 100–110, Piscataway, NJ, USA, 2017. IEEE Press. URL: `http://dl.acm.org/citation.cfm?id=3049832.3049844`.
13. Adam Chlipala. A Certified Type-preserving Compiler from Lambda Calculus to Assembly Language. *SIGPLAN Not.*, 42(6):54–65, June 2007.
14. Adam Chlipala. A Verified Compiler for an Impure Functional Language. *SIGPLAN Not.*, 45(1):93–106, January 2010.
15. cppreference.com. The C++11 Standard, 2018. URL: `https://www.cppreference.com`.
16. Maulik A. Dave. Compiler Verification: A Bibliography. *SIGSOFT Softw. Eng. Notes*, 28(6):2–2, November 2003.
17. Mike Dodds, Mark Batty, and Alexey Gotsman. C/C++ Causal Cycles Confound Compositionality. *TinyToCS*, 2, 2013. URL: `http://tinytocs.org/vol2/papers/tinytocs2-dodds.pdf`.
18. Lisa Higham, Jalal Kawash, and Nathaly Verwaal. Weak Memory Consistency Models. Part I: Definitions and Comparisons. Technical report, Department of Computer Science, The University of Calgary, 1998.

19 Alan Jeffrey and James Riely. 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*, LICS '16, pages 759–767, New York, NY, USA, 2016. ACM. URL: `http://doi.acm.org/10.1145/2933575.2934536`, `doi:10.1145/2933575.2934536`.

20 Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. A Promising Semantics for Relaxed-memory Concurrency. *SIGPLAN Not.*, 52(1):175–189, January 2017. URL: `http://doi.acm.org/10.1145/3093333.3009850`, `doi:10.1145/3093333.3009850`.

21 Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. Taming release-acquire consistency. *SIGPLAN Not.*, 51(1):649–662, January 2016. URL: `http://doi.acm.org/10.1145/2914770.2837643`, `doi:10.1145/2914770.2837643`.

22 Ori Lahav and Viktor Vafeiadis. Owicki-gries reasoning for weak memory models. In *Proceedings, Part II, of the 42Nd International Colloquium on Automata, Languages, and Programming - Volume 9135*, ICALP 2015, pages 311–323, Berlin, Heidelberg, 2015. Springer-Verlag.

23 Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. Repairing Sequential Consistency in C/C++11. *SIGPLAN Not.*, 52(6):618–632, June 2017. URL: `http://doi.acm.org/10.1145/3140587.3062352`, `doi:10.1145/3140587.3062352`.

24 L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.*, 28(9):690–691, September 1979.

25 Xavier Leroy. A Formally Verified Compiler Back-end. *J. Autom. Reason.*, 43(4):363–446, December 2009. URL: `http://dx.doi.org/10.1007/s10817-009-9155-4`, `doi:10.1007/s10817-009-9155-4`.

26 Hongjin Liang, Xinyu Feng, and Zhong Shao. 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)*, CSL-LICS '14, pages 65:1–65:10, New York, NY, USA, 2014. ACM. URL: `http://doi.acm.org/10.1145/2603088.2603123`, `doi:10.1145/2603088.2603123`.

27 Xinxin Liu, Tingting Yu, and Wenhui Zhang. Analyzing Divergence in Bisimulation Semantics. *SIGPLAN Not.*, 52(1):735–747, January 2017. URL: `http://doi.acm.org/10.1145/3093333.3009870`, `doi:10.1145/3093333.3009870`.

28 llvm.org. Llvm language reference manual, 2018. URL: `http://releases.llvm.org/6.0.0/docs/LangRef.html`.

29 Andreas Lochbihler. Mechanising a Type-Safe Model of Multithreaded Java with a Verified Compiler. *Journal of Automated Reasoning*, 61(1):243–332, Jun 2018.

30 Sela Mador-Haim, Luc Maranget, Susmit Sarkar, Kayvan Memarian, Jade Alglave, Scott Owens, Rajeev Alur, Milo M. K. Martin, Peter Sewell, and Derek Williams. An Axiomatic Memory Model for POWER Multiprocessors. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification*, pages 495–512, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

31 William Mansky, Dmitri Garbuzov, and Steve Zdancewic. An Axiomatic Specification for Sequential Memory Models. In Daniel Kroening and Corina S. Păsăreanu, editors, *Computer Aided Verification*, pages 413–428, Cham, 2015. Springer International Publishing.

32 William Mansky and Elsa Gunter. A framework for formal verification of compiler optimizations. In Matt Kaufmann and Lawrence C. Paulson, editors, *Interactive Theorem Proving*, pages 371–386, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

33 William Mansky, Elsa L. Gunter, Dennis Griffith, and Michael D. Adams. Specifying and executing optimizations for generalized control flow graphs. *Science of Computer Programming*, 130:2–23, November 2016. `doi:10.1016/j.scico.2016.06.003`.

34 Jeremy Manson, William Pugh, and Sarita V. Adve. The java memory model. *SIGPLAN Not.*, 40(1):378–391, January 2005. URL: `http://doi.acm.org/10.1145/1047659.1040336`, `doi:10.1145/1047659.1040336`.

35 John Mccarthy and James Painter. Correctness of a compiler for arithmetic expressions. pages 33–41. American Mathematical Society, 1967.

36 Yuri Meshman, Noam Rinetzky, and Eran Yahav. Pattern-based Synthesis of Synchronization for the C++ Memory Model. In *Proceedings of the 15th Conference on Formal Methods in Computer-Aided Design*, FMCAD '15, pages 120–127, Austin, TX, 2015. FMCAD Inc.

37 Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer-Verlag, Berlin, Heidelberg, 2002.

38 Brian Norris and Brian Demsky. Cdschecker: Checking concurrent data structures written with c/c++ atomics. *SIGPLAN Not.*, 48(10):131–150, October 2013.

39 David Park. Concurrency and automata on infinite sequences. In Peter Deussen, editor, *Theoretical Computer Science*, pages 167–183, Berlin, Heidelberg, 1981. Springer Berlin Heidelberg.

40 Lawrence C. Paulson. Isabelle: The Next 700 Theorem Provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.

41 Jean Pichon-Pharabod and Peter Sewell. A Concurrency Semantics for Relaxed Atomics That Permits Optimisation and Avoids Thin-air Executions. *SIGPLAN Not.*, 51(1):622–633, January 2016.

42 Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. Bridging the Gap Between Programming Languages and Hardware Weak Memory Models. *Proc. ACM Program. Lang.*, 3(POPL):69:1–69:31, January 2019. URL: `http://doi.acm.org/10.1145/3290382`, `doi:10.1145/3290382`.

43 Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. Simplifying ARM Concurrency: Multicopy-atomic Axiomatic and Operational Models for ARMv8. *Proc. ACM Program. Lang.*, 2(POPL):19:1–19:29, December 2017.

44 CORPORATE SPARC International, Inc. *The SPARC Architecture Manual: Version 8*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

45 Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. Common Compiler Optimisations Are Invalid in the C11 Memory Model and What We Can Do About It. *SIGPLAN Not.*, 50(1):209–220, January 2015.

46 Viktor Vafeiadis and Chinmay Narayan. Relaxed Separation Logic: A Program Logic for C11 Concurrency. *SIGPLAN Not.*, 48(10):867–884, October 2013.

47 Johan Van Benthem. *Correspondence Theory*, pages 167–247. Springer Netherlands, Dordrecht, 1984. URL: `https://doi.org/10.1007/978-94-009-6259-0_4`.

48 Johan van Benthem. *Exploring Logical Dynamics*. Center for the Study of Language and Information, Stanford, CA, USA, 1997.

49 Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. Relaxed-memory Concurrency and Verified Compilation. *SIGPLAN Not.*, 46(1):43–54, January 2011.

50 Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency. *J. ACM*, 60(3):22:1–22:50, June 2013. URL: `http://doi.acm.org/10.1145/2487241.2487248`, `doi:10.1145/2487241.2487248`.