Formalizing the Concurrency Semantics of an LLVM Fragment

Soham Chakraborty Viktor Vafeiadis
Max Planck Institute for Software Systems (MPI-SWS), Germany

Abstract

The LLVM compiler follows closely the concurrency model of C/C++ 2011, but with a crucial difference. While in C/C++ a data race between a non-atomic read and a write is declared to be undefined behavior, in LLVM such a race has defined behavior: the read returns the special 'undef' value. This subtle difference in the semantics of racy programs has profound consequences on the set of allowed program transformations, but it has been not formally been studied before.

This work closes this gap by providing a formal memory model for a substantial fragment of LLVM and showing that it is correct as a concurrency model for a compiler intermediate language: (1) it is stronger than the C/C++ model, (2) weaker than the known hardware models, and (3) supports the expected program transformations. In order to support LLVM's semantics for racy accesses, our formal model does not work on the level of single executions as the hardware and the C/C++ models do, but rather uses more elaborate structures called event structures.

1. Introduction

With the advent of the multi-core era, programming languages such as C/C++ and Java have introduced first-class platform-independent support for concurrent programming. For example, the 2011 ISO C standard (termed C11) introduced a library of atomic operations together with a concurrency model, a complex set of rules detailing which outcomes a concurrent program may produce. This concurrency model is a set of promises that a compiler has to fulfill and that programmers can rely upon.

LLVM is a *state-of-the-art* optimizing compiler that fully supports the C11 concurrency primitives and has some rudimentary support for Java concurrency. As such, it has its own concurrency model, which determines how the various concurrency primitives should be compiled and optimized. In the LLVM documentation, the LLVM concurrency model is described as a slight variant of the C11 model, though the exact correspondence is not clearly specified. In fact, the entire specification of the LLVM model is extremely informal. It consists of some informal prose that often refers to the C11 model, a couple of examples, and more importantly a collection of transformations which should be allowed or disallowed. This list of transformations is arguably the most valuable part of the specification, because it serves as a guideline to the developers implementing the compiler optimizations.

To date, no formal definition of the LLVM concurrency model exists. (Prior LLVM formalizations, such as VeLLVM [28], have restricted attention to sequential programs.) This lack of a formal model has had a negative effect on LLVM concurrency compilation. First, the LLVM compiler is often overly cautious in optimizing code involving shared memory accesses missing out some optimization opportunities (e.g., eliminating redundant atomic accesses). Second, the difference between the C11 and LLVM semantics remains unappreciated, which has led to subtle compiler bugs [7].

In this paper, we formalize the concurrency semantics for a substantial subset of LLVM, and show that the transformations intended to be correct according to the LLVM documentation are indeed so in our formal model. We also show that our model is stronger than the C11 model, meaning that the standard compilation from C/C++ to the LLVM intermediate representation is correct, and weaker than the Total Store Order (TSO) and Power memory models, meaning that compilation to these hardware models is also correct.

The key challenge that our formalization addresses is the different semantics for racy programs between LLVM and C11. In C11, racy programs are completely undefined. In LLVM, however, read-write races are always well-defined: the read may simply return an *arbitrary* value. This seemingly innocent difference has an important impact on the set of allowed program transformations: it enables hoisting loads outside of conditionals and loops but disallows common subexpression elimination (CSE) over acquire-atomic accesses and fences [7].

To model LLVM's semantics for read-write races, we depart from the standard "per candidate execution" axiomatic style of defining memory models (e.g., [1, 2, 16, 22, 24, 25]). As noted by Batty et al. [3], this standard style of defining memory models cannot adequately distinguish between the next two programs in the case they return $a = b = 1$.

$$\left.\begin{array}{l} a = X; \\ \mathsf{if}(a) \\ \quad Y = 1; \end{array}\right\| \begin{array}{l} b = Y; \\ \mathsf{if}(b) \\ \quad X = 1; \end{array} \quad (\mathsf{CYC}) \qquad \left.\begin{array}{l} a = X; \\ Y = 1; \end{array}\right\| \begin{array}{l} b = Y; \\ \mathsf{if}(b) \\ \quad X = 1; \end{array} \quad (\mathsf{LB})$$

For both programs, let the initial state be $X = Y = 0$ and consider whether the outcome $a = b = 1$ should be allowed. In the case of CYC, the outcome is clearly undesirable because it violates the *data-race-freedom* (DRF) property of

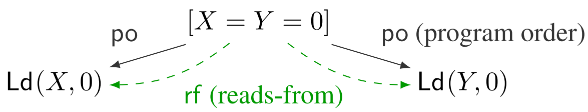

Figure 1. Event structure of the CYC program.

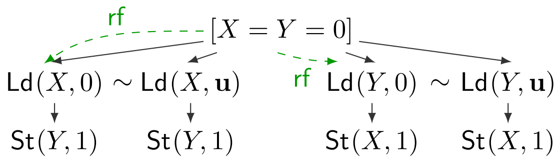

Figure 2. Event structure of LB and LB+false-dep.

a memory model: that is, under *sequential consistent* (SC) semantics [18], the program is race-free; and thus should not have weak behaviors. In the case of LB, however, this outcome must be allowed because it can be observed on ARM [9]. The problem is that the C11-style candidate executions of these two programs yielding $a = b = 1$ are identical (treating the accesses as C11 "relaxed"), and hence any model based on C11-style executions cannot distinguish them. Hardware models [1] typically resolve this problem is by recording (syntactic) control, data, and address dependencies. This, however, does not work for us, because compiler optimizations often remove such dependencies. For example, in the program below may be optimized into LB and so the outcome $a = b = 1$ should be allowed.

$$\begin{array}{l} a = X; \\ \mathsf{if}(a)\ Y = 1; \\ \mathsf{else}\ Y = 1; \end{array} \left\| \begin{array}{l} b = Y; \\ \mathsf{if}(b)\ X = 1; \end{array} \right. \qquad (\mathsf{LB{+}false\text{-}dep})$$

Instead, we base our model on an adaptation of *prime event structures* [27], which we build incrementally via an operational semantics that checks that each appended read event is justified by some existing write event. The beauty of event structures is that they appropriately distinguish between CYC and LB, because they capture multiple executions in a single structure. In our formal model, CYC produces only the event structure shown in Figure 1 where the po-edges represent the program order (the initialization occurs before the two threads) and the rf-edges denote the read-from relation matching each (non-racy) load to the store, from where the load got its value. The event structure justifies only the outcome $a = b = 0$, whereas LB and LB+false-dep also produce the structure in Figure 2, which contains conflicts ($\sim$) among the read events generated from a single load operation along with the po and rf edges. Among others, this event structure justifies the outcome $a = b = 1$, when each $\mathbf{u}$ evaluates to 1 (see §2 for an explanation of $\mathbf{u}$).

In our formal development, we omit LLVM's *monotonic accesses*, which correspond the C11's relaxed accesses, because of the known problems with "out of thin air" reads [3, 5, 12, 23, 26] and the long outstanding problem of providing an adequate solution. We believe that the treatment

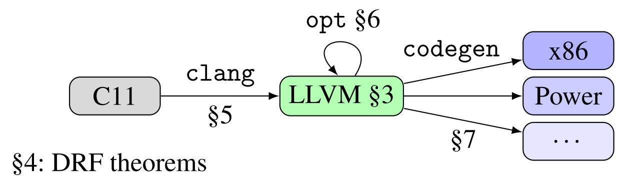

Figure 3. LLVM compilation and paper overview

of monotonic accesses is essentially orthogonal to the treatment of racy non-atomic accesses, which constitutes the focus of this work. Nevertheless, event structures might still be a good mechanism to also model monotonic accesses (e.g., following Jeffrey and Riely [12]). An alternative very promising approach to model monotonic accesses was recently proposed in [13]; incorporating it with our semantics of the remaining accesses is left as future work.

Outline The structure of the remainder of the paper follows largely that of the LLVM compiler (see Figure 3). After some background material in §2, we present our formal LLVM concurrency model in §3, and derive some DRF theorems in §4. Next, in §5, we show that compilation from C11 to our model is correct. In §6, we show that the intended reordering and elimination transformations are allowed under our model. In §7, we show that compilation to the TSO and Power models is correct. We conclude with a discussion of related and future work. The proofs are available online [8].

2. Background

In this section, we introduce some basic notation and a simple programming language. We also discuss the semantics of LLVM's undefined value and of racy programs.

Notation Given a set E and binary relations $R, S \subseteq E \times E$, we write $R; S$ for the relational composition of R and S; formally, $(R; S)(x, y) \triangleq \exists z.\ R(x, z) \wedge S(z, y)$. We write $R^?$, R^+, R^* for the reflexive, the transitive and the reflexive-transitive closures of R respectively, and R^{-1} for the inverse of R. The $[A]$ notation denotes an identity relation on set A, i.e. $[A](x, y) \triangleq x = y \wedge x \in A$. Finally, we write $\mathbf{one}(A)$ for the relation saying that at least one of its components belongs to the set A; i.e. $\mathbf{one}(A)(x, y) \triangleq (x \in A \vee y \in A)$.

Language Our formal model is essentially orthogonal to the syntax of the programming language, but for concreteness we present a simple concurrent imperative language in Figure 4. As already explained in the introduction, in this work we focus on a subset of operations and memory order, excluding monotonic, unordered accesses and fences.

LLVM values, v, can be either constants or the special *undefined value* $\mathbf{u}$, which is roughly a placeholder for any possible value. Given two values v and v', we write $v \preceq v'$ if $v = v'$ or $v' = \mathbf{u}$. Expressions, E, can be built from local variables and values using arithmetic operations.

Commands, C, are sequences of instructions including assignments, shared memory operations, as well as uncon-

$$
\begin{aligned}
v &::= c \mid \mathbf{u} && \text{(Value)}\\
E &::= t \mid v \mid X \mid E+E \mid E*E \mid E \le E \mid \dots && \text{(Expr)}\\
C &::= \mathsf{skip} \mid C;C \mid t = E \mid t = \mathsf{load}_{(\mathrm{NA}|\mathrm{ACQ}|\mathrm{SC})}\ X\\
&\quad \mid \mathsf{store}_{(\mathrm{NA}|\mathrm{REL}|\mathrm{SC})}(X,E) \mid \mathsf{CAS}_{(\mathrm{ACQ\_REL}|\mathrm{SC})}(X,E,E)\\
&\quad \mid \mathit{label}\colon C \mid \mathsf{br}\ \mathit{label} \mid \mathsf{br}\ t\ \mathit{label}\ \mathit{label} && \text{(Cmd)}\\
P &::= X = v; \cdots X = v; \{C \parallel \cdots \parallel C\} && \text{(Program)}
\end{aligned}
$$

Figure 4. Syntax of a minimal programming language. Here, X denotes a shared and t a thread-local location.

ditional and conditional branches. The shared memory operations (load, store, compare-and-swap) are annotated with a *memory order*, o. For loads, this can be non-atomic (NA), acquire (ACQ), or sequentially consistent (SC). For stores, it can be non-atomic, release (REL), or sequentially consistent. For CAS, it can be either acquire-release (ACQ_REL) or sequentially consistent. In increasing strength, these orders are: $\mathrm{NA} \sqsubset \{\mathrm{ACQ}, \mathrm{REL}\} \sqsubset \mathrm{ACQ\_REL} \sqsubset \mathrm{SC}$.

Finally, programs, P, consist of a sequence of initialization writes followed by a fixed parallel composition of thread commands. In our examples, we freely use C-like syntax.

2.1 The Semantics of the Undefined Value

In LLVM, the special undefined value $\mathbf{u}$ is introduced as the result of erroneous computations, such as reading from an uninitialized memory location as a replacement of an arbitrary constant value. This special value propagates through every assignment and arithmetic operation. So, for example, $\mathbf{u}+1 \rightsquigarrow \mathbf{u}$ and even $\mathbf{u}*0 \rightsquigarrow \mathbf{u}$.[1]

The intended semantics is that the compiler may replace $\mathbf{u}$ with any concrete value it finds most convenient, and that moreover different uses of the same $\mathbf{u}$ may be even replaced by different values by the compiler. This weak semantics leads to some rather unexpected behaviors. For example,

$$\mathsf{int}\ t;\ \mathsf{if}(t \le 1 \,\&\&\, t > 1)\ \mathsf{printf}(\text{“Hi”});$$

may print “Hi” even though the if-condition seems unsatisfiable. The reason is that t is uninitialized and hence returns $\mathbf{u}$ in each use, which can be used to satisfy the condition.

Strange though it may seem, LLVM's treatment of uninitialized reads is allowed by the C standard, which says that performing any computation with a value returned by an uninitialized read results in *undefined* behavior.

2.2 The Semantics of Data Races

A program execution is racy if it has two concurrent memory accesses to the same location, such that at least one of them is a write and at least one of them is non-atomic. There are two types of races: *read-write* races, and *write-write* races.

A read-write race occurs between a load and a store or update operation. The intended semantics for LLVM is that in such cases the racy load returns $\mathbf{u}$. Although stronger than the C11 model, where races result in undefined behavior, the LLVM semantics may still lead to unintuitive behavior. Consider the following program where initially $Y = 0$.

$$Y_{\mathrm{NA}} = 1;\ \Big\|\ t = Y_{\mathrm{NA}};\ \mathsf{if}(t \le 1 \,\&\&\, t > 1)\ \mathsf{printf}(\text{“Hi”});$$

As with the program in §2.1, the current program may also print “Hi” just because the non-atomic load of Y is racy and thus returns $\mathbf{u}$. The reason that the treatment of read-write races in the LLVM semantics differs from that in C11 is because LLVM readily performs the following transformation

$$\mathsf{if}(\mathit{cond})\ t = X_{\mathrm{NA}};\quad \rightsquigarrow\quad t' = X_{\mathrm{NA}}; t = \mathit{cond}\ ?\ t' : t;$$

that converts a conditional branch into a conditional move instruction. This transformation may, however, introduce a read-write race if there were some other parallel thread writing to X only when the condition cond is false. The transformation is correct because the target execution uses the racy read value only when the source execution is also racy.

A write-write race occurs whenever both of the accesses racing with one another are stores or updates. In this case, the intended semantics according to the LLVM documentation [19, Section “Optimization outside atomic”] is the same as in C11: even a single consistent execution with a write-write race results in the program having *undefined behavior*. This semantics allows the read-after-write elimination over an acquire access as shown in the following example:

$$
\begin{array}{l}
X_{\mathrm{NA}} = 4;\\
\mathsf{if}(Y_{\mathrm{ACQ}})\\
\quad t = X_{\mathrm{NA}};
\end{array}
\left\|
\begin{array}{l}
X_{\mathrm{NA}} = 8;\\
Y_{\mathrm{REL}} = 1;
\end{array}
\right.
\quad\rightsquigarrow\quad
\begin{array}{l}
X_{\mathrm{NA}} = 4;\\
\mathsf{if}(Y_{\mathrm{ACQ}})\\
\quad t = 4;
\end{array}
\left\|
\begin{array}{l}
X_{\mathrm{NA}} = 8;\\
Y_{\mathrm{REL}} = 1;
\end{array}
\right.
$$

Because of the write-write race on X, the source program has undefined behavior, and hence the transformation is trivially sound. If, however, write-write races were not considered to be undefined behavior, but rather that one of the accesses occurred before the other, then the transformation would be unsound, because in the source program, t would have to contain the final value of X (which may well be 8).

This optimization was performed by LLVM version 3.6 but was later dropped in version 3.7 while fixing another concurrency bug (Bug #22514 [6]). This demonstrates that it is important for LLVM to have a clear concurrency semantics because it affects the validity of basic optimizations.

3. The Formal LLVM Concurrency Model

In this section, we present our formalization of LLVM's concurrency model.

Events The unit of execution in our model is called an *event* and represents either a shared memory access or the creation of a thread. An event, $e = \langle \mathit{id}, C, \mathit{lab} \rangle$, is a tuple consisting of a (unique) identifier for the event, a command representing the thread's continuation after the event, and a

[1] This is needed to justify the distributivity of $+$ over $*$. Consider the transformations: $\mathbf{u}*0 \rightsquigarrow \mathbf{u}*(1-1) \rightsquigarrow \mathbf{u}*1+\mathbf{u}*(-1) \rightsquigarrow \mathbf{u}+\mathbf{u} \rightsquigarrow \mathbf{u}$.

label representing the type of the event. We use the notation e.id, e.code, and e.lab to return the components of an event e. Event labels, lab, are given by the following grammar:

$$lab ::= \mathsf{Ld}_o(\ell, v) \mid \mathsf{St}_o(\ell, v) \mid \mathsf{U}_o(\ell, v, v') \mid \mathsf{Init}$$

We have loads (Ld), stores (St), updates (U), and thread initialization events. A load event is generated from a load or an unsuccessful CAS access, a store is generated from a store access, and update events result from a successful CAS operation. Load, store, and update labels record the location accessed (ℓ), the values read and/or written (v, v'), and the annotated memory order (o).

A *read event* is either a load or an update, whereas a *write event* is either a store or an update. Let $\mathcal{R}$ denote the set of all read events and $\mathcal{W}$ the set of all write events. Given an event e, we write e.lab to return the type of the event (Ld, St, U, Init), and e.loc (resp. e.ord) to return its location (resp. memory order), whenever applicable. If e is a read event, e.rval returns the value read by e. Similarly, if e is a write event, e.wval denotes the value written by e.

Event Structures A *prime event structure* [27] consists of a set V of events equipped with two relations: the program order and the conflict relation. The program order, po, is a strict partial order on events recording when an event precedes another one in the program, whereas the conflict relation, cf, is a symmetric irreflexive relation denoting that two events cannot belong to the same execution. The conflict relation is assumed to be forward-closed with respect to the program order (i.e., cf; po $\subseteq$ cf), which intuitively means that if two events conflict, then so are all their future events.

In this paper, we use an extension of prime event structures, which we call *memory event structures* (or event structures, for short). A memory event structure is a prime event structure with an additional component, the *reads-from relation*, rf, that relates a write to the read events that read from it. We require that whenever rf(w, r), then w.wval $\preceq$ r.rval (racy reads may return $\mathbf{u}$). We denote an event structure G as a tuple $\langle \mathsf{V}, \mathsf{po}, \mathsf{rf}\rangle$ where G.V, G.po, and G.rf denote the respective components of G. (We often omit the "G." when G is clear from the context.) In our setting, the conflict relation, G.cf is a derived relation:

$$G.\mathsf{cf} \triangleq (G.\mathsf{po}^{-1}; G.\mathsf{po}) \setminus (G.\mathsf{po}^{?} \cup G.\mathsf{po}^{-1})$$

It relates all events that are unordered by the program order, but have a common po-ancestor in the same thread. Immediate conflicts are created by read events corresponding to the same program command, but which return different values.

Auxiliary Definitions We define the sets of non-atomic accesses (NA), sequentially consistent accesses (SC), acquire (or stronger) accesses (Acq), and release (or stronger) accesses (Rel).

$$\mathsf{NA} \triangleq \{e \mid e.\mathsf{ord} = \textsc{na}\} \qquad \mathsf{SC} \triangleq \{e \mid e.\mathsf{ord} = \textsc{sc}\}$$
$$\mathsf{Acq} \triangleq \{e \mid e.\mathsf{ord} \sqsupseteq \textsc{acq}\} \qquad \mathsf{Rel} \triangleq \{e \mid e.\mathsf{ord} \sqsupseteq \textsc{rel}\}$$

We say that a release write *synchronizes* (sw) with an acquire read that reads from it. An event a *happens before* an event b, if a reaches b by a path of po or sw edges.

$$G.\mathsf{sw} \triangleq [\mathsf{Rel}]; \mathsf{rf}; [\mathsf{Acq}] \qquad G.\mathsf{hb} \triangleq (\mathsf{po} \cup \mathsf{sw})^{+}$$

Two events race with one another (race) if they are concurrent accesses to the same location (i.e., neither happens before the other), at least one of them is non-atomic and at least one of them is a write.

$$\mathsf{locs} \triangleq \{(e, e') \mid e.\mathsf{loc} = e'.\mathsf{loc}\}$$
$$G.\mathsf{race} \triangleq (\mathsf{locs} \cap \mathbf{one}(\mathcal{W}) \cap \mathbf{one}(\mathsf{NA})) \setminus (\mathsf{hb}^{?} \cup \mathsf{hb}^{-1})$$

An event structure is write-write racy if it contains two write events that race with each other.

$$\mathsf{WWrace}(G) \triangleq \exists w, w' \in \mathcal{W}.\ G.\mathsf{race}(w, w')$$

Next, G.hbW(e) checks if the location read by e is initialized in the event structure G; that is, if there exists a write to it that happens before e.

$$G.\mathsf{hbW}(e) \triangleq \exists w \in \mathcal{W}.\ G.\mathsf{hb}(w, e) \wedge e.\mathsf{loc} = w.\mathsf{loc}$$

Finally, AddRF creates an rf edge between two events in an event structure, G, and returns the updated event structure.

$$\mathsf{AddRF}(G, e, e') \triangleq \langle G.\mathsf{V}, G.\mathsf{po}, G.\mathsf{rf} \cup \{(e, e')\}\rangle$$

3.1 Event Structure Construction

The event structures of a program are constructed incrementally with the operational semantics shown in Figure 5. For a program $P = (X_1 = v_1; \ldots; X_k = v_k; \{C_1 \| \ldots \| C_n\})$, we define the program's initial event structure, $G_{init}(P)$, to be $\langle A \cup B, A \times B, \emptyset\rangle$ where $A = \{a_1, \ldots, a_k\}$ with each a_i having label $\mathsf{St}(X_i, v_i)$ and empty continuation and $B = \{b_1, \ldots, b_n\}$ with each b_i having label Init and continuation C_i.

Each rule from Figure 5 then takes an event structure G and typically extends it by adding one more event to it. An exception is the WW-RACE rule, which checks whether the event structure has a *write-write race*. If so, the program behavior is *undefined* and thus the program can produce any arbitrary event structure G'.

The other rules first call the helper rule BASIC, which selects an event e from the event structure and executes its continuation e.code to get a possible next event e' to be added. The new event has a fresh identifier (i.e., e'.id does not exist in the event structure), and must have not already been added to the event structure (i.e., there does not exist another event e'' with the same label as e' immediately after the previous event, e). Finally, the new event has its label and continuation determined by the thread semantics (which is a parameter to the memory model). Assuming all these premises hold, the rule adds e' to the event structure, recording that is immediately after e in program order.

$$\frac{e \in \mathsf{V} \qquad \forall e'' \in \mathsf{V}.\ e'.\mathsf{id} \neq e''.\mathsf{id} \qquad e.\mathsf{code} \xrightarrow{e'.\mathsf{lab}} e'.\mathsf{code} \qquad \forall e''.\ \mathsf{po}(e, e'') \implies e'.\mathsf{lab} \neq e''.\mathsf{lab}}{\langle \mathsf{V}, \mathsf{po}, \mathsf{rf}\rangle \xrightarrow{e'} \langle \mathsf{V} \cup \{e'\}, (\mathsf{po} \cup \{(e, e')\})^+, \mathsf{rf}\rangle}\ (\textsc{basic})$$

$$\frac{\mathsf{WWrace}(G)}{G \rightsquigarrow G'}\ (\textsc{ww-race}) \qquad \frac{G \xrightarrow{e'} G' \qquad e' \notin \mathcal{R}}{G \rightsquigarrow G'}\ (\textsc{non-read})$$

$$\frac{G \xrightarrow{e'} G' \qquad e' \in \mathcal{R} \qquad e'.\mathsf{rval} = \mathbf{u} \qquad \neg G'.\mathsf{hbW}(e')}{G \rightsquigarrow G'}\ (\textsc{r-uninit})$$

$$\frac{G \xrightarrow{e'} G' \qquad e' \in \mathcal{R} \qquad e'.\mathsf{rval} = \mathbf{u} \qquad G'.\mathsf{hbW}(e') \qquad \exists w.\ G'.\mathsf{race}(w, e')}{G \rightsquigarrow G'}\ (\textsc{r-race})$$

$$\frac{G \xrightarrow{e'} G' \qquad e' \in \mathcal{R} \qquad e'.\mathsf{rval} = w.\mathsf{wval} \qquad G'.\mathsf{hbW}(e') \qquad \neg G'.\mathsf{race}(w, e') \qquad G'' = \mathsf{AddRF}(G', w, e') \qquad \mathsf{isCons}(G'')}{G \rightsquigarrow G''}\ (\textsc{r-norace})$$

Figure 5. Event structure reduction steps.

If the new event is a store, it is simply added to the event structure (NON-READ). If, however, it is a read (i.e., a load or an update), we need also need to check that the value read is correct. There are three cases to consider:

- The location is *uninitialized*, namely there does not exists a write to that location that happens before e. In this case, the read must return the undefined value $\mathbf{u}$. (R-UNINIT)
- The location is initialized, but the access *races* with some other write w. In this case, the read again returns the special undefined value $\mathbf{u}$. (R-RACE)
- The location is initialized and there exists a *non-racy* write w from which the new event can read. Here, we extend $G'.\mathsf{rf}$ with the edge (w, e') to record that e' reads from w, and insist that $e'.\mathsf{rval} = w.\mathsf{wval}$. We then check that the resulting event structure is consistent (see §3.2) and discard it otherwise. (R-NORACE)

The rules can be applied in any sequence, which may result in multiple event structures. We define $[\![P]\!]_{\mathsf{LLVM}}$ to return the set of all event structures generated from $G_{init}(P)$; that is, $[\![P]\!]_{\mathsf{LLVM}} \triangleq \{G \mid G_{init}(P) \rightsquigarrow^* G\}$.

3.2 Event Structure Consistency Checking

We now move on to the consistency checking of an event structure G by $\mathsf{isCons}(G)$. A key constraint to check is that each write justifying a read in G is indeed *visible* to the read.

Overwritten Writes We first consider cases of writes that are *not* visible. One simple case is that of overwritten writes. For example, in the program $X = 1;\ X = 2;\ t = X;$ the load of X should only be able to read from the second store and return the value 2.

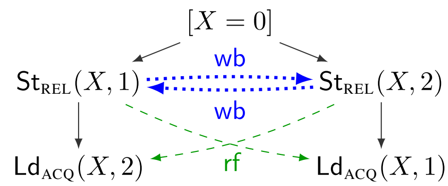

Figure 6. Behavior of Coh forbidden due to a wb cycle.

More generally, we must rule out executions breaking *coherence* (a.k.a. "SC per location" [1]). Consider the program:

$$\begin{array}{l} X_{\text{REL}} = 1; \\ t = X_{\text{ACQ}}; \end{array} \;\Big\|\; \begin{array}{l} X_{\text{REL}} = 2; \\ t' = X_{\text{ACQ}}; \end{array} \qquad (\mathsf{Coh})$$

Here, we ought to prohibit the $t = 2 \wedge t' = 1$ outcome, because it violates coherence. In terms of the C11 memory model, all stores to the same location have to be totally ordered by the modification order, mo. The WR-coherence axiom says that a read cannot read from write earlier in the modification order than some other write that happened before it. Formally, the $\mathsf{rf}^{-1}; \mathsf{mo}; \mathsf{hb}$ should be irreflexive.

In our model, we do not record a modification order, mo. Instead we base our model on a derived partial order over writes to the same location, which is called the *writes-before* order, wb, by Lahav and Vafeiadis [14]. We have already seen two cases of the writes-before order: (1) A write w happening before another write w' to the same location induces the writes-before relation $\mathsf{wb}(w, w')$. (2) A write w' happens before a same location read r and r reads from a write w induces the writes-before relation $\mathsf{wb}(w', w)$. There is one additional case that arises because of the atomicity of update instructions. Consider the following program:

$$\begin{array}{l} X_{\text{REL}} = 1; \\ t = X_{\text{ACQ}}; \end{array} \;\Big\|\; \begin{array}{l} X_{\text{REL}} = 2; \\ t' = X_{\text{ACQ}}; \end{array} \;\Big\|\; \mathsf{CAS}_{\text{ACQ\_REL}}(X, 2, 3); \quad (\mathsf{UCoh})$$

Here, the outcome $t = 3 \wedge t' = 1$ should again be forbidden, because it violates coherence and/or the atomicity of updates. In C11 terms, since $t = 3$, this means that the $X = 1$ store must precede the CAS in modification order. Similarly, since $t' = 1$, the $X = 2$ store must precede the $X = 1$ in modification order. The atomicity of updates, however, says the CAS must immediately follow the $X_{\text{REL}} = 2$ store in modification order; so the $X = 1$ store must precede the $X = 2$ store, which leads to a contradiction.

We therefore define writes-before as follows:

$$G.\mathsf{wb} \triangleq ([\mathcal{W}]; ((\mathsf{brf}; (\mathsf{hb} \cap \mathsf{locs}); \mathsf{brf}) \setminus \mathsf{brf}); [\mathcal{W}])^+$$

where $\mathsf{brf} \triangleq (\mathsf{rf}^{-1})^*$. To prohibit coherence violations, we insist that wb be irreflexive. This rules out the inconsistent behaviors of Coh (see Figure 6) and UCoh.

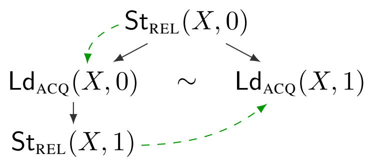

Figure 7. Returning $t = 1$ in Rconflict should be forbidden.

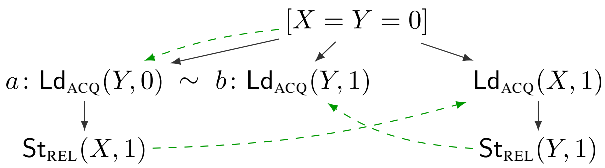

Figure 8. Conflicting write forbids behavior of IncLoop.

Conflicting Writes A second class of writes that a read cannot possibly read from are conflicting writes. For example, in the program

$$X_{\mathrm{REL}} = 0;\ t = X_{\mathrm{ACQ}};\ X_{\mathrm{REL}} = 1; \qquad (\mathsf{Rconflict})$$

t should only be able to read 0, and not 1 from the conflicting $X = 1$ store as shown in Figure 7.

Slightly generalizing this condition, we disallow conflicts between hb-related events, i.e. require cf; hb to be irreflexive. This rules out strange behaviors of examples like the program IncLoop. where initially $X = Y = 0$.

$$X_{\mathrm{REL}} = Y_{\mathrm{ACQ}} + 1; \parallel Y_{\mathrm{REL}} = X_{\mathrm{ACQ}}; \qquad (\mathsf{IncLoop})$$

Returning $X > 1$ is forbidden according to release-acquire semantics. Figure 8 shows that indeed reading $Y = 1$ in the first thread is impossible. If it were allowed, then b would have taken place in conflict with a and in turn would result in $X = 2$ as a possible outcome. Clearly, this would be an *out-of-thin-air* value in any consistent execution.

Non-Conflicting Justifications To avoid weird behaviors reminiscent of "out-of-thin-air" values, we forbid conflicting writes to justify non-conflicting hb-related reads. For example, in the following program $Z = 2$ should not be a valid outcome, where all locations are initialized to zero.

$$Z_{\mathrm{REL}} = 1; \left\| \begin{array}{l} \mathsf{if}(Z_{\mathrm{ACQ}})\ Y_{\mathrm{REL}} = 1; \\ \mathsf{else}\ \ X_{\mathrm{REL}} = 1; \end{array} \right\| \begin{array}{l} \mathsf{if}(X_{\mathrm{ACQ}}\ \&\&\ Y_{\mathrm{ACQ}}) \\ \quad Z_{\mathrm{REL}} = 2; \end{array} \qquad (\mathsf{Cwrites})$$

As shown in Figure 9, if a justifies c and b justifies d then $\mathsf{St}_{\mathrm{REL}}(Z, 2)$ would be an event in the event structure which is actually should not have happened. To restrict such cases, we check that $\mathsf{rf}; \mathsf{hb}^{-1}; \mathsf{rf}^{-1}; \mathsf{cf}$ is irreflexive.

Preserving the Order of Sequential Consistent Accesses The LLVM specification states that the semantics of SC accesses follows that of C11, which puts all the SC events in a total order and applies certain constraints on SC-reads. However, it has recently been shown that the C11 semantics of SC accesses is broken in that the expected compilation

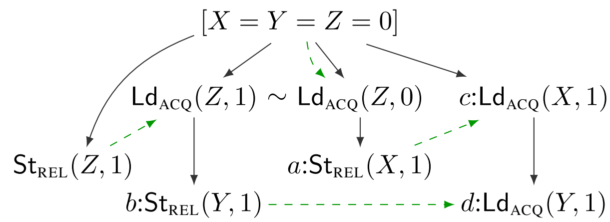

Figure 9. Conflicting justifier forbids behavior of Cwrites.

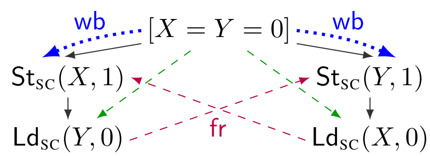

Figure 10. SC constraint forbids $t = t' = 0$ in SB.

schemes to Power and ARM are unsound [17, 21]. (The recent strengthening of Batty et al. [4] is also broken for the same reason.)

Therefore, instead of the C11 semantics, we adapt the solution of Lahav et al. [17]. In their solution, one checks for the acyclicity of a union of relations on the SC events. We say that a read a *reads before* a (different) write b if it reads from a write that was written before b. Formally,

$$G.\mathsf{fr} \triangleq (\mathsf{rf}^{-1}; \mathsf{wb}) \setminus [G.\mathsf{V}] \qquad (\textit{reads before})$$

The "$\setminus[G.\mathsf{V}]$" is takes care of updates because updates are $(\mathsf{rf}^{-1}; \mathsf{mo})$-before themselves. (This definition is a slight adaptation of the definition in Lahav et al. [17] that uses wb instead of the modification order.)

We next define hbsc to denote a restricted hb path between two SC events that either (*i*) does not change location, or (*ii*) starts and ends with a program order edge that changes location, which in turn ensures that the compilation to Power and ARM will have an appropriate fence along the path.

$$G.\mathsf{po}|_{\neq\mathsf{locs}} \triangleq \mathsf{po} \setminus \mathsf{locs}$$

$$G.\mathsf{hbsc} \triangleq [\mathsf{SC}]; \begin{pmatrix} (\mathsf{hb} \cap \mathsf{locs}) \cup \mathsf{po}|_{\neq\mathsf{locs}} \cup \\ (\mathsf{po}|_{\neq\mathsf{locs}}; \mathsf{hb}; \mathsf{po}|_{\neq\mathsf{locs}}) \end{pmatrix}; [\mathsf{SC}]$$

We then require $(\mathsf{hbsc} \cup \mathsf{wb} \cup \mathsf{fr}); [\mathsf{SC}]$ to be acyclic in the event structure. To illustrate this condition, consider the store buffer program with SC accesses, where the outcome $t = t' = 0$ should be forbidden.

$$\begin{array}{l} X_{\mathrm{SC}} = 1; \\ t = Y_{\mathrm{SC}}; \end{array} \left\| \begin{array}{l} Y_{\mathrm{SC}} = 1; \\ t' = X_{\mathrm{SC}}; \end{array} \right. \qquad (\mathsf{SB})$$

The weak behavior is ruled out because the relevant event structure (see Figure 10) contains a $(\mathsf{po} \cup \mathsf{fr}); [\mathsf{SC}]$ cycle.

A slightly more complex example is the following, where the outcome $t = 2 \wedge t' = 0$ should also be forbidden.

$$\begin{array}{l} X_{\mathrm{SC}} = 1; \\ Y_{\mathrm{SC}} = 1; \\ t = Y_{\mathrm{ACQ}}; \end{array} \left\| \begin{array}{l} Y_{\mathrm{SC}} = 2; \\ t' = X_{\mathrm{SC}}; \end{array} \right. \qquad (\mathsf{SCR})$$

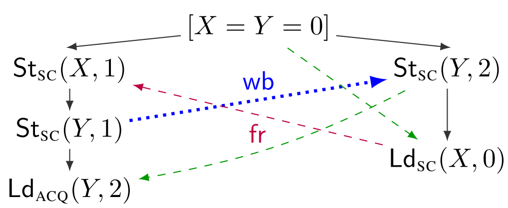

Figure 11. SC constraint forbids $t = 2 \wedge t' = 0$ in SCR.

Again this is so because of a $(\mathsf{po} \cup \mathsf{wb} \cup \mathsf{fr}); [\mathsf{SC}]$ cycle in the relevant event structure (see Figure 11).

Definition of isCons We say that an event structure G is *consistent* if it satisfies all the aforementioned constraints:

$$\begin{aligned} \mathsf{isCons}(G) \triangleq\; & \mathsf{irreflexive}(\mathsf{wb}) \wedge \mathsf{irreflexive}(\mathsf{cf}; \mathsf{hb}) \\ & \wedge \mathsf{irreflexive}(\mathsf{rf}; \mathsf{hb}^{-1}; \mathsf{rf}^{-1}; \mathsf{cf}) \\ & \wedge \mathsf{acyclic}((\mathsf{hbsc} \cup \mathsf{wb} \cup \mathsf{fr}); [\mathsf{SC}]) \end{aligned}$$

3.3 Consistent Executions

So far, we have discussed the construction of the event structures of a program. Since an event structure may contain events arising from multiple executions of a program, we use the function $\mathsf{exec}(G)$ to extract individual (consistent) executions from a fully constructed event structure.

$$\mathsf{exec}(G) \triangleq \left\{ \begin{array}{l} \mathsf{E} = \langle A, G.\mathsf{po} \cap (A \times A), \mathit{rf} \rangle \mid \mathsf{isCons}(\mathsf{E}) \\ \quad \wedge A \subseteq G.\mathsf{V} \;\wedge\; G.\mathsf{hb}; [A] \subseteq (A \times A) \\ \quad \wedge [A]; \mathsf{cf}; [A] = \emptyset \;\wedge\; [A]; G.\mathsf{rf}; [A] \subseteq \mathit{rf} \\ \quad \wedge \forall r \in A.\ (\exists w.\ \mathit{rf}(w, r)) \iff G.\mathsf{hbW}(r) \end{array} \right\}$$

Each execution, $\mathsf{E} \in \mathsf{exec}(G)$, is a consistent conflict-free hb-prefix of the event structure, whose reads-from relation has been extended to provide a justification for each initialized read in the prefix. Note that executions of an event structure constructed by the operational semantics, unlike the event structure itself, may have $(\mathsf{po} \cup \mathsf{rf})$ cycles.

The consistency check together with the requirement that all initialized read events be justified removes some strange behaviors that would otherwise be allowed. Consider the following program, in which the outcome $t = 0 \wedge t' = 1$ should be forbidden.

$$\begin{array}{l} t = Z_{\text{ACQ}}; \\ \mathsf{if}(t) \\ \quad X_{\text{REL}} = 1; \end{array} \left\| \begin{array}{l} \mathsf{if}(Y_{\text{ACQ}}) \\ \quad t' = X_{\text{ACQ}}; \end{array} \right\| \begin{array}{l} Z_{\text{REL}} = 1; X_{\text{REL}} = 1; \\ X_{\text{REL}} = 2; Y_{\text{REL}} = 1; \end{array} \quad (\mathsf{CEX})$$

Consider, for example, the candidate execution of this program highlighted in Figure 12. The highlighted execution is, however, cannot be made consistent because it has to provide a justification for the $\mathsf{Ld}_{\text{ACQ}}(X, 1)$ event. Its only possible justification is the $\mathsf{St}_{\text{REL}}(X, 1)$ event, but this would violate coherence.

3.4 Observable Behaviors

We take the observable behavior of an execution (a conflict-free event structure) to be the set of the last values written to each variable, namely by writes that were not written before some other write (to the same location).

$$\begin{aligned} & \mathsf{Behavior}(G) \triangleq \\ & \quad \{(\ell, v) \mid \exists e \in G.\mathsf{V}.\ v \preceq e.\mathsf{wval} \wedge \nexists e'.\ G.\mathsf{wb}(e, e')\} \end{aligned}$$

The $\preceq$ in the definition above allows treating $\mathbf{u}$ as any possible value. The LLVM-behaviors of a program P are those of its consistent executions.

$$\mathsf{Behavior}_{\mathsf{LLVM}}(P) \triangleq \{\mathsf{Behavior}(G) \mid G \in \mathsf{exec}([\![P]\!]_{\mathsf{LLVM}})\}$$

This definition can straightforwardly be extended to other memory models such as (O)SC, RA, C11, TSO, and Power by substituting the appropriate event structure consistency definition. In these execution-based axiomatic models each execution can be considered as conflict-free event structure.

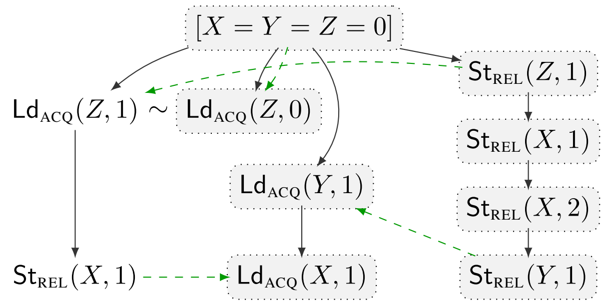

Figure 12. Forbidden execution $t = 0 \wedge t' = 1$ in CEX.

4. DRF Theorems

In this section, we prove that the proposed LLVM model satisfies two expected DRF theorems: DRF-RA and DRF-OSC. These theorems enable developers programming over the model to follow a defensive style of programming and avoid the need of understanding the model. Although our model's intended use is for an intermediate language, our DRF theorems can be seen as sanity checks that the model is not overly weak. For readability, we opted for an informal high-level presentation of the theorems in this section; formal statements and proofs can be found in [8].

4.1 LLVM and Release-Acquire Consistency

Release-acquire (RA) consistency is a strengthening of our model, where during the event structure construction each initialized read *reads from* some *happens-before* write. In other words, the event structure is constructed without using the R-RACE rule. Our first theorem says that if a program does not have any read-write races under RA semantics, then RA-consistency and LLVM-consistency produce the same set of event structures. A trivial corollary of DRF-RA is that if a program contains only atomic operations, then its behaviors under LLVM and RA coincide.

Theorem 1 (DRF-RA)**.** *If under* RA *consistency, a program has no read-write races, then its* LLVM*-consistent behaviors coincide with its* RA*-consistent ones.*

Considering the scenarios based on the read-write races, Theorem 1 follows from the following lemmas.

Lemma 1. *If an* LLVM*-consistent event structure G has no read-write races, then G is also* RA*-consistent.*

Lemma 2. *Given a program P and an* LLVM*-consistent event structure of P with a read-write race, there exists some* RA*-consistent event structure of P with a read-write race.*

4.2 LLVM and (Observable) Sequential Consistency

We model *observable sequential consistency* (OSC) by dropping the WW-RACE rule in the event structure construction and treating all the operations as having SC memory order in the graph consistency checks, $\mathsf{isCons}(G)$. We note that this definition is technically slightly different from the standard SC definition [18]; in particular, writes that are never observed (i.e. read) may appear somewhat loosely ordered. Concretely, consider the following program:

$$\begin{array}{l} X_{\text{SC}} = 1; \\ Y_{\text{SC}} = 2; \end{array} \;\Big\|\; \begin{array}{l} Y_{\text{SC}} = 1; \\ X_{\text{SC}} = 2; \end{array} \qquad \text{(2+2W)}$$

Under SC, the final result cannot be $X = Y = 1$. Yet it is a OSC-behavior: since there are no reads in the program, $\mathsf{wb} = \emptyset$. Intuitively, however, OSC and SC are essentially the same. We conjecture that this is also formally so if we restrict the program behaviors to observations made by the program itself (e.g., recorded in thread-local variables).

We say that a program is RA-race-free under a memory model M, if all concurrent accesses to the same location are either both loads or both SC. Our main theorem states that LLVM-consistency and sequential consistency coincide for programs that are RA-race-free under OSC.

Theorem 2 (DRF-OSC)**.** *If a program is* RA*-race-free under* OSC*, then* OSC *and* LLVM *consistency coincide.*

Note that the RA-race-freedom implies lack of read-write races. Hence, it suffices to prove correspondence between the RA-consistency and OSC in absence of RA races. To do so, we prove the following lemmas.

Lemma 3. *If an* RA*-consistent event structure G is* RA*-race-free, then G is* OSC*-consistent.*

Lemma 4. *For each* RA*-racy* LLVM*-consistent event structure there is a* RA*-racy* OSC*-consistent event structure.*

5. Compilation from C11 to LLVM

In this section, we discuss the correctness of the LLVM's front-end, `clang`, as far as concurrency is concerned. Specifically, we show that the straightforward mapping from C11 memory accesses to LLVM memory accesses used by `clang` is correct with respect to our model.[2]

Compilation correctness means that the behaviors of the target program are included in those of the source program.

[2] Recall that LLVM's memory orders include those of C11's except for *consume* reads, which `clang` converts to the strictly stronger *acquire* reads.

$\downarrow a \setminus b \rightarrow$	$\mathsf{St}_{\sqsupseteq\text{REL}}$	$\mathsf{Ld}_{\sqsupseteq\text{ACQ}}$	U	$(\mathsf{St}\vert\mathsf{Ld})_{\text{NA}}$	Ld_{SC}
$\mathsf{St}_{\sqsupseteq\text{REL}}$	×	✓	×	✓	✓
St_{SC}	×	✓	×	✓	×
$\mathsf{Ld}_{\sqsupseteq\text{ACQ}}$	×	×	×	×	×
U	×	×	×	×	×
$(\mathsf{St}\vert\mathsf{Ld})_{\text{NA}}$	×	✓	×	✓	✓

Table 1. Safe and unsafe memory access reorderings.

Theorem 3. *For every program P,* $\mathsf{Behavior}_{\mathsf{LLVM}}(P) \subseteq \mathsf{Behavior}_{\mathsf{C11}}(P)$.

First, for the theorem to even hold, we assume that C11's treatment of SC accesses is changed to match the one in our LLVM model (cf. §3.2). Then, the proof of this theorem is straightforward because the remaining difference between the models is only in the treatment of read-write races: in LLVM, such reads may return $\mathbf{u}$ (using R-RACE), whereas in C11, they produce completely arbitrary behavior.

6. Program Transformations/Optimizations

Next, we consider program transformations that may be performed as part of the LLVM optimization passes, and prove their correctness. Generally, a transformation is correct if it does not introduce any new behaviors; namely, if every behavior of the target program is also a behavior of the source program. For space reasons, we just give an overview of the results in this section: the actual proofs can be found in our technical appendix [8].

6.1 Reordering of Independent Memory Accesses

One standard transformation is to reorder two independent instructions, which may improve the locality of computation or enable further optimizations. Two independent adjacent instructions a and b are *safely reorderable* if the following conditions hold: (i) do not access the same location, (ii) they are not both SC, (iii) the first instruction, a, is not an acquire read, and (iv) the second instruction, b, is not a release write. Table 1 summarizes the safe (✓) and unsafe (×) reordering transformations. For the unsafe ones, we provide counterexamples in our technical appendix [8].

Theorem 4. *Reordering two adjacent safely reoderable instructions is a correct transformation.*

To prove this theorem, we show that the safe reordering conditions ensure that the target event structure has fewer behaviors compared to the source event structure. Since, however, each memory access instruction in a program may produce multiple conflicting events in the event structure, reordering two such instructions may result in quite different every structure that are not obvious to relate at first.

Let A and B be the sets of conflicting events generated by instructions a and b respectively. In general, the events in these two sets do not have one-to-one po-correspondence, since an event in A may have multiple immediate posuccessors (all must be in B). To relate the two event structures, we first *expand* the predecessor event set such that the events in these two sets have one-to-one po-relation and then perform the reordering and finally *collapse* the *expanded* event set to create the target event structure.

6.2 Elimination Optimizations

Next, we consider the possible safe deletions of the redundant shared memory accesses in the proposed LLVM model. Some of these transformations are performed in the common subexpression elimination (CSE) or common subexpression elimination optimization passes.

We begin with the deletion of accesses because of another adjacent access to the same location. We have three cases:

Overwritten Write (OW) Deletes the first of two same location adjacent store operations:

$$\mathsf{St}_{o'}(\ell, v'); \mathsf{St}_o(\ell, v) \rightsquigarrow \mathsf{St}_o(\ell, v) \quad \text{when } o' \sqsubseteq o\,.$$

We prove this transformation correct by simulation: we match the target execution steps by the exact same source execution steps, except when the target inserts the event corresponding to the $\mathsf{St}_o(\ell, v)$ store. In this case, we perform two execution steps in the source, adding events for both the eliminated and the remaining store.

Read after Write (RAW) Deletes a load that immediately follows a store to the same location:

$$\mathsf{St}_o(\ell, v); \mathsf{Ld}_{o'}(\ell) \rightsquigarrow \mathsf{St}_o(\ell, v) \quad \text{when } o' \sqsubseteq o\,.$$

We prove this transformation correct again by simulation: when the target inserts the event corresponding to the $\mathsf{St}_o(\ell, v)$ store, the source also inserts a $\mathsf{Ld}_{o'}(\ell, v)$ event reading from that store.

Read after Read (RAR) Deletes the second of two adjacent same location reads.

$$\mathsf{Ld}_o(\ell); \mathsf{Ld}_{o'}(\ell) \rightsquigarrow \mathsf{Ld}_o(\ell) \quad \text{when } o' \sqsubseteq o\,.$$

This transformation is correct and in the proof, we simulate the respective load step by two load steps reading from the same write.

In all these cases, LLVM actually performs these transformations only if the deleted instruction is non-atomic.

An access may also be deleted if the access justifying the deletion is non-adjacent, but can be made adjacent by reordering the access to be deleted over all the intermediate accesses. For example, in the program below, the second load of ℓ can be eliminated as follows:

$$\begin{array}{l} a : \mathsf{Ld}_{\text{NA}}(\ell); \\ b : \mathsf{St}_{\text{REL}}(X, 1); \\ c : \mathsf{Ld}_{\text{NA}}(\ell); \end{array} \rightsquigarrow \begin{array}{l} a : \mathsf{Ld}_{\text{NA}}(\ell); \\ c : \mathsf{Ld}_{\text{NA}}(\ell); \\ b : \mathsf{St}_{\text{REL}}(X, 1); \end{array} \rightsquigarrow \begin{array}{l} a : \mathsf{Ld}_{\text{NA}}(\ell); \\ b : \mathsf{St}_{\text{REL}}(X, 1); \end{array}$$

Note that if instead of the release store, there were an acquire load of X between the two loads of ℓ, then the second load cannot be eliminated in this way, because reordering it with the acquire load is unsafe. It may, however, be eliminated, if the first load is moved after the acquire:

$$\begin{array}{l} a : \mathsf{Ld}_{\text{NA}}(\ell); \\ b : \mathsf{Ld}_{\text{ACQ}}(X); \\ c : \mathsf{Ld}_{\text{NA}}(\ell); \end{array} \rightsquigarrow \begin{array}{l} b : \mathsf{Ld}_{\text{ACQ}}(X); \\ a : \mathsf{Ld}_{\text{NA}}(\ell); \\ c : \mathsf{Ld}_{\text{NA}}(\ell); \end{array} \rightsquigarrow \begin{array}{l} b : \mathsf{Ld}_{\text{ACQ}}(X); \\ a : \mathsf{Ld}_{\text{NA}}(\ell); \end{array}$$

Note that in this case the remaining load of ℓ cannot be moved back in its original place.

There are two more cases of eliminating non-adjacent non-atomic accesses, which have a somewhat weaker correctness condition, namely that there has to be no *release-acquire pair* between the eliminated access and the justifying access, which has to be a non-atomic store. A *release-acquire pair* is formed by a release write followed by an acquire read access. The transformations are:

$$\mathsf{St}_{\text{NA}}(\ell, v'); C; \mathsf{St}_{\text{NA}}(\ell, v) \rightsquigarrow C; \mathsf{St}_{\text{NA}}(\ell, v) \qquad \text{(NA-OW)}$$

$$\mathsf{St}_{\text{NA}}(\ell, v); C; \mathsf{Ld}_{\text{NA}}(\ell) \rightsquigarrow \mathsf{St}_{\text{NA}}(\ell, v); C \qquad \text{(NA-RAW)}$$

where C does not have any *release-acquire* pair nor any accesses of location ℓ. For NA-OW, these conditions ensure that the only reads reading from the eliminated store are actually racy, and hence do not need to read from there. Similarly, for NA-RAW, one can show that the eliminated load can always read from the preceding store because there can be no other same-location write in between.

6.3 Speculative Load Introduction

One sound and occasionally useful transformation is to insert a load, whose value is just never used. Assuming that the address, whence the inserted load reads, is valid and allocated, this transformation is trivially correct according to our LLVM model. The load introduction is, however, incorrect according to the C11 model, because it may introduce a data race (and hence undefined behavior in C11). LLVM frequently performs such load introductions in the "simplify CFG" pass; e.g., when hoisting loads outside of conditionals.

6.4 Access Strengthening

Finally, a sound, but not so useful, transformation is strengthening the memory order of memory accesses, i.e., converting some access_o of the program into $\mathsf{access}_{o'}$ where $o \sqsubset o'$. For example, a non-atomic write may be changed into a release write. Soundness of access strengthening follows directly from the *monotonicity* property of the LLVM model. Given $P \xRightarrow{\text{strengthen}} P'$, we show that that for each $G' \in [\![P']\!]_{\mathsf{LLVM}}$, then we can construct a similar event structure in $[\![P]\!]_{\mathsf{LLVM}}$ (actually, the same modulo memory orders).

7. Compilation to x86-TSO and Power

The LLVM compiler generates target code for various architectures including x86 and Power. In this section, we show that the standard compilation schemes from LLVM to x86-TSO and Power shown in Figure 13 are correct.

LLVM	x86-TSO	Power
$\mathsf{store}_{\mathrm{NA}}$	mov	stw
$\mathsf{store}_{\mathrm{REL}}$	mov	lwsync; stw
$\mathsf{store}_{\mathrm{SC}}$	mov; mfence	hwsync; stw
$\mathsf{load}_{\mathrm{NA}}$	mov	lwz
$\mathsf{load}_{\mathrm{ACQ}}$	mov	lwz; lwsync
$\mathsf{load}_{\mathrm{SC}}$	mov	hwsync; lwz; lwsync
$\mathsf{CAS}_{\mathrm{ACQ\_REL}}$	lock cmpxchg	lwsync; M; lwsync
$\mathsf{CAS}_{\mathrm{SC}}$	lock cmpxchg	hwsync; M; lwsync

where $M \triangleq L:$ lwarx; cmpw; bne L'; stwcx.; bne L; L' :

Figure 13. Compilation schemes to x86 and Power.

7.1 Compilation to x86-TSO

In x86, almost all shared accesses get compiled down to plain mov instructions, which serve as both loads and stores. The two exceptions are: (1) sequentially consistent stores, whose compilation includes a memory fence (mfence) after the actual store, and (2) atomic updates, such as compare-swap, which get compiled to special 'locked' instructions.

The correctness of compilation to TSO follows easily from our existing results. First, by monotonicity, we can strengthen all non-atomic accesses of a program to become release stores or acquire loads. (Note that the compilation schemes for non-atomic accesses and the corresponding release/acquire accesses are identical, so we do not incur any performance penalty by this strengthening.) Next, by the DRF-RA theorem, since there are no non-atomic accesses left in the program (and thus no races), the semantics of the program according to the LLVM model corresponds exactly to that according to the RA model. Further, it is well-known that aforementioned compilation scheme to TSO is correct for release/acquire and SC accesses [2]. Putting everything together, we get that the compilation mappings from LLVM to x86-TSO [22] is correct.

7.2 Compilation to Power

The shared memory access instructions in the Power architecture are load (lwz), store (stw), load-linked (lwarx) and store-conditional (stwcx) along with various fence instructions such as hwsync (hardware sync), lwsync (lightweight sync), isync (instruction sync).

As mappings in the Figure 13 show, the non-atomic accesses do not require any fence, release and SC stores have lightweight and full fence respectively before the store. The acquire and SC loads place an lwsync after the load.[3] In addition, SC loads also place a full fence before the load. Updates are implemented using a loop with load-linked and store-conditional instructions, and have fences at both ends.

For the correctness proof, we use the empirically validated axiomatic memory model of Power by Alglave et al. [1] and a recent result by Lahav and Vafeiadis [15]. This result reduces the correctness of compilation to Power to the correctness of reordering independent plain memory accesses of different locations and the correctness of compilation to a stronger model, SPower, which strengthens the Power model of Alglave et al. [1] with the acyclic $(\mathsf{po} \cup \mathsf{rf})$ requirement. The full definition of SPower and the LLVM to SPower compilation correctness proof can be found in [8].

[3] An alternative—possibly more efficient—compilation scheme for acquire loads places a control-dependency to an isync fence instead of an lwsync fence. LLVM, however, does not yet implement this other scheme.

8. Related Work and Conclusion

Our work is the first to formalize a non-trivial fragment of the concurrency model of LLVM. Earlier work [20, 28] has studied LLVM's memory model only for sequential programs. Our formal concurrency model is based on the informal descriptions in the LLVM language reference manual [19]. These descriptions often refer to the C/C++11 memory model [10, 11], which they follow quite closely, both in terms of concurrency primitives and reordering constraints. The major difference, as discussed in Chakraborty and Vafeiadis [7], is the treatment of the read-write data races, whose modelling has been the main subject of this paper. The LLVM informal specification also provides constructs corresponding to C/C++11 relaxed and Java ordinary accesses, which are beyond the purview of this work.

Prior to our work, Pichon-Pharabod and Sewell [23] and Jeffrey and Riely [12] introduce memory models based on event structures as attempts to solve the *out-of-thin-air* problem of relaxed memory models (see [3, 5] for discussions about the problem and some of its implications). Both models attempt to capture all the executions of a program in a single event structure and start with an event structure recording all the potential executions of a program.

Pichon-Pharabod and Sewell [23] have an operational semantics that gradually simplifies the event structure, by either committing an event and pruning the event structure, or transforming it as part of an optimization step. Our operational semantics is of a very different flavor to theirs; we do not attempt to prune a big event structure, but rather enlarge a small event structure.

Jeffrey and Riely [12] target Java and therefore do not guarantee read-read coherence. Their model, however, fails to validate the reordering of independent read events, as shown in §8 of their paper. LLVM's semantics for read-write races is weaker than that of Java, and thus we do not have any problems with read-read reorderings.

Finally, Kang et al. [13] very recently proposed a promising solution to the out-of-thin-air problem, which is based on operational semantics with timestamps and a special reduction step that allows a thread to make a locally certifiable promise to perform a write. While the set up there is quite different from our event structures, it would be extremely useful if one can combine the approaches to extend our semantics to handle LLVM's monotonic accesses. We leave this as future work.

References

[1] J. Alglave, L. Maranget, and M. Tautschnig. Herding cats: modelling, simulation, testing, and data-mining for weak memory. *ACM Trans. Program. Lang. Syst.*, 36(2):7:1–7:74, 2014. doi: 10.1145/2627752.

[2] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *POPL'11*, pages 55–66. ACM, 2011. doi: 10.1145/1926385.1926394.

[3] M. Batty, K. Memarian, K. Nienhuis, J. Pichon-Pharabod, and P. Sewell. The problem of programming language concurrency semantics. In *ESOP'15*, pages 283–307, 2015. doi: 10.1007/978-3-662-46669-8_12.

[4] M. Batty, A. F. Donaldson, and J. Wickerson. Overhauling SC atomics in C11 and OpenCL. In *POPL '16*, pages 634–648. ACM, 2016. doi: 10.1145/2837614.2837637.

[5] H.-J. Boehm and B. Demsky. Outlawing ghosts: avoiding out-of-thin-air results. In *MSPC'14*. ACM, 2014. doi: 10.1145/2618128.2618134.

[6] Bug #22514. Wrong transformation due to semantic gap between C11 and LLVM semantics. https://llvm.org/bugs/show_bug.cgi?id=22514.

[7] S. Chakraborty and V. Vafeiadis. Validating optimizations of concurrent C/C++ programs. In *CGO'16*, pages 216–226. ACM, 2016. doi: 10.1145/2854038.2854051.

[8] S. Chakraborty and V. Vafeiadis. Technical appendix, 2016. Available at http://plv.mpi-sws.org/llvmcs/.

[9] S. Flur, K. E. Gray, C. Pulte, S. Sarkar, A. Sezgin, L. Maranget, W. Deacon, and P. Sewell. Modelling the ARMv8 architecture, operationally: concurrency and ISA. In *POPL'16*, pages 608–621. ACM, 2016. doi: 10.1145/2837614.2837615.

[10] ISO/IEC 14882:2011. Programming language C++.

[11] ISO/IEC 9899:2011. Programming language C.

[12] A. Jeffrey and J. Riely. On thin air reads: Towards an event structures model of relaxed memory. In *LICS'16*. ACM/IEEE, 2016. doi: 10.1145/2933575.2934536.

[13] J. Kang, C.-K. Hur, O. Lahav, V. Vafeiadis, and D. Dreyer. A promising semantics for relaxed-memory concurrency. In *POPL'17*. ACM, 2017.

[14] O. Lahav and V. Vafeiadis. Owicki-Gries reasoning for weak memory models. In *ICALP'15*, pages 311–323, 2015. doi: 10.1007/978-3-662-47666-6_25.

[15] O. Lahav and V. Vafeiadis. Explaining relaxed memory models with program transformations. In *FM'16*, pages 479–495, 2016. doi: 10.1007/978-3-319-48989-6_29.

[16] O. Lahav, N. Giannarakis, and V. Vafeiadis. Taming release-acquire consistency. In *POPL'16*, pages 649–662. ACM, 2016. doi: 10.1145/2837614.2837643.

[17] O. Lahav, V. Vafeiadis, J. Kang, C.-K. Hur, and D. Dreyer. Repairing sequential consistency in C/C++11. Technical Report MPI-SWS-2016-011, MPI-SWS, 2016.

[18] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, 1979. doi: 10.1109/TC.1979.1675439.

[19] LLVM documentation. LLVM atomic instructions and concurrency guide. http://llvm.org/docs/Atomics.html.

[20] N. P. Lopes, D. Menendez, S. Nagarakatte, and J. Regehr. Provably correct peephole optimizations with Alive. In *PLDI*, pages 22–32. ACM, 2015. doi: 10.1145/2737924.2737965.

[21] Y. A. Manerkar, C. Trippel, D. Lustig, M. Pellauer, and M. Martonosi. Counterexamples and proof loophole for the C/C++ to POWER and ARMv7 trailing-sync compiler mappings, 2016. arXiv:1611.01507.

[22] S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-TSO. In *TPHOLs*, pages 391–407, 2009. doi: 10.1007/978-3-642-03359-9_27.

[23] J. Pichon-Pharabod and P. Sewell. A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions. In *POPL'16*, pages 622–633. ACM, 2016. doi: 10.1145/2676726.2676995.

[24] D. E. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst.*, 10(2):282–312, 1988. doi: 10.1145/42190.42277.

[25] R. C. Steinke and G. J. Nutt. A unified theory of shared memory consistency. *J. ACM*, 51(5):800–849, 2004. doi: 10.1145/1017460.1017464.

[26] V. Vafeiadis and C. Narayan. Relaxed separation logic: A program logic for C11 concurrency. In *OOPSLA'13*. ACM, 2013. doi: 10.1145/2509136.2509532.

[27] G. Winskel. Event structures. In *Advances in Petri Nets*, pages 325–392. Springer, 1986. doi: 10.1007/3-540-17906-2_31.

[28] J. Zhao, S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. In *POPL'12*, pages 427–440. ACM, 2012. doi: 10.1145/2103656.2103709.