Formalizing the Concurrency Semantics of an LLVM Fragment

Soham Chakraborty, Viktor Vafeiadis

Presented by
Akshay Gopalakrishnan

August 15, 2021

Introduction

- Compiler intermediate languages(IR) are useful for smoothly translating a source code to target, while also in the process exposing various performance based program transformations.
- While a lot of foundation has been built on the sequential semantics of such IRs, little formal work is done when translating concurrency semantics.
- In fact, most compiler IRs are too conservative in this aspect, thus not being able to perform certain aggressive transformations in this respect.
- Taking LLVM as an example, the concurrency (shared-memory) semantics is also specified informally, with suggestions for program transformations that should be (without formal proof) sound to perform.

What the paper is about

- Formal description of a fragment of LLVM concurrency semantics.
- Model axiomatic using Event Structures as opposed to per-candidate execution.
- Proving standard theorems of Data Race Free (DRF).
- Proving soundness of compilation from C11 to proposed model of LLVM.
- Proving soundness of program transformations.

LLVM

- An intermediate representation(IR) for C/C11 programming language (rudimentary support for Java).
- As an IR, it also has its own concurrency model to support program analysis and optimizations.
- List of transformations it performs is the most valuable part of the specification.
- However, its concurrency model is informal and in relative prose format, thus not able to make use of program transformations to the fullest (eg: Redundant elimination).

Notion of Undefined Value

LLVM assigns a read an undefined value u if required. This can happen, if the read value is possible to return any random value (eg: Uninitialized value). The properties of this value u are:

- $u + 1 = u$
- $u * v = v$ (v is some defined value).
- $u * 0 = u$
- .... and so on.

Example: Sequential

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

The above program can print the text "Hi" as variable "t" is Uninitialized and hence its read will be assigned undefined value u.

Implications in Concurrency - Program Transformations

In C11, any program with data races has undefined behavior. This means, some read value is non-deterministic and hence can be assigned as per LLVM value u.
This semantics is stronger than that of C11, which declares data race programs to have undefined behavior.

Example: Concurrent

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

The above Transformation is possible only because the two writes to X are in a race. In terms of LLVM, the read value will be undefined u. Then the above Transformation could be applied, giving the read any appropriate value (in this case 4).

LLVM Formalization: Event Structures

Prime event structure. Consists of

- Events of a program (V).
- Program order (po) defining intrathread syntactic order.
- Conflict relation (cf), relating events and also implying thread executions that can never occur together.

Memory event structure.

- Prime event structure.
- Reads-from relation (rf) relates every read to the write from which its value comes from.

LLVM Formalization: Auxilary Definitions

- Race(G) : Two events race if at least one of them is a write and at least one of them is non-atomic.
- WWRace(G) : Checks if event structure has race between two writes.
- hbW(e) : Checks if read event e has some write which is initialized that it can read from.
- AddRF(G,e,e') : Creates an rf edge between two events in an event structure.
- locs : $(e, e') \| loc(e) = loc(e')$.

LLVM Formalization: Auxilary Definitions

Access orders/modes

$$\mathsf{NA} \triangleq \{e \mid e.\mathsf{ord} = \mathrm{NA}\} \qquad \mathsf{SC} \triangleq \{e \mid e.\mathsf{ord} = \mathrm{SC}\}$$
$$\mathsf{Acq} \triangleq \{e \mid e.\mathsf{ord} \sqsupseteq \mathrm{ACQ}\} \qquad \mathsf{Rel} \triangleq \{e \mid e.\mathsf{ord} \sqsupseteq \mathrm{REL}\}$$

- Synchronize with (sw): $[Rel]; rf; [Acq]$.
- Happens before (hb): $(po \cup sw)^+$.

LLVM Formalization: Event Structure Building Rules

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

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

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

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

LLVM Formalization: Event Structure Construction

Consider an event structure G.

- Pick an event not in the current event structure (BASIC).
- This event should be po after(if exists a po relation) events in an event structure (BASIC).
- If this is a write, just add this to event (NON-READ).
- If this is a read, then we also add a reads-from relation using AddRF if no race otherwise just add u (R-UNINIT, R-RACE, R-NORACE).

LLVM Formalization: Consistency Rules

- Overwritten writes ¿¿ W-W'-R.
- Conflicting writes ¿¿ Writes on conflicting branches cannot satisfy reads.
- Non-Conflicting Justifications ¿¿ Conflicting writes cannot justify non-conflicting reads (cuz the program would then have OOTA behavior).
- Preserving order of SC accesses.

Overwritten Writes

We want to disallow reads from reading write values that were clearly overwritten. For this, we have the *writes before* relation.

$$wb = [W]; ((brf; ((hb \cap locs); brf) \setminus brf); [W]).$$

We require wb to be irreflexive, thus ensuring overwritten writes are properly ordered.

Overwritten Writes

We want to disallow reads from reading write values that were clearly overwritten. For this, we have the *writes before* relation.

$$wb = [W]; ((brf; ((hb \cap locs); brf) \setminus brf); [W]).$$

We require wb to be irreflexive, thus ensuring overwritten writes are properly ordered.

The intuition behind the writes before definition is the following:

- Firstly, we need to order the writes in a program.
- But we mainly need to only order the writes which are read from (we are talking about execution graphs).
- It is only among these writes that conflict might occur, as reads-from relations would imply some order between writes.
- Also, the conflict may occur only between writes both of which can be read from.
- The brf basically collects also the writes having reads-from relations.
- The wb then only takes those writes ordered by hb.
- Thus, you could say wb is a subset of hb restricted to writes which are definitely read from some read in the program.

Conflicting Writes

If a read is justified by a write it conflicts, then it is also an incorrect execution. More generally, hb related events must not be in conflict. So we require cf;hb to be irreflexive.

If a read is justified by a write it conflicts, then it is also an incorrect execution. More generally, hb related events must not be in conflict. So we require cf;hb to be irreflexive.

As far as the model in the paper goes of LLVM, reads are either non-atomic, acquire or SC. In C11, it is required that non-atomic read values only come from writes that happen before it. Whereas the release and SC reads, while forming a reads-from relation, also imply a happens-before relation. Since LLVM closely resembles (except for the Data Race condition) C11 model, the above irreflexivity condition will suffice.

Non-Conflicting Justifications

If a program has reads justified by writes which conflict, then even such executions are incorrect. More specifically, we want to disallow cases reminiscent of Out-of-Thin-Air behaviors. For this, we disallow writes to justify non-conflicting hb related reads. The authors frame this requirement as $rf; hb^{-1}; rf^{-1}; cf$ to be irreflexive.

If a program has reads justified by writes which conflict, then even such executions are incorrect. More specifically, we want to disallow cases reminiscent of Out-of-Thin-Air behaviors. For this, we disallow writes to justify non-conflicting hb related reads. The authors frame this requirement as $rf; hb^{-1}; rf^{-1}; cf$ to be irreflexive.

The intuition behind the above irreflexivity relation is as follows:

- We want to ensure that conflicting writes do not justify non-conflicting hb related reads.
- So firstly we do not want $[W]; cf; [W]$ to be having reads-from relations with non-conflicting reads.
- For this, we first connect the first write with some read that reads from it (rf).
- Then we trace back happens before, reaching another read. $(rf; hb^{-1}; [R])$.
- Then we connect this read to its read-value $(rf; hb^{-1}; rf^{-1})$.
- Now this write should not be in conflict with the initial one, thus giving us $rf; hb^{-1}; rf^{-1}; cf$ to be irreflexive.

Preserving Order of SC Accesses

- Not relying on official C11 model's definition as it is flawed, in that several architecture compilation schemes are unsound.
- Instead, rely on the SC definition in RC11 paper (Lahav et.al.)
- This requires a union of relations to be acyclic.

Formalizing the Concurrency Semantics of an LLVM Fragment

Preserving Order of SC Accesses

- Not relying on official C11 model's definition as it is flawed, in that several architecture compilation schemes are unsound.
- Instead, rely on the SC definition in RC11 paper (Lahav et.al.)
- This requires a union of relations to be acyclic.

To understand intuition on the new SC definition, one would need to read the RC11 paper in detail. Also, I am unsure of why the definition of SC in C11 is flawed. This is also mentioned using counter example in the RC11 paper. So to understand better the definition, one must go into that.

Preserving Order of SC Accesses

Basic definitions:

- Read-before (fr) : $rf^{-1}; wb \backslash [V]$.
- Program location based order (po_{locs}) : $po \backslash locs$.
- Happens-before-SC ($hbsc$) :
  $[SC]; ((hb \cap locs) \cup (po_{|\neq locs}) \cup (po_{|\neq locs}; hb; po_{|\neq locs})); [SC]$

The axiom for preserving SC then requires $(hbsc \cup wb \cup fr); [SC]$ to be acyclic.

Formalizing the Concurrency Semantics of an LLVM Fragment

Preserving Order of SC Accesses

Basic definitions:

- Read-before (fr) : $rf^{-1}; wb \backslash [V]$.
- Program location based order (po_{locs}) : $po \backslash locs$.
- Happens-before-SC ($hbsc$) : $[SC]; ((hb \cap locs) \cup (po_{|\neq locs}) \cup (po_{|\neq locs}; hb; po_{|\neq locs})); [SC]$

The axiom for preserving SC then requires $(hbsc \cup wb \cup fr); [SC]$ to be acyclic.

I am unsure why instead of relying on a modification of the original SC definition $(po \cup rf \cup mo \cup fr)$ acyclic, the authors have, or rather the RC11 paper has adopted to have these new definitions. Maybe there is an obvious connection, which I am unable to decipher at this point of my understanding.

Consistent Event Strucutre and Observable Behavior

To summarize an event structure is consistent if the following hold:

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

Further, an observable behavior of an event structure is the set of last write values written to memory. That is:

$$\mathit{Behavior}(G) = \{(l, v) | \exists e \in G.V.\ v \leq e.\mathit{wval} \ \wedge\ \nexists e'.\ G.\mathit{wb}(e, e')\}$$

Formalizing the Concurrency Semantics of an LLVM Fragment

Consistent Event Strucutre and Observable Behavior

Consistent Event Strucutre and Observable Behavior

To summarize an event structure is consistent if the following hold:

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

Further, an observable behavior of an event structure is the set of last write values written to memory. That is:

$$\mathit{Behavior}(G) = \{(l, v) | \exists e \in G.V.\ v \leq e.\mathit{wval}\ \wedge\ \nexists e'.\ G.\mathit{wb}(e, e')\}$$

Its interesting to see that observable behavior has this definition, in contrast to being the read values a program gives in any execution. This allows you, as Soham mentioned, reason about programs without even having any read accesses in them. This is better.

DRF Theorems

The above model respects the following DRF Guarantees.

- **DRF-RA**: *If under RA Consistency a program has no read-write races, then its LLVM consistent behaviors coincide with RA-Consistent ones.*
- **DRF-OSC**: *If a program is RA race-free under OSC, then OSC and LLVM consistency coincide.*

DRF-RA Proof Elements

Lemma

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

Lemma

Lemma 2 Given a program P and 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.

TODO

Lemma

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

Lemma

Lemma 2 Given a program P and 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.

Proof

DRF-RA, formally stated:

$$[P]_{RA} \wedge \neg RWRace_{RA}(P) \Rightarrow [[P]]_{RA} = [[P]]_{LLVM}.$$

To prove this, we need to show:

$$[[P]]_{RA} \subseteq [[P]]_{LLVM}$$
$$[[P]]_{LLVM} \subseteq [[P]]_{RA}$$

Proof cnt'd

The first case holds trivially, as by definition LLVM model is stronger than RA.
For the second case, divide into the following two cases for every event structure G of P:

$$Cons_{LLVM}(G) \ \wedge \ \neg RWRace_{LLVM}(G).$$
$$Cons_{LLVM}(G) \ \wedge \ RWRace_{LLVM}(G).$$

The first case, if so, by Lemma **??** guarantees that G is also RA-Consistent. The second case, if so, by Lemma **??** guarantees that G is also RA-Consistent and has a read-write race, which contradicts our assumption that P has no read-write races under RA.
Hence proved.

Formalizing the Concurrency Semantics of an LLVM Fragment

└Proof cnt'd

TODO

Proof cnt'd

The first case holds trivially, as by definition LLVM model is stronger than RA.
For the second case, divide into the following two cases for every event structure G of P:

$$Cons_{LLVM}(G) \ \wedge \ \neg RWRace_{LLVM}(G).$$
$$Cons_{LLVM}(G) \ \wedge \ RWRace_{LLVM}(G).$$

The first case, if so, by Lemma **??** guarantees that G is also RA-Consistent. The second case, if so, by Lemma **??** guarantees that G is also RA-Consistent and has a read-write race, which contradicts our assumption that P has no read-write races under RA.
Hence proved.

DRF-OSC Preliminary definitions

- RARace: $\exists a, b \in G.V \ . \ a.loc = b.loc \ \wedge \ \neg(a.ord = SC \wedge b.ord = SC) \ \wedge \ (a \in W \vee b \in W)$.
- Observable Sequential Consistency : SC without the WW-Race rule for constructing event structure. Only wb relation is used to order writes.

DRF-OSC Proof elements

Lemma

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

Lemma

Lemma 4 For every RA-racy LLVM-consistent event structure, there exists an RA-racy OSC-consistent event structure.

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

Lemma
Lemma 4 For every RA-racy LLVM-consistent event structure, there exists an RA-racy OSC-consistent event structure.

Proof of Lemma 3 is not well written in the appendix. Firstly, RWRace is contained in RARace. This means, by Theorem 1, RA-consistent coincides with LLVM consistent one. So our Lemma then requires us to just prove LLVM-consistent without RA-race(since behaviors coincide, it also means orders between events, hence no RARace too) is also OSC-Consistent. This the authors prove by contradiction. If it isn't OSC-consistent, then there is a read which reads from some old write (which also means no other write w' is hb before this read). Two cases for this, both the read and write are SC, which by construction then make it trivially OSC. If both not SC, then they should be in an RARace by definition, but by our assumption this is not true. Hence proved. (poorly written proof but whatever) Lemma 4 is proved by constructing event structure from point where no races exist. The step which causes a race is also OSC-consistent and has an RARace.

Proof Intuition

DRF-OSC formally stated:

$$\forall P.DRF_{OSC}(P) \implies [[P]]_{LLVM} = [[P]]_{RA} = [[P]]_{OSC}.$$

Note that the RARace definition here corresponds to the same function DRF_{SC} that we use above. Note also that we get $[[P]]_{RA}$ here as its usage in the above lemmas will help us prove this theorem. Proof of the above statement is in two parts:

- $\forall P.DRF_{OSC}(P) \implies [[P]]_{LLVM} = [[P]]_{RA}.$
- $\forall P.DRF_{OSC}(P) \implies [[P]]_{LLVM} = [[P]]_{OSC}.$

Formalizing the Concurrency Semantics of an LLVM Fragment

Proof Intuition

Proof Intuition

DRF-OSC formally stated:

$$\forall P. DRF_{OSC}(P) \implies [[P]]_{LLVM} = [[P]]_{RA} = [[P]]_{OSC}.$$

Note that the RARace definition here corresponds to the same function DRF_{SC} that we use above. Note also that we get $[[P]]_{RA}$ here as its usage in the above lemmas will help us prove this theorem. Proof of the above statement is in two parts:

- $\forall P. DRF_{OSC}(P) \implies [[P]]_{LLVM} = [[P]]_{RA}$.
- $\forall P. DRF_{OSC}(P) \implies [[P]]_{LLVM} = [[P]]_{OSC}$.

The fact that we get RA consistent programs too in the picture is kind of ingenious to me. Is it possible to prove this without involving release acquire? If not, how did the authors figure out that RA consistency as an intermediate step would be the solution? This only tells me that I have yet to learn from my own experience working with proofs.

Proof Intuition cnt'd

The first case can be proven directly using $DRF - RA$ theorem proven before. Because $DRF_S C$ contains $RWDRF_{RA}$.
The second case can be proven because we have

$$\forall G \in P. \neg RARace(G).$$

By Lemma 3, we have then $[[P]]_{RA} = [[P]]_{OSC}$ and by the first case proven, we can infer by transitivity $[[P]]_{LLVM} = [[P]]_{OSC}$.

Proof Intuition cnt'd

The first case can be proven directly using $DRF - RA$ theorem proven before. Because DRF_SC contains $RWDRF_{RA}$.
The second case can be proven because we have

$$\forall G \in P. \neg RARace(G).$$

By Lemma 3, we have then $[[P]]_{RA} = [[P]]_{OSC}$ and by the first case proven, we can infer by transitivity $[[P]]_{LLVM} = [[P]]_{OSC}$.

The above proof is slightly different from the original. The actual proof considers two cases to prove, firstly LLVM subset of OSC and vice versa. I found this a bit confusing, as we already have that RARace does not exist in the program, thus every RA-consistent is also OSC-Consistent as per Lemma 3. Need to investigate their reasoning better (perhaps ask viktor or soham?).

From C11 to Proposed LLVM model

Since the semantics are the same except that regarding data races, it suffices to show that undefined behavior of C11 contains that of LLVM read values having value u. So the following:

$$[[P]]_{LLVM} \subseteq [[P]]_{C11}.$$

is trivial as undefined behavior can contain another undefined behavior.
Note that the above is assuming the SC axioms are the new ones proposed in RC11 for C11.

Validity of Program Transformations

The transformations considered are:

- Reordering of independent accesses.
- Elimination of reads.
- Strengthening of accesses.
- Introduction of reads.

Note also that the following must hold for any valid transformation:

$$P_{src} \mapsto P_{trnsf} \;\Rightarrow\; [[P]]_{trsf} \subseteq [[P]]_{src}.$$

Reordering

The proof for this is done using induction and constructing event structures for both programs. Firstly, note that because we have multiple event structures, we have to reason about the soundness in all of them. For this, firstly, we set up event structure such that the sets of one-to-one po relations for both programs are same. Then, we follow for each a different path (based on the reordering) and then collapse the event set to represent the desired event structure. The crux is that there will always be a way to collapse to find an event structure for both programs representing the same behavior.

The proof for this is done using induction and constructing event structures for both programs. Firstly, note that because we have multiple event structures, we have to reason about the soundness in all of them. For this, firstly, we set up event structure such that the sets of one-to-one po relations for both programs are same. Then, we follow for each a different path (based on the reordering) and then collapse the event set to represent the desired event structure. The crux is that there will always be a way to collapse to find an event structure for both programs representing the same behavior.

This can be done using our thesis idea, where we ensure that equality of sets remain (po or hb). Because of this, the behaviors in the transformed programs will be contained in that of the source. The axioms/ consistency rules will ensure that read values either come from racy writes or one that respects coherence. Since we add more hb relations by reordering, the set of behaviors will trivially be a subset of the source program.

Elimination

Three forms of elimination are considered:

- Overwritten Write (OW) - $x = 1; x = 2; \mapsto x = 2;$. -
- Read After Write (RAW) - $x = 1; a = x; \mapsto x = 1;$.
- Read After Read (RAR) - $a = x; b = x; \mapsto a = x;$

Note that the eliminated event in the above transformation is only those that are non-atomic. The proof for each of this transformation is done by considering one single event structure step in target program to be a two event addition step in the source program. This shows that behaviors in transformed program can be reproduced in the source.

Strengthening

This involves modifying the type of the access such that for the current type o and modified type o', the latter is stronger ($o \subset o'$). The proof for this is straightforward as LLVM model respects monotonicity, which requires that:

$$P \Longrightarrow P'_{strong} \;\Rightarrow\; [[P']] \subseteq [[P]].$$

This involves modifying the type of the access such that for the current type o and modified type o', the latter is stronger ($o \subset o'$). The proof for this is straightforward as LLVM model respects monotonicity, which requires that:

$$P \Longrightarrow P'_{strong} \Rightarrow [[P']] \subseteq [[P]].$$

The above proof can be done by showing that any event structure consistent in P' has an equivalent event structure in P. The strengthening of accesses would mean at teh very least, non-atomic to be release/acquire/SC. In either case, we will have more hb relations, which inevitably means lesser behaviors and they are included in the source program's set of behaviors.

Introduction

Here, only load introduction is considered. Note that in C11, such a transformation is unsound as it may lead to data races, thus giving a program potentially undefined behavior. However, in LLVM such a transformation is frequently performed.

From LLVM to x86-TSO

The mapping from LLVM model proposed to x86-TSO is proven to be sound. This is implied directly due to DRF-RA based results. The proof intuition is as follows:

- We know that for every LLVM racy execution, we have a racy execution in RA.
- We also know that for every LLVM program with no races, it implies program in RA has no races.
- From monotonicity property, the mapping of accesses from LLVM to x86-TSO makes all weaker accesses to either release or acquire (SC will be followed by an MFENCE).
- Now since all accesses are atomic, we have no races. Thus by DRF-RA, the behaviors under LLVM coincide with RA.
- Using existing result of RA to x86-TSO mapping being correct, the proof follows by simply transitivity.

Formalizing the Concurrency Semantics of an LLVM Fragment

From LLVM to x86-TSO

From LLVM to x86-TSO

The mapping from LLVM model proposed to x86-TSO is proven to be sound. This is implied directly due to DRF-RA based results. The proof intuition is as follows:

- We know that for every LLVM racy execution, we have a racy execution in RA.
- We also know that for every LLVM program with no races, it implies program in RA has no races.
- From monotonicity property, the mapping of accesses from LLVM to x86-TSO makes all weaker accesses to either release or acquire (SC will be followed by an MFENCE).
- Now since all accesses are atomic, we have no races. Thus by DRF-RA, the behaviors under LLVM coincide with RA.
- Using existing result of RA to x86-TSO mapping being correct, the proof follows by simply transitivity.

The proof of mapping from RA to TSO is similar to the proof we have written for TSO to SC. Proof by contradiction is the trivial way to do it.

Thank you

Questions?