Formalizing the Concurrency Semantics of an LLVM Fragment

Soham Chakraborty, Viktor Vafeiadis

Max Planck Institute for Software Systems (MPI-SWS)

CGO 2017
LLVM Compilation

C/C++ → frontend → IR

IR: opt → codegen

LLVM

x86 → Power

Result: Lack of understanding of the correctness of the transformations
LLVM Concurrency Compilation

C11 \rightarrow \text{frontend} \rightarrow \text{IR} \xrightarrow{\text{opt}} \text{codegen} \rightarrow \text{x86} \rightarrow \text{Power}
LLVM Concurrency Compilation

- C11 frontend
- IR opt codegen
- x86 Power

formalized formalized
Correctness of the transformations is unclear
Limitation of LLVM Informal Concurrency

too many opt.  ← LLVM compiler ← too few opt.

bugs

no elimination, reordering of atomics

Valid opt is removed by over-restriction in bug fix
This Work

Formalized fragment of LLVM concurrency (except monotonic/relaxed accesses and fences)

Proved correctness of transformations
Informal text in *Language Reference Manual*

Frequent references to C11 concurrency

- "*This model is inspired by the C++0x memory model.*"
- "*These semantics are borrowed from Java and C++0x, but are somewhat more colloquial.*"
- "*This is intended to match shared variables in C/C++ ...*"
- ...
Why not adopt C11 concurrency?

Subtle differences
- A program has write-read race on non-atomics
  - C11: the behavior of the program is *undefined*
  - LLVM: *defined* behavior;

    * racy read returns *undef* \((u)\)

\[
X = 1; \quad \text{if}(X) \quad t = 4; \quad \text{else} \quad t = 4;
\]

\[
t \neq 4 ? : \quad \text{C11 } \checkmark \quad \text{LLVM } \times
\]

- Set of allowed optimizations are different
- Program

\[ \text{int } X = 0, Y = 0; \]
\[ a = X; \quad b = Y; \]
\[ Y = 1; \quad X = 1; \]

- Event Structure

\[ WX0 \]
\[ \text{read-from} \]
\[ RXu \sim RX0 \]
\[ \text{program-order} \]
\[ WY0 \]
\[ \text{conflict relation} \]
\[ RY0 \sim Ryu \]
\[ WY1 \]
\[ WX1 \]
Event Structure Construction

```plaintext
int X = 0, Y = 0;

a = X;          |  b = Y;
Y = 1;          |  X = 1;

WX0
  ↓ program-order
WY0
```
int X = 0, Y = 0;
a = X; \quad b = Y;
Y = 1; \quad X = 1;

read-from
WX0

<table>
<thead>
<tr>
<th>program-order</th>
</tr>
</thead>
<tbody>
<tr>
<td>WY0</td>
</tr>
</tbody>
</table>

RX0
Event Structure Construction

\[
\begin{align*}
\text{int } & \ X = 0, \ Y = 0; \\
\text{a} & = X; \quad \text{b} = Y; \\
\text{Y} & = 1; \quad \text{X} = 1;
\end{align*}
\]

\[
\begin{array}{c}
\text{WX0} \\
\text{RY0} \\
\text{RX0} \\
\text{WY0}
\end{array}
\]

read-from

\[
\text{program-order}
\]

9
int X = 0, Y = 0;
a = X;  b = Y;
Y = 1;  X = 1;

R
\[\sim\]

\text{read-from}
WX0
\text{program-order}

RX0
\rightarrow
WY1

WY0
\rightarrow
RY0

WX1
int X = 0, Y = 0;

a = X;  b = Y;

Y = 1;  X = 1;

read-from

program-order

RX0

WX0

RY0

WY0

WY1

WX1

RACE
int X = 0, Y = 0;

a = X;   b = Y;

Y = 1;   X = 1;

RX0 ↓ WY0 ↓ RX1

WX0 ↓ program-order ↓ WY1

RY0 ∼ RYu

reach relation

read-from
Event Structure Construction

\[
\begin{align*}
\text{int } & X = 0, Y = 0; \\
\text{a } & = X; \quad \text{b } = Y; \\
Y & = 1; \quad X = 1;
\end{align*}
\]
int $X = 0$, $Y = 0$;

$a = X$; \quad b = Y$;

$Y = 1$; \quad X = 1$;
Program Behavior

```
int X = 0, Y = 0;
a = X;  \parallel  b = Y;
Y = 1;  \parallel  X = 1;
a = b = 1  \checkmark
```
int X = 0, Y = 0;

\[ \begin{align*}
& a = X; \quad \mathbin{\|} \quad b = Y; \\
& Y = 1; \quad \mathbin{\|} \quad X = 1; \\
& a = b = 1 \; ? \; \checkmark 
\end{align*} \]

\[ \begin{align*}
\text{int } X &= 0, Y = 0; \\
( a &= X; \quad \mathbin{\|} \quad b = Y; ) \quad \sim \quad Y = 1; \quad \mathbin{\|} \quad X = 1; \\
Y &= 1; \quad \mathbin{\|} \quad X = 1; \\
a &= X; \quad \mathbin{\|} \quad b = Y; 
\end{align*} \]
Execution from Event Structure

```plaintext
int X = 0, Y = 0;
a = X;  ||  b = Y;
Y = 1;  ||  X = 1;

W_X_0  \downarrow  W_Y_0
\quad R_X_u \sim R_X_0  \quad R_Y_0 \sim R_Y_u
\quad W_Y_1  \quad W_X_1
```

11
Execution from Event Structure

```plaintext
int X = 0, Y = 0;

a = X;  b = Y;

Y = 1;  X = 1;
```

Diagram:

```
WX0

/   \
|   |
WY0

R Xu ~ RX0

W Y1

RX0 ~ R Xu

W Y1

RY0 ~ R Y u

W X1

RX0 ~ R X u

W Y1

WX1
```

11
Execution from Event Structure

\[\text{int } X = 0, Y = 0;\]
\[a = X; \quad b = Y;\]
\[Y = 1; \quad X = 1;\]

\[\text{a} = \text{u} = 1, \quad b = \text{u} = 1\]
Aspects of Event Structure

- Proposed formalization handles
  - Memory operations: load, store, CAS
  - Memory orders: non-atomic, acquire, release, acquire_release, sequentially consistent (SC)

- Preserves *consistency* at each construction step

- Multiple consistent event structures per program
Behavior($P_{tgt}$) $\subseteq$ Behavior($P_{src}$)

Behavior. final values observed in each location
Transformation Correctness

Behavior($P_{tgt}$) ⊆ Behavior($P_{src}$)

Behavior. final values observed in each location

⇑

Behavior($G_{tgt}$) ⊆ Behavior($G_{src}$)
LLVM performs these eliminations

Adjacent read after read/write elimination
- \( a = X_o; b = X_{na}; \leadsto a = X_o; b = a; \)
- \( X_o = v; b = X_{na}; \leadsto X_o = v; b = v; \)

Adjacent overwritten write elimination
- \( X_{na} = v'; X_{na} = v; \leadsto X_{na} = v; \)

Non-adjacent overwritten write elimination
- \( X_{na} = v'; C; X_{na} = v; \leadsto C; X_{na} = v; \)
  where rel-acq-pair \( \not\in C \)
LLVM does NOT perform these eliminations

Adjacent read after read/write elimination
- \( a = X_{acq}; b = X_{acq}; \sim \Rightarrow a = X_{acq}; b = a; \)
- \( a = X_{sc}; b = X_{(acq|sc)}; \sim \Rightarrow a = X_{sc}; b = a; \)
- \( X_{rel} = v; b = X_{acq}; \sim \Rightarrow X_{rel} = v; b = v; \)
- \( X_{sc} = v; b = X_{(acq|sc)}; \sim \Rightarrow X_{sc} = v; b = v; \)

Adjacent overwritten write elimination
- \( X_{rel} = v'; X_{rel} = v; \sim \Rightarrow X_{rel} = v; \)
- \( X_{(rel|sc)} = v'; X_{sc} = v; \sim \Rightarrow X_{sc} = v; \)
LLVM does NOT perform these eliminations

Adjacent read after read/write elimination
- \( a = X_{\text{acq}}; b = X_{\text{acq}}; \sim \rightarrow a = X_{\text{acq}}; b = a; \)
- \( a = X_{\text{sc}}; b = X_{(\text{acq}\mid\text{sc})}; \sim \rightarrow a = X_{\text{sc}}; b = a; \)
- \( X_{\text{rel}} = v; b = X_{\text{acq}}; \sim \rightarrow X_{\text{rel}} = v; b = v; \)
- \( X_{\text{sc}} = v; b = X_{(\text{acq}\mid\text{sc})}; \sim \rightarrow X_{\text{sc}} = v; b = v; \)

Adjacent overwritten write elimination
- \( X_{\text{rel}} = v'; X_{\text{rel}} = v; \sim \rightarrow X_{\text{rel}} = v; \)
- \( X_{(\text{rel}\mid\text{sc})} = v'; X_{\text{sc}} = v; \sim \rightarrow X_{\text{sc}} = v; \)

Non-adjacent read after write elimination
- \( X_{\text{na}} = v; C; a = X_{\text{na}}; \sim \rightarrow X_{\text{na}} = v; C; a = v; \)
  where rel-acq-pair \( \notin C \)
LLVM Reorderings

LLVM performs(✓) these reorderings

\[ a; b \sim b; a \]

| \( \downarrow a \ \backslash \ b \rightarrow \) | \((\text{St}|\text{Ld})_{na}\) | \text{St}_{rel} | \text{Ld}_{acq} | \text{Ld}_{sc} | \text{U}_{(\text{acq}\_\text{rel}|\text{sc})} |
|-----------------|-----------------|------------|-------------|------------|-----------------|
| \((\text{St}|\text{Ld})_{na}\) | ✓ | - | ✓ | ✓ | - |
| \text{St}_{rel} | ✓ | - | - | - | - |
| \text{St}_{sc} | ✓ | - | - | - | - |
| \text{Ld}_{acq} | - | - | - | - | - |
| \text{U}_{(\text{acq}\_\text{rel}|\text{sc})} | - | - | - | - | - |

\[ X_{rel} = v; \ Y_{na} = v'; \sim \ Y_{na} = v'; \ X_{rel} = v; \ \checkmark \]
LLVM restricts (×) these reorderings

\( a; b \sim b; a \)

| \( \downarrow a \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ b \rightarrow \) | \( (\text{St}|\text{Ld})_{\text{na}} \) | \( \text{St}_{\text{rel}} \) | \( \text{Ld}_{\text{acq}} \) | \( \text{Ld}_{\text{sc}} \) | \( U_{(\text{acq}_{\text{rel}}|\text{sc})} \) |
|----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| \( (\text{St}|\text{Ld})_{\text{na}} \) | \( \checkmark \) | \( \times \) | \( \checkmark \) | \( \checkmark \) | \( \times \) |
| \( \text{St}_{\text{rel}} \) | \( \checkmark \) | \( \times \) | \( - \) | \( - \) | \( \times \) |
| \( \text{St}_{\text{sc}} \) | \( \checkmark \) | \( \times \) | \( - \) | \( \times \) | \( \times \) |
| \( \text{Ld}_{\text{acq}} \) | \( \times \) | \( \times \) | \( \times \) | \( \times \) | \( \times \) |
| \( U_{(\text{acq}_{\text{rel}}|\text{sc})} \) | \( \times \) | \( \times \) | \( \times \) | \( \times \) | \( \times \) |

\( Y_{\text{na}} = v'; \ X_{\text{rel}} = v; \sim \) \( X_{\text{rel}} = v; \ Y_{\text{na}} = v'; \times \)
LLVM does NOT perform these reorderings

\[ a; b \leadsto b; a \]

<table>
<thead>
<tr>
<th>( \downarrow a \ \backslash \ b \rightarrow )</th>
<th>((\text{St} \mid \text{Ld})_{\text{na}})</th>
<th>(\text{St}_{\text{rel}})</th>
<th>(\text{Ld}_{\text{acq}})</th>
<th>(\text{Ld}_{\text{sc}})</th>
<th>(U_{(\text{acq_rel}\mid\text{sc})})</th>
</tr>
</thead>
<tbody>
<tr>
<td>((\text{St} \mid \text{Ld})_{\text{na}})</td>
<td>\checkmark</td>
<td>\xmark</td>
<td>\checkmark</td>
<td>\checkmark</td>
<td>\xmark</td>
</tr>
<tr>
<td>(\text{St}_{\text{rel}})</td>
<td>\checkmark</td>
<td>\xmark</td>
<td>\checkmark</td>
<td>\checkmark</td>
<td>\xmark</td>
</tr>
<tr>
<td>(\text{St}_{\text{sc}})</td>
<td>\checkmark</td>
<td>\xmark</td>
<td>\checkmark</td>
<td>\xmark</td>
<td>\xmark</td>
</tr>
<tr>
<td>(\text{Ld}_{\text{acq}})</td>
<td>\xmark</td>
<td>\xmark</td>
<td>\xmark</td>
<td>\xmark</td>
<td>\xmark</td>
</tr>
<tr>
<td>(U_{(\text{acq_rel}\mid\text{sc})})</td>
<td>\xmark</td>
<td>\xmark</td>
<td>\xmark</td>
<td>\xmark</td>
<td>\xmark</td>
</tr>
</tbody>
</table>

\[ X_{\text{rel}} = v; \ t = Y_{\text{acq}}; \leadsto t = Y_{\text{acq}}; X_{\text{rel}} = v; \checkmark \]
- LLVM has operations (Ld/St/CAS) and memory orders (na/rel/acq/acq_rel/SC) similar to C11.

- LLVM model is stronger than C11.
(LLVM $\sim x86$/Power) = (C11 $\sim x86$/Power)

Proved correctness of these mappings

- LLVM to SC
- LLVM to SPower

Ensure correctness of LLVM $\sim x86$/Power
(results from Lahav & Vafeiadis. FM’16)
What's More in The Paper

Event structure construction rules

\[
\begin{align*}
\forall e'' \in V. \ e'.id & \neq e'' .id \\
\forall e''. \ po(e, e'') & \implies e'.lab \neq e''.lab
\end{align*}
\]

(BASIC)

\[
\langle V, po, rf \rangle \xrightarrow{e'} \langle V \cup \{e'\}, (po \cup \{(e, e')\})^+, rf \rangle
\]

\[
\frac{WWrace(G)}{G \xrightarrow{e'} G'} \quad (WW-RACE) \
\frac{e' \notin \mathcal{R}}{G \xrightarrow{e'} G'} \quad (NON-READ)
\]

(R-UNINIT)

\[
\frac{G \xrightarrow{e'} G'' \quad e' \in \mathcal{R} \quad e'.rval = u \quad \neg G''.hbW(e')}{G \xrightarrow{e'} G''} \quad (R-RACE)
\]

(R-NORACE)
What’s More in The Paper

Event structure construction rules

Consistency constraints

\[
isCons(G) \triangleq \text{irreflexive}(wb) \land \text{irreflexive}(cf; hb) \\
\land \text{irreflexive}(rf; hb^{-1}; rf^{-1}; cf) \\
\land \text{acyclic}((hbsc \cup wb \cup fr); [SC])
\]
What's More in The Paper

Event structure construction rules

Consistency constraints

Data race freedom (DRF) theorems
What’s More in The Paper

Event structure construction rules

Consistency constraints

Data race freedom (DRF) theorems

More transformations
  - Speculative load
  - Strengthening memory order of accesses

Proofs: http://plv.mpi-sws.org/llvmcs/
What’s More in The Paper

Event structure construction rules

Consistency constraints

Data race freedom (DRF) theorems

More transformations
  - Speculative load
  - Strengthening memory order of accesses

Proofs: http://plv.mpi-sws.org/llvmcs/
Conclusions & Future Directions

- Contributions

  - clang ✓
  - IR ✓
  - opt ✓
  - codegen ✓
  - x86 ✓
  - C11 ✓
  - codegen ✓
  - Power ✓

Formalized ✓
DRF Theorems ✓

- Future: extend the LLVM concurrency model
  - With relaxed accesses and fences
  - Prove/disprove more optimizations
  - Mechanize the formalization

Thank You!
Examples

\begin{align*}
\text{int } X &= 0, Y = 0; \\
a &= X; & b &= Y; \\
Y &= 1; & X &= 1; \\
a &= b = 1 & \checkmark
\end{align*}

\text{(LB)}

\begin{align*}
\text{int } X &= 0, Y = 0; \\
a &= X; & b &= Y; \\
\text{if}(a == 1) & \quad \text{if}(b == 1) \\
Y &= 1; & X &= 1; \\
a &= b = 1 & \times
\end{align*}

\text{(CYC)}
LLVM vs C11

**LLVM**

```
[WX0, WY0]
RXu ~ RX0

WY1

RY0 ~ RYu

WX1
```

**C11**

```
[WX0, WY0]
RX0 ~ RX1

WY1

RY0 ~ RY1

WX1
```

(LB & CYC)
LLVM performs speculative load

\[ \begin{align*}
X &= 1; \\
\text{if}(\text{flag})\{ \\
    a &= X; \\
\} \\
\sim &\quad X = 1; \\
\begin{align*}
    t &= X; &// \text{undef} \\
\text{if}(\text{flag})\{ \\
    a &= t; \\
\} 
\end{align*}
\]