Formalizing the Concurrency Semantics of an LLVM Fragment

Soham Chakraborty, Viktor Vafeiadis

Max Planck Institute for Software Systems (MPI-SWS)

EuroLLVM 2017
LLVM Compilation

C/C++ -> frontend -> IR -> codegen -> x86

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

- C11
- frontend
- IR
- opt
- codegen
- Power
- x86

Correctness of the transformations is unclear
LLVM Concurrency Compilation

- C11 to frontend
- IR (opt + codegen) to Power
- x86

Correctness of the transformations is unclear.

formalized

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

too many opt. \( \xrightarrow{\text{LLVM compiler}} \) 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

Verified correctness of transformations

Validated LLVM opt-phase 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 \texttt{undef(u)}
Why not adopt C11 concurrency?

Subtle differences
- A program has write-read race on non-atomics
  - C11: the behavior of the program is \textit{undefined}
  - LLVM: \textit{defined} behavior;
    racy read returns \texttt{undef(u)}

\begin{align*}
X &= 1; \\
\text{if}(X) & \quad t = 4; \\
\text{else} & \quad t = 4;
\end{align*}
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;
if (X)
  t = 4;
else
  t = 4;
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 \texttt{undef}(u)

\[
\begin{align*}
X &= 1; \\
\text{if}(X) & \quad t = 4; \\
\text{else} & \quad t = 4; \\
t & \neq 4 ?
\end{align*}
\]
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 $\text{undef}(u)$

\[
X = 1; \\
\begin{align*}
\text{if}(X) & \quad t = 4; \\
\text{else} & \quad t = 4;
\end{align*}
\]

\[t \neq 4 \quad \text{C11} \checkmark\]
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)*

\[
\begin{align*}
X &= 1; \\
\text{if}(X) & \quad t = 4; \\
\text{else} & \quad t = 4;
\end{align*}
\]

\[t \neq 4 \ ? \quad \text{C11} \ \checkmark \quad \text{LLVM} \ \xmark\]
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)`

\[
\begin{align*}
X &= 1; & \text{if}(X) & t = 4; \\
\text{else} & t = 4;
\end{align*}
\]

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

- Set of allowed optimizations are different
C11 vs LLVM

Context:

\[
\begin{bmatrix}
X = 1; & \parallel \\
\end{bmatrix}
\]

if (flag) {
\[
\begin{align*}
a &= X; \\
\end{align*}
\]
}

\[t = X;\]
if (flag) {
\[
\begin{align*}
a &= t; \\
\end{align*}
\]
}

C11 \xmark LLVM \checkmark
C11 vs LLVM

Context:
\[
\begin{bmatrix}
X = 1; & \|
\end{bmatrix}
\]
\[
\text{if}(\text{flag})\{
  a = X ;
\}
\]
\[t = X ;
\text{if}(\text{flag})\{
  a = t ;
\}
\]

C11 ✗ LLVM ✓

Context:
\[
\begin{bmatrix}
X = 4; & \|
Y_{rel} = 1 ;
\end{bmatrix}
\]
\[
t_1 = X ;
\text{if}(Y_{acq})\{
  t_2 = X ;
\}
\]
\[t_1 = X ;
\text{if}(Y_{acq})\{
  t_2 = t_1 ;
\}
\]

C11 ✓ LLVM ✗
Formalization of LLVM concurrency

Verified correctness of transformations

Validated LLVM opt-phase transformations
Example

```
int X = 0, Y = 0;
a = X; || b = Y;
Y = 1; || X = 1;
Is a == b == 1 possible?
```
Example

\[
\begin{align*}
\text{int } X &= 0, Y = 0; \\
\text{a }\Rightarrow X; &\quad | \quad \text{b }\Rightarrow Y; \\
Y &= 1; &\quad | \quad X &= 1; \\
\text{Is}\ a &== b == 1 \text{ possible? } \checkmark
\end{align*}
\]
Event Structure Construction

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

```c
da = X; || b = Y;
```

```c
Y = 1; || X = 1;
```

![Diagram]

- **WX0**
- **WY0**
\begin{verbatim}
int X = 0, Y = 0;
a = X;  \quad b = Y;
Y = 1;  \quad X = 1;
\end{verbatim}
\texttt{int X = 0, Y = 0;}
\texttt{a = X;} \quad \| \quad \texttt{b = Y;}
\texttt{Y = 1;} \quad \| \quad \texttt{X = 1;}

\begin{center}
\begin{tikzpicture}[dashed, thick, ->, >=latex]

\node (RX0) at (0, 0) {\texttt{RX0}};
\node (WX0) at (2, 2) {\texttt{WX0}};
\node (WY0) at (2, 0) {\texttt{WY0}};
\node (RY0) at (4, 0) {\texttt{RY0}};

\draw[->] (WX0) -- (WY0) node [midway, above] {\texttt{program-order}};
\draw[->] (RX0) -- (WX0) node [midway, below, sloped] {\texttt{read-from}};
\draw[->] (WY0) -- (RY0);
\end{tikzpicture}
\end{center}
**Event Structure Construction**

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

![Event structure diagram](image-url)
int X = 0, Y = 0;
a = X;    b = Y;
Y = 1;   X = 1;
Event Structure Construction

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

Diagram:

- `WX0` connected by `program-order` to `WY0`
- `read-from` from `RX0` to `WY0`
- `RACE` from `WX0` to `WX1`
- `conflict relation` from `RY0` to `RYub`
int X = 0, Y = 0;
a = X;
b = Y;
Y = 1;
X = 1;
Event Structure Construction

\[ int \; X = 0, \; Y = 0; \]
\[ a = X; \quad \| \quad b = Y; \]
\[ Y = 1; \quad \| \quad X = 1; \]
Example

int X = 0, Y = 0;
a = X; \quad b = Y;
Y = 1; \quad X = 1;

Is \( a == b == 1 \) possible?  

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

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

WX0

RY0 ∼ RYu

RX0

RXu_a ∼ RX0

WX1  WY1

WY1  WX1
int X = 0, Y = 0;
a = X;  \parallel  b = Y;
Y = 1;  \parallel  X = 1;

\[
\begin{align*}
WX0 \\
\downarrow \\
WY0 \\
RXu_a \sim RX0 \\
WY1 \\
RY0 \sim RYu_b \\
WX1 \\
\end{align*}
\]
int X = 0, Y = 0;

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

\[ a = u_a = 1, \quad b = u_b = 1 \]
Proposed Formalization Handles

- Memory operations:
  - load
  - store
  - compare_and_swap (CAS)

- Memory orders:
  - non-atomic (na)
  - acquire (acq)
  - release (rel)
  - acquire_release (acq_rel)
  - sequentially consistent (sc)
Formalized fragment of LLVM concurrency

Verified correctness of transformations

- Elimination
- Reordering
- Mappings (C11 $\leadsto$ LLVM $\leadsto$ X86/Power)

Validated LLVM opt-phase transformations
Transformation Correctness

Behavior($P_{tgt}$) $\subseteq$ Behavior($P_{src}$)

Behavior: final values observed in each location
Transformation Correctness

Behavior($P_{tgt}$) \subseteq Behavior($P_{src}$)

Behavior: final values observed in each location

Behavior($G_{tgt}$) \subseteq Behavior($G_{src}$)
Elimination Optimizations

Adjacent read after read/write elimination

- \( a = X_o; b = X_{na}; \sim \rightarrow a = X_o; b = a; \)
- \( X_o = v; b = X_{na}; \sim \rightarrow X_o = v; b = v; \)

Adjacent overwritten write elimination

- \( X_{na} = v'; X_{na} = v; \sim \rightarrow X_{na} = v; \)

Non-adjacent overwritten write elimination

- \( X_{na} = v'; C; X_{na} = v; \sim \rightarrow C; X_{na} = v; \)
  where rel-acq-pair \( \notin C \) and access(\( X \)) \( \notin C \)

**LLVM performs these eliminations**
Elimination Optimizations

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

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

Non-adjacent overwritten write elimination
- \(X_{na} = v'; X_{na} = v; \sim \Rightarrow C; X_{na} = v;\)
- \(\text{rel-acq-pair} \notin C \text{ and } access(X) \notin C\)

LLVM performs these eliminations
Adjacent read after read/write elimination

- \( a = X_{acq}; b = X_{acq}; \leadsto a = X_{acq}; b = a; \)
- \( a = X_{sc}; b = X_{(acq|sc)}; \leadsto a = X_{sc}; b = a; \)
- \( X_{rel} = v; b = X_{acq}; \leadsto X_{rel} = v; b = v; \)
- \( X_{sc} = v; b = X_{(acq|sc)}; \leadsto X_{sc} = v; b = v; \)

Adjacent overwritten write elimination

- \( X_{rel} = v'; X_{rel} = v; \leadsto X_{rel} = v; \)
- \( X_{(rel|sc)} = v'; X_{sc} = v; \leadsto X_{sc} = v; \)

LLVM does NOT perform these eliminations
Also Proved...

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

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

LLVM does NOT perform these eliminations

Non-adjacent read after write elimination
- \( X_{na} = v; C; a = X_{na}; \leadsto X_{na} = v; C; a = v; \)

where rel-acq-pair \( \notin C \) and \( access(X) \notin C \)
Also Proved...

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

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

LLVM does NOT perform these eliminations

Non-adjacent read after write elimination
- \( X_{na} = v; C; a = X_{na}; \sim \rightarrow X_{na} = v; C; a = v; \)
  where rel-acq-pair \( \notin C \) and access(\( X \)) \( \notin C \)
Verifying Transformations

Formalized fragment of LLVM concurrency

**Verified correctness of transformations**

- Elimination
- **Reordering** \((a; b \leadsto b; a)\)
- Mappings (C11 \(\leadsto\) LLVM \(\leadsto\) X86/Power)

Validated LLVM opt-phase transformations
LLVM Reorderings

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

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

\[ X_{rel} = v; \; Y_{na} = v'; \leadsto Y_{na} = v'; \; X_{rel} = v; \; \checkmark \]

LLVM performs(✓) these reorderings.
LLVM Reorderings

$$a; b \leadsto b; a$$

| ↓ $a \ \downarrow b \rightarrow$ | $(St|Ld)_{na}$ | $St_{rel}$ | $Ld_{acq}$ | $Ld_{sc}$ | $U_{(acq\_rel|sc)}$ |
|--------------------------------|----------------|------------|-----------|-----------|------------------|
| $(St|Ld)_{na}$                | ✓              | ×          | ✓         | ✓         | ×                |
| $St_{rel}$                    | ✓              | ×          | -         | -         | ×                |
| $St_{sc}$                     | ✓              | ×          | -         | ×         | ×                |
| $Ld_{acq}$                    | ×              | ×          | ×         | ×         | ×                |
| $U_{(acq\_rel|sc)}$           | ×              | ×          | ×         | ×         | ×                |

$$Y_{na} = v'; X_{rel} = v; \leadsto X_{rel} = v; Y_{na} = v'; \times$$

**LLVM restricts (×) these reorderings**
Also Analyzed...

\[ a; b \rightsquigarrow b; a \]

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

\[ X_{\text{rel}} = v; t = Y_{\text{acq}}; \rightsquigarrow t = Y_{\text{acq}}; X_{\text{rel}} = v; \]

LLVM does NOT perform these reorderings
Also Analyzed...

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

| \( \downarrow a \ \backslash \ b \rightarrow \) | (St|Ld)_na | St_rel | Ld_acq | Ld_d | U(acq_rel|sc) |
|---------------------------------|---------|--------|--------|------|-------------|
| (St|Ld)_na                      | ✓       | ✗      | ✓      | ✓    | ✗           |
| St_rel                         | ✓       | ✗      | ✓      | ✓    | ✗           |
| St_sc                          | ✓       | ✓      | ✓      | ✓    | ✗           |
| Ld_acq                         | ✓       | ✓      | ✗      | ✗    | ✗           |
| U(acq_rel|sc)                    | ✗       | ✗      | ✗      | ✗    | ✗           |

\[ X_{rel} = v; t = Y_{acq}; \sim t = Y_{acq}; X_{rel} = v; \checkmark \]

LLVM does NOT perform these reorderings
Formalized fragment of LLVM concurrency

**Verified correctness of transformations**

- Elimination
- Reordering
- **Mappings** (C11 $\leadsto$ LLVM $\leadsto$ X86/Power)

Validated LLVM opt-phase transformations
- 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 has operations (Ld/St/CAS) and memory orders (na/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)
LLVM to Architecture Mapping Correctness

\[(\text{LLVM } \sim \text{x86/Power}) = (\text{C11 } \sim \text{x86/Power})\]

Proved correctness of these mappings:
- LLVM to SC
- LLVM to SPower

Ensuring correctness of LLVM \(\sim\) x86/Power
(results from Lahav & Vafeiadis. FM’16)
Validation

Formalized fragment of LLVM concurrency

Proved correctness of transformations

Validated LLVM opt-phase transformations

$P_{src} \xrightarrow{LLVM} P_{tgt}$ ? Correct : Potential Error
LLVM Validation

\[ P_{src} \xrightarrow{LLVM} P_{tgt} \quad ? \quad \text{Correct : Potential Error} \]

\[ P_{src} \xrightarrow{(R \cup E)^*} P_{tgt} \quad ? \quad \text{Correct : Potential Error} \]

- **R**: Safe reorderings
- **E**: Safe eliminations
\( s_1 = X !A \)
\( s_2 = X !B \)
\( V = 1 !C \)
\( s_4 = Z_{acq} !D \)
\( Y = 1 !E \)
\( Y = 2 !F \)
\textbf{Metadata Based Matching}

\begin{align*}
\checkmark s_1 &= X \!A \\
 s_2 &= X \!B \\
 V &= 1 \!C \\
 s_4 &= Z_{acq} \!D \\
 Y &= 1 \!E \\
 Y &= 2 \!F 
\end{align*}

Check that unmatched accesses are deletable
Check that reorderings are allowed
Metadata Based Matching

✓ $s_1 = X !A$

✗ $s_2 = X !B$

$V = 1 !C$

$s_4 = Z_{acq} !D$

$Y = 1 !E$

$Y = 2 !F$
Metadata Based Matching

\[ \checkmark \ s_1 = X \ !A \]
\[ \times \ s_2 = X \ !B \]
\[ \ V = 1 \ !C \]
\[ \checkmark \ s_4 = Z_{\text{acq}} \ !D \]
\[ \ Y = 1 \ !E \]
\[ \ Y = 2 \ !F \]
Metadata Based Matching

✓ $s_1 = X \ !A$

✗ $s_2 = X \ !B$

$V = 1 \ !C$

✓ $s_4 = Z_{acq} \ !D$

$Y = 1 \ !E$

✓ $Y = 2 \ !F$
Metadata Based Matching

- $s_1 = X \, !A$
- $s_2 = X \, !B$
- $V = 1 \, !C$
- $s_4 = Z_{acq} \, !D$
- $Y = 1 \, !E$
- $Y = 2 \, !F$
Metadata Based Matching

\[
\begin{align*}
\checkmark \quad s_1 &= X \ !A \\
\times \quad s_2 &= X \ !B \\
\checkmark \quad V &= 1 \ !C \\
\checkmark \quad s_4 &= Z_{acq} \ !D \\
\times \quad Y &= 1 \ !E \\
\checkmark \quad Y &= 2 \ !F
\end{align*}
\]
\begin{itemize}
  \item $s_1 = X !A$
  \item $s_2 = X !B$
  \item $V = 1 !C$
  \item $s_4 = Z_{acq} !D$
  \item $Y = 1 !E$
  \item $Y = 2 !F$
  \item $t_1 = X !A$
  \item $t_2 = Z_{acq} !D$
  \item $Y = 2 !F$
  \item $V = 1 !C$
\end{itemize}
Correct

Check that unmatched accesses are deletable
Check that reorderings are allowed
Check that unmatched accesses are deletable
Check that reorderings are allowed
Metadata Based Matching

\[ \begin{align*}
\checkmark \ s_1 &= X \ !A \\
\times \ s_2 &= X \ !B \\
\checkmark \ V &= 1 \ !C \\
\checkmark \ s_4 &= Z_{acq} \ !D \\
\times \ Y &= 1 \ !E \\
\checkmark \ Y &= 2 \ !F
\end{align*} \]

Correct

- Check that unmatched accesses are deletable
- Check that reorderings are allowed
Validation

Formalized fragment of LLVM concurrency

Proved correctness of transformations

Validated LLVM opt-phase transformations

- Generate a test case ($P_{src}$).
- Apply LLVM transformations ($P_{tgt}$).

$P_{src} \xrightarrow{\text{LLVM}} P_{tgt} \ ? \ \text{Correct} : \ \text{Potential Error}$
LLVM Formalization [CGO’17]
- Event structure construction rules
- Consistency constraints
- Data race freedom (DRF) theorems
- Proofs: http://plv.mpi-sws.org/llvmcs/

Translation validation [CGO’16]
- Programs with control flow
- Experimental evaluations
- Artifact: http://plv.mpi-sws.org/validc/
Summary

C11

IR

opt ✓

codegen ✓

codegen ✓

x86

Power

clang ✓

Formalized ✓

DRF Theorems ✓

Validated opt-phase transformations ✓
Future Directions

Extend the LLVM concurrency model
- With relaxed accesses and fences
- Verify more optimizations
- Mechanize the formalization

- Improve the validator
  - Integrate with sequential transformations
  - Handle loops, pointer etc

Thank You!