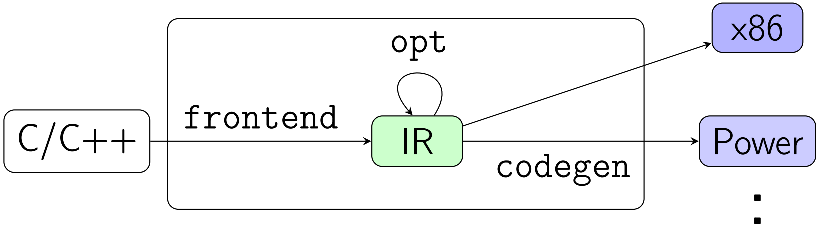
Formalizing the Concurrency Semantics of an LLVM Fragment

Soham Chakraborty, Viktor Vafeiadis

Max Planck Institute for Software Systems (MPI-SWS)

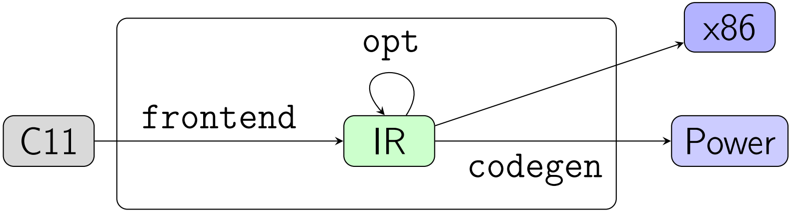
EuroLLVM 2017

LLVM Compilation

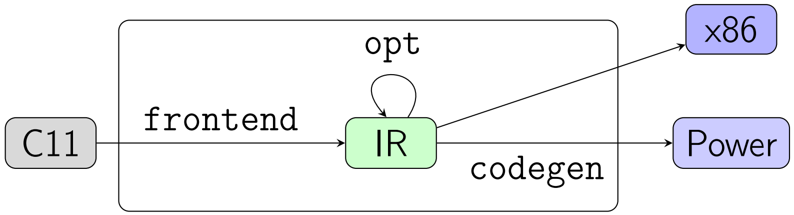


LLVM

LLVM Concurrency Compilation

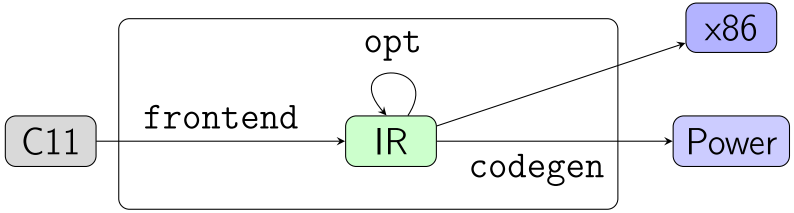


LLVM Concurrency Compilation



formalized **formalized**

LLVM Concurrency Compilation



formalized **informal** **formalized**

Correctness of the transformations is unclear

Limitation of LLVM Informal Concurrency

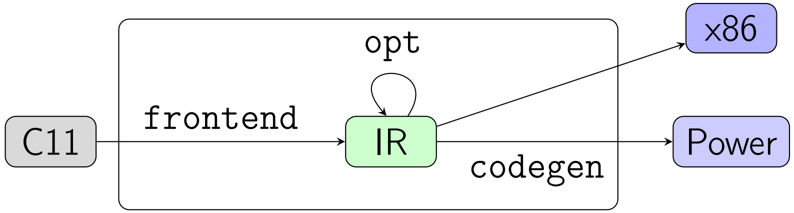
$$\text{too many opt.} \xleftrightarrow{\text{LLVM compiler}} \text{too few opt.}$$

bugs

no elimination, reordering of atomics

Valid opt is removed by over-restriction in bug fix

This Work



formalized informal formalized

Formalized fragment of LLVM concurrency

Verified correctness of transformations

Validated LLVM opt-phase transformations

Informal LLVM Concurrency

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)**

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)**

$$X = 1; \quad \left\| \quad \begin{array}{l} \text{if}(X) \\ \quad t = 4; \\ \text{else} \\ \quad t = 4; \end{array} \right.$$

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)**

$$X = 1; \quad \left\| \quad \begin{array}{l} \text{if}(X) \\ \quad t = 4; \\ \text{else} \\ \quad t = 4; \end{array} \right.$$

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)**

$$X = 1; \;\Big\|\; \begin{array}{l} \text{if}(X) \\ \quad t = 4; \\ \text{else} \\ \quad t = 4; \end{array}$$

$t \neq 4$?

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 **undef(u)**

$$X = 1; \;\Big\|\; \begin{array}{l} \text{if}(X) \\ \quad t = 4; \\ \text{else} \\ \quad t = 4; \end{array}$$

$t \neq 4$? C11 ✓

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)**

$$
X = 1; \quad \Bigg\| \quad \begin{array}{l} \text{if}(X) \\ \quad t = 4; \\ \text{else} \\ \quad t = 4; \end{array}
$$

$t \neq 4$?   C11 ✓   LLVM ✗

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)**

$$X = 1; \quad \Bigg\| \quad \begin{array}{l} \text{if}(X) \\ \quad t = 4; \\ \text{else} \\ \quad t = 4; \end{array}$$

$t \neq 4$? C11 ✓ LLVM ✗

- Set of allowed optimizations are different

C11 vs LLVM

Context:

$$\left[X = 1; \;\; \| \;\; \right]$$

```
if(flag){
    a = X;
}
```

$\rightsquigarrow$

```
t = X;
if(flag){
    a = t;
}
```

C11 ✗ LLVM ✓

C11 vs LLVM

Context:

$$\left[\; X = 1; \;\; \| \;\; \right]$$

$$\begin{array}{l} \text{if}(\mathit{flag})\{ \\ \quad a = X; \\ \} \end{array} \quad \rightsquigarrow \quad \begin{array}{l} t = X; \\ \text{if}(\mathit{flag})\{ \\ \quad a = t; \\ \} \end{array}$$

C11 ✗ LLVM ✓

Context:

$$\left[\; \begin{array}{l} X = 4; \\ Y_{\text{rel}} = 1; \end{array} \;\; \| \;\; \right]$$

$$\begin{array}{l} t_1 = X; \\ \text{if}(Y_{\text{acq}})\{ \\ \quad t_2 = X; \\ \} \end{array} \quad \rightsquigarrow \quad \begin{array}{l} t_1 = X; \\ \text{if}(Y_{\text{acq}})\{ \\ \quad t_2 = t_1; \\ \} \end{array}$$

C11 ✓ LLVM ✗

Formalization

Formalization of LLVM concurrency

Verified correctness of transformations

Validated LLVM opt-phase transformations

Example

$$int\ X = 0, Y = 0;$$

$$\begin{array}{r||l} a = X; & b = Y; \\ Y = 1; & X = 1; \end{array}$$

Is $a == b == 1$ possible?

Example

$$int\ X = 0, Y = 0;$$
$$\begin{array}{r|l} a = X; & b = Y; \\ Y = 1; & X = 1; \end{array}$$

Is $a == b == 1$ possible? ✓

$$int\ X = 0, Y = 0; \qquad\qquad int\ X = 0, Y = 0;$$
$$\left.\begin{array}{r|l} a = X; & b = Y; \\ Y = 1; & X = 1; \end{array}\right. \rightsquigarrow \begin{array}{r|l} Y = 1; & X = 1; \\ a = X; & b = Y; \end{array}$$

Event Structure Construction

$int\ X = 0, Y = 0;$

$a = X;$	‖	$b = Y;$
$Y = 1;$	‖	$X = 1;$

$$\begin{tikzcd}\mathrm{W}X0 \arrow[d, "\text{program-order}"] \\ \mathrm{W}Y0\end{tikzcd}$$

Event Structure Construction

$$\textit{int } X = 0, Y = 0;$$

$$\begin{array}{l||l} a = X; & b = Y; \\ Y = 1; & X = 1; \end{array}$$

$$\begin{tikzcd}
 & \mathrm{W}X0 \arrow[d, "\text{program-order}"] \arrow[ddl, dashed, bend right, "\text{read-from}"'] \\
 & \mathrm{W}Y0 \arrow[dl] \\
\mathrm{R}X0 &
\end{tikzcd}$$

Event Structure Construction

$$\mathit{int}\ X = 0, Y = 0;$$

$$\begin{array}{l|l} a = X; & b = Y; \\ Y = 1; & X = 1; \end{array}$$

$$\begin{tikzcd}
 & \mathrm{W}X0 \arrow[d, "\text{program-order}"] \arrow[dl, dashed, bend right, "\text{read-from}"'] & \\
 & \mathrm{W}Y0 \arrow[dl] \arrow[dr] \arrow[dr, dashed, bend right] & \\
\mathrm{R}X0 & & \mathrm{R}Y0
\end{tikzcd}$$

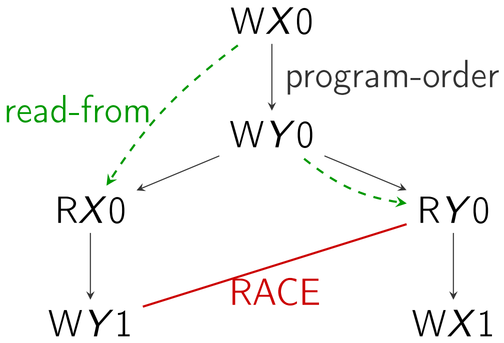
Event Structure Construction

$$\begin{array}{c} \mathit{int}\ X = 0, Y = 0; \\ \begin{array}{l||l} a = X; & b = Y; \\ Y = 1; & X = 1; \end{array} \end{array}$$

$$\begin{tikzcd}
 & \mathrm{W}X0 \arrow[d, "\text{program-order}"] \arrow[dl, dashed, bend right, "\text{read-from}"'] & \\
 & \mathrm{W}Y0 \arrow[dl] \arrow[dr] \arrow[dr, dashed, bend right] & \\
\mathrm{R}X0 \arrow[d] & & \mathrm{R}Y0 \arrow[d] \\
\mathrm{W}Y1 & & \mathrm{W}X1
\end{tikzcd}$$

Event Structure Construction

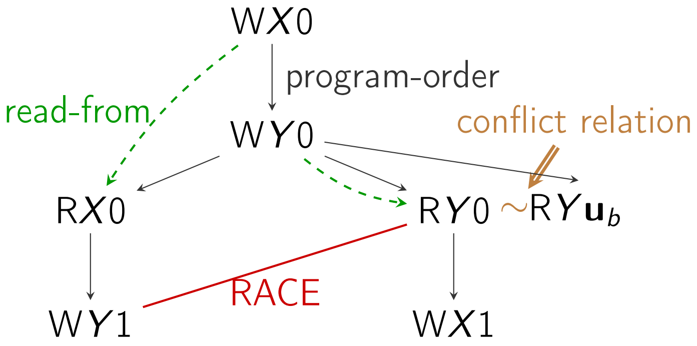
$$\begin{array}{c}
\textit{int}\ X = 0, Y = 0; \\
\begin{array}{l|l}
a = X; & b = Y; \\
Y = 1; & X = 1;
\end{array}
\end{array}$$



Event Structure Construction

$$int\ X = 0, Y = 0;$$

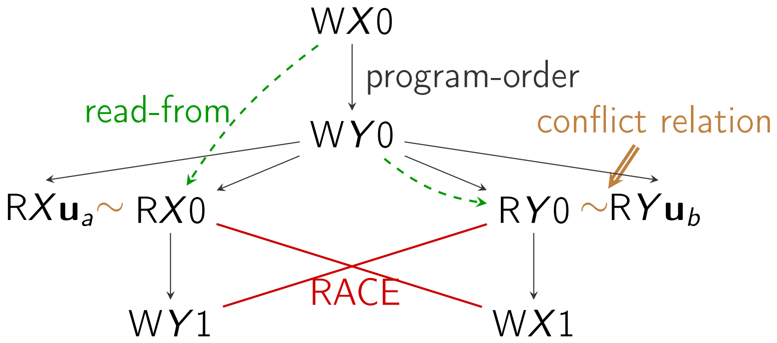
$a = X;$	$\|$	$b = Y;$
$Y = 1;$	$\|$	$X = 1;$



Event Structure Construction

$$int\ X = 0, Y = 0;$$

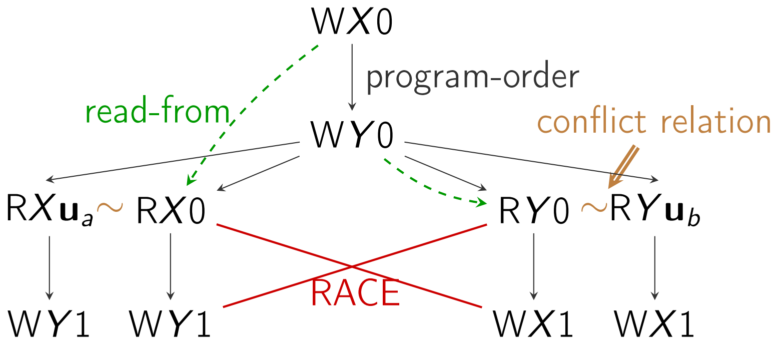
$a = X;$	$\|$	$b = Y;$
$Y = 1;$	$\|$	$X = 1;$



Event Structure Construction

$int\ X = 0, Y = 0;$

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



Example

$$\begin{array}{c} \mathit{int}\ X = 0, Y = 0; \\ \begin{array}{r|l} a = X; & b = Y; \\ Y = 1; & X = 1; \end{array} \end{array}$$

Is $a == b == 1$ possible? ✓

$$\begin{array}{c} \mathit{int}\ X = 0, Y = 0; \\ \left\updownarrow \begin{array}{r|l} a = X; & b = Y; \\ Y = 1; & X = 1; \end{array} \right\updownarrow \end{array} \rightsquigarrow \begin{array}{c} \mathit{int}\ X = 0, Y = 0; \\ \begin{array}{r|l} Y = 1; & X = 1; \\ a = X; & b = Y; \end{array} \end{array}$$

Execution from Event Structure

$$\mathit{int}\ X = 0, Y = 0;$$

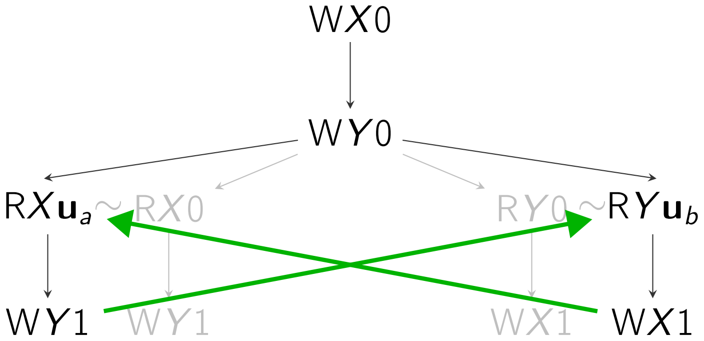
$$\begin{array}{l||l} a = X; & b = Y; \\ Y = 1; & X = 1; \end{array}$$

$$\begin{tikzcd}
 & & \mathrm{W}X0 \arrow[d] & & \\
 & & \mathrm{W}Y0 \arrow[dll] \arrow[dl] \arrow[dr] \arrow[drr] & & \\
\mathrm{R}X\mathbf{u}_a \arrow[r, phantom, "\sim"] \arrow[d] & \mathrm{R}X0 \arrow[d] & & \mathrm{R}Y0 \arrow[r, phantom, "\sim"] \arrow[d] & \mathrm{R}Y\mathbf{u}_b \arrow[d] \\
\mathrm{W}Y1 & \mathrm{W}Y1 & & \mathrm{W}X1 & \mathrm{W}X1
\end{tikzcd}$$

Execution from Event Structure

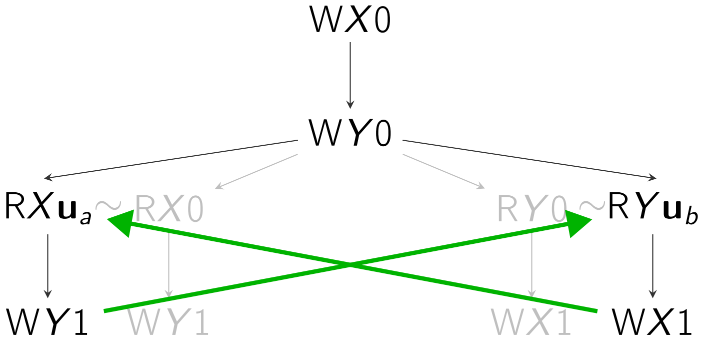
$$\textit{int}\ X = 0, Y = 0;$$

$$\begin{array}{l||l} a = X; & b = Y; \\ Y = 1; & X = 1; \end{array}$$



Execution from Event Structure

$$\textit{int}\ X = 0, Y = 0;$$
$$\begin{array}{l||l} a = X; & b = Y; \\ Y = 1; & X = 1; \end{array}$$



$$a = \mathbf{u}_a = 1,\ b = \mathbf{u}_b = 1$$

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)

Verifying Transformations

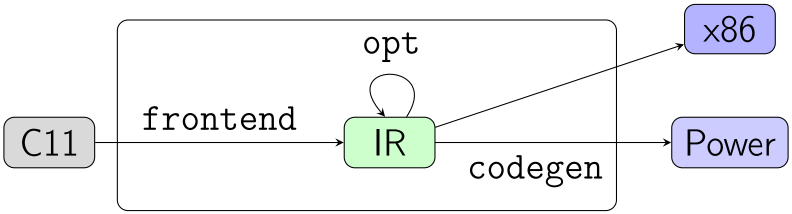
Formalized fragment of LLVM concurrency

Verified correctness of transformations

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

Validated LLVM opt-phase transformations

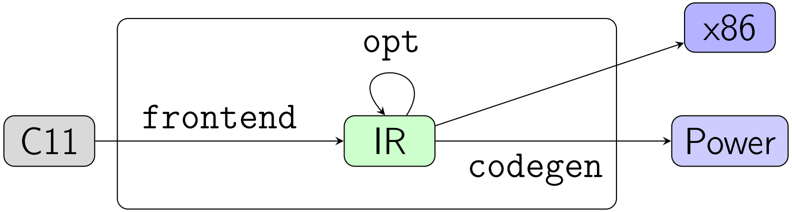
Transformation Correctness



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

Behavior: final values observed in each location

Transformation Correctness



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

Behavior: final values observed in each location

$$\Uparrow$$

$$\mathrm{Behavior}(G_{tgt}) \subseteq \mathrm{Behavior}(G_{src})$$

Elimination Optimizations

Adjacent read after read/write elimination

- $a = X_o; b = X_{\mathrm{na}}; \leadsto a = X_o; b = a;$
- $X_o = v; b = X_{\mathrm{na}}; \leadsto X_o = v; b = v;$

Adjacent overwritten write elimination

- $X_{\mathrm{na}} = v'; X_{\mathrm{na}} = v; \leadsto X_{\mathrm{na}} = v;$

Non-adjacent overwritten write elimination

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

LLVM performs these eliminations

Elimination Optimizations

Adjacent read after read/write elimination

- $a = X_o; b = X_{na}; \leadsto a = X_o; b = a;$
- $X_o = v; b = X_{na}; \leadsto X_o = v; b =$

Adjacent overwritten write el

- $X_{na} = v'; X_{na} = v;$ … $v;$

Non-adjacent … en write elimination

- X_n … $_{na} = v; \leadsto C; X_{na} = v;$

  -acq-pair $\notin$ C and $access(X) \notin C$

PROVEN CORRECT !

LLVM performs these eliminations

Also Proved...

Adjacent read after read/write elimination

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

Adjacent overwritten write elimination

- $X_{\text{rel}} = v'; X_{\text{rel}} = v; \rightsquigarrow X_{\text{rel}} = v;$
- $X_{(\text{rel}|\text{sc})} = v'; X_{\text{sc}} = v; \rightsquigarrow X_{\text{sc}} = v;$

LLVM does NOT perform these eliminations

Also Proved...

Adjacent read after read/write elimination

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

Adjacent overwritten write elimination

- $X_{\mathsf{rel}} = v'; X_{\mathsf{rel}} = v; \rightsquigarrow X_{\mathsf{rel}} = v;$
- $X_{(\mathsf{rel}|\mathsf{sc})} = v'; X_{\mathsf{sc}} = v; \rightsquigarrow X_{\mathsf{sc}} = v;$

LLVM does NOT perform these eliminations

Non-adjacent read after write elimination

- $X_{\mathsf{na}} = v; \mathsf{C}; a = X_{\mathsf{na}}; \rightsquigarrow X_{\mathsf{na}} = v; \mathsf{C}; a = v;$

  where rel-acq-pair $\notin$ C and $access(X) \notin C$

Also Proved...

Adjacent read after read/write elimination

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

Adjacent overwritten write elim

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

PROVEN CORRECT !

LLVM doe perform these eliminations

Non ead after write elimination

- X $= v; C; a = X_{na}; \leadsto X_{na} = v; C; a = v;$

  where rel-acq-pair $\notin$ C and $access(X) \notin C$

Verifying Transformations

Formalized fragment of LLVM concurrency

Verified correctness of transformations

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

Validated LLVM opt-phase transformations

LLVM Reorderings

$a; b \leadsto b; a$

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

$$X_{\mathsf{rel}} = v;\ Y_{\mathsf{na}} = v';\leadsto\ Y_{\mathsf{na}} = v';\ X_{\mathsf{rel}} = v; \qquad \checkmark$$

LLVM performs(✓) these reorderings

LLVM Reorderings

$a; b \rightsquigarrow b; a$

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

$$Y_{\mathsf{na}} = v'; X_{\mathsf{rel}} = v; \rightsquigarrow X_{\mathsf{rel}} = v; Y_{\mathsf{na}} = v'; \quad \times$$

LLVM restricts(×) these reorderings

Also Analyzed...

$a; b \leadsto b; a$

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

$$X_{\mathsf{rel}} = v; t = Y_{\mathsf{acq}}; \leadsto t = Y_{\mathsf{acq}}; X_{\mathsf{rel}} = v; \quad ✓$$

LLVM does NOT perform these reorderings

Also Analyzed...

$a; b \rightsquigarrow b; a$

$\downarrow a \setminus b \rightarrow$	$(\mathrm{St}\vert\mathrm{Ld})_{\mathrm{na}}$	$\mathrm{St}_{\mathrm{rel}}$	$\mathrm{Ld}_{\mathrm{acq}}$	L[illegible]	[illegible]q_rel|sc)
$(\mathrm{St}\vert\mathrm{Ld})_{\mathrm{na}}$	✓	×	✓	[illegible]	×
$\mathrm{St}_{\mathrm{rel}}$	✓	×	[illegible]	✓	×
$\mathrm{St}_{\mathrm{sc}}$	✓	[illegible]	✓	×	×
$\mathrm{Ld}_{\mathrm{acq}}$	×	×	×	×	×
$\mathrm{U}_{(\mathrm{acq\_rel}\vert\mathrm{s}}$[illegible]	[illegible]	×	×	×	×

PROVEN CORRECT !

[illegible]$; t = Y_{\mathrm{acq}}; \rightsquigarrow t = Y_{\mathrm{acq}}; X_{\mathrm{rel}} = v;$ ✓

LLVM does NOT perform these reorderings

Verifying Transformations

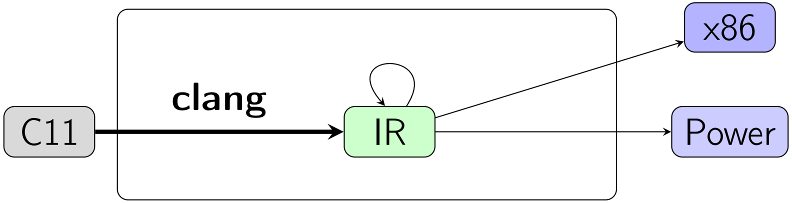
Formalized fragment of LLVM concurrency

Verified correctness of transformations

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

Validated LLVM opt-phase transformations

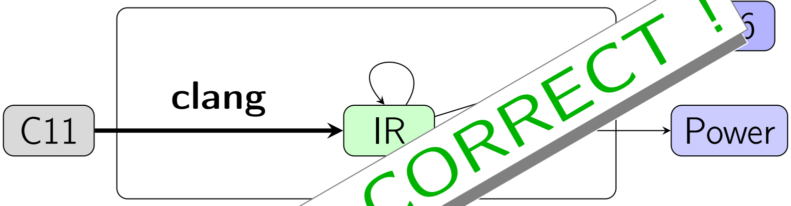
C11 to LLVM Mapping Correctness



- LLVM has operations (Ld/St/CAS) and memory orders (na/rel/acq/acq_rel/SC) similar to C11.

- LLVM model is stronger than C11.

C11 to LLVM Mapping Correctness

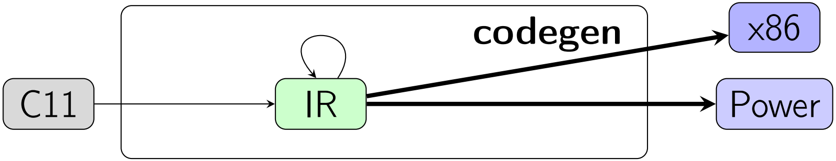


PROVEN CORRECT !

- LLVM h [illegible] tions (Ld/St/CAS) and memory orde [illegible] l/acq/acq_rel/SC) similar to C11.

- LLVM model is stronger than C11.

LLVM to Architecture Mapping Correctness

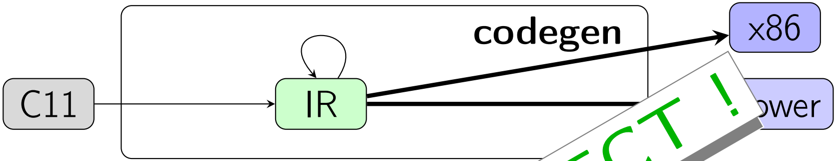


(LLVM $\rightsquigarrow$ x86/Power) = (C11 $\rightsquigarrow$ x86/Power)
Proved correctness of these mappings

- LLVM to SC
- LLVM to `SPower`

Ensure correctness of LLVM $\rightsquigarrow$ x86/Power
(results from Lahav & Vafeiadis. FM'16)

LLVM to Architecture Mapping Correctness



PROVEN CORRECT !

(LLVM $\rightsquigarrow$ x86/Power) = (… 86/Power)

Proved correctness o … appings

- LLVM to S
- LLV … wer

Ensu … orrectness of LLVM $\rightsquigarrow$ x86/Power

(results from Lahav & Vafeiadis. FM'16)

Validation

Formalized fragment of LLVM concurrency

Proved correctness of transformations

Validated LLVM opt-phase transformations

- $P_{src} \xrightarrow{\text{LLVM}} P_{tgt}$? **Correct** : **Potential Error**

LLVM Validation

$$P_{src} \xRightarrow{\text{LLVM}} P_{tgt}\ ?\ \textbf{Correct} : \textbf{Potential Error}$$

$$\Uparrow$$

$$P_{src} \xRightarrow{(R \cup E)^*} P_{tgt}\ ?\ \textbf{Correct} : \textbf{Potential Error}$$

- R: Safe reorderings
- E: Safe eliminations

Metadata Based Matching

$s_1 = X$!A

$s_2 = X$!B

$V = 1$!C

$s_4 = Z_{\text{acq}}$!D

$Y = 1$!E

$Y = 2$!F

Metadata Based Matching

✓ $s_1 = X$!A

$s_2 = X$!B

$V = 1$!C

$s_4 = Z_{\mathrm{acq}}$!D

$Y = 1$!E

$Y = 2$!F

Metadata Based Matching

✓ $s_1 = X$!A

✗ $s_2 = X$!B

$V = 1$!C

$s_4 = Z_{\text{acq}}$!D

$Y = 1$!E

$Y = 2$!F

Metadata Based Matching

✓ $s_1 = X$!A

✗ $s_2 = X$!B

$V = 1$!C

✓ $s_4 = Z_{\mathrm{acq}}$!D

$Y = 1$!E

$Y = 2$!F

Metadata Based Matching

✓ $s_1 = X$!A

✗ $s_2 = X$!B

$V = 1$!C

✓ $s_4 = Z_{\mathrm{acq}}$!D

$Y = 1$!E

✓ $Y = 2$!F

Metadata Based Matching

✓ $s_1 = X$!A

✗ $s_2 = X$!B

$V = 1$!C

✓ $s_4 = Z_{\text{acq}}$!D

✗ $Y = 1$!E

✓ $Y = 2$!F

Metadata Based Matching

- ✓ $s_1 = X$!A
- ✗ $s_2 = X$!B
- ✓ $V = 1$!C
- ✓ $s_4 = Z_{\mathrm{acq}}$!D
- ✗ $Y = 1$!E
- ✓ $Y = 2$!F

Metadata Based Matching

✓ $s_1 = X$!A

✗ $s_2 = X$!B

✓ $V = 1$!C

✓ $s_4 = Z_{\text{acq}}$!D

✗ $Y = 1$!E

✓ $Y = 2$!F

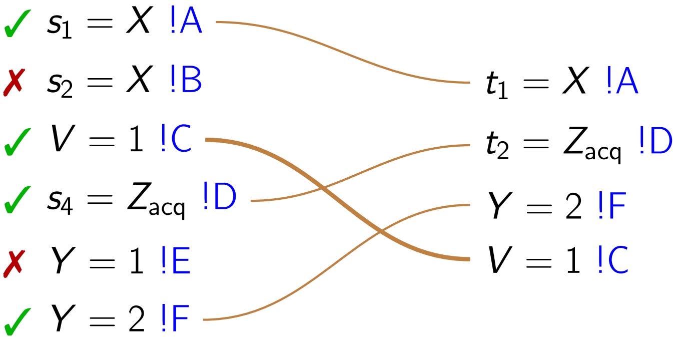
$t_1 = X$!A

$t_2 = Z_{\text{acq}}$!D

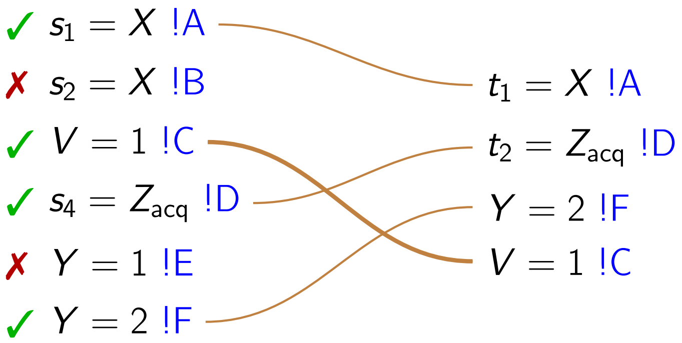
$Y = 2$!F

$V = 1$!C

Metadata Based Matching

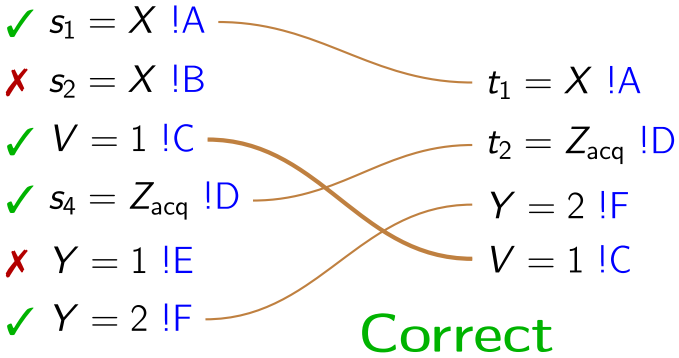


Metadata Based Matching



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

Metadata Based Matching



- 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}).
- $P_{src} \xRightarrow{\text{LLVM}} P_{tgt}$? **Correct** : **Potential Error**

More Details

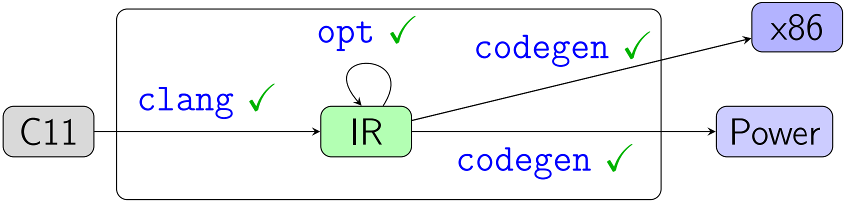
LLVM Formalization [CGO'17]

- 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



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 !