Validating optimisations of concurrent C/C++ programs

Viktor Vafeiadis

Max Planck Institute for Software Systems (MPI-SWS)

Joint work with Soham Chakraborty

Outline

Some questions:

- When is a compiler optimisation correct?
- Which optimisations are correct?
- How does one check that an optimisation is correct?
- Are compilers actually correct?
- What else can we do?

Compilation/optimisation correctness

Compilation:

$$source \xrightarrow{transform} target$$

Correctness:

$$Behaviours(target) \subseteq Behaviours(source)$$

Examples:

$$\begin{aligned} x = 1\,;\mathsf{print}(x); &\rightsquigarrow x = 1\,;\mathsf{print}(1); && (\checkmark)\\ \mathsf{print}(\mathit{unitialised\_variable}); &\rightsquigarrow \mathsf{print}(78); && (\checkmark)\\ \mathsf{print}(1)\,;\mathsf{print}(2) &\rightsquigarrow \mathsf{print}(1) \parallel \mathsf{print}(2) && (\times)\\ \mathsf{print}(1) \parallel \mathsf{print}(2) &\rightsquigarrow \mathsf{print}(1)\,;\mathsf{print}(2) && (\checkmark) \end{aligned}$$

Defining concurrent program behaviours

Sequential consistency (SC):

- Interleave each thread's atomic accesses.
- Most verification work assumes this model.

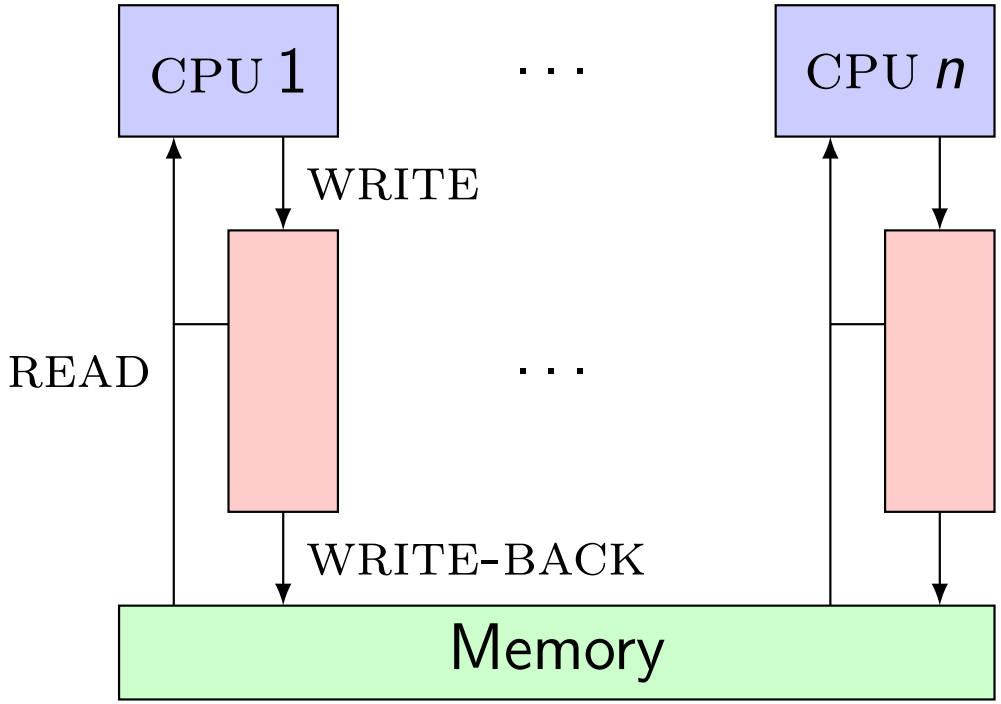
Initially, $X = Y = 0$.

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

Under SC, this program cannot return $a = b = 0$.

SC is, however, obsolete. It is invalidated by modern hardware and compiler optimisations.

Store buffering in x86-TSO



Initially, $X = Y = 0$.

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

Allowed outcome: $a = b = 0$.

Basic compiler optimisations break SC / TSO

Initially, $X = Y = 0$.

$$\begin{array}{l} X = 1; \\ Y = 1; \end{array} \;\Big\|\; \begin{array}{l} \text{print}(X); \\ \text{print}(Y); \\ \text{print}(X); \end{array}$$

Can the program print 010?

Justification:

The compiler may perform CSE:

Load X into a temporary t
and print t, Y, and t.

Data races are evil (Adve, Boehm, . . .)

Observe:

- Programs with weak behaviours have racy accesses.

$$\begin{array}{l} X = 1; \\ a = Y; \end{array} \;\Big\|\; \begin{array}{l} Y = 1; \\ b = X; \end{array} \qquad\qquad \begin{array}{l} X = 1; \\ Y = 1; \end{array} \;\Big\|\; \begin{array}{l} \mathrm{print}(X); \\ \mathrm{print}(Y); \\ \mathrm{print}(X); \end{array}$$

- Declare such programs wrong/undefined.
- Provide some support for intensional races.
  This is needed, e.g., to implement locks.

Consequences:

- Blame the programmer for writing racy programs!
- Allow the compiler to optimise aggresively!

Some optimisations are incorrect (Roach motel principle)

- Even so, some optimisations that are correct for sequential programs are not correct for concurrent programs.

```
atomic_int L = 0; int X = 0;
lock() {...}
unlock() {L = 0; /* atomic access */}
```

```
lock();          ||  lock();          ⇝   lock();          ||  lock();
X = 42;          ||  r = X;           ⇝   unlock();        ||  r = X;
unlock();        ||  unlock();        ⇝   X = 42;          ||  unlock();
```

- The transformation above introduces a race.

Should these transformations be allowed?

1. **CSE over a lock acquire:**

$$\begin{array}{l} t_1 = X; \\ lock(); \\ t_2 = X; \end{array} \quad \rightsquigarrow \quad \begin{array}{l} t_1 = X; \\ lock(); \\ t_2 = t_1; \end{array}$$

   If X changes in between, the program is racy.

2. **Load hoisting:**

$$\begin{array}{l} \text{if}(c) \\ \quad r = X; \end{array} \quad \rightsquigarrow \quad \begin{array}{l} t = X; \\ r = c \,?\, t : r; \end{array}$$

   This may introduce a race, but the racy value is not used.

Allowing both is clearly wrong!

Consider the transformation sequence:

$$\begin{array}{l} \text{if } (c) \\ \quad r_1 = X; \\ lock(); \\ r_2 = X; \end{array} \quad \rightsquigarrow \quad \begin{array}{l} t = X; \\ r_1 = c \ ? \ t : r_1; \\ lock(); \\ r_2 = X; \end{array} \quad \rightsquigarrow \quad \begin{array}{l} t = X; \\ r_1 = c \ ? \ t : r_1; \\ lock(); \\ r_2 = t; \end{array}$$

When c is false, X is moved out of the critical region!

So we have to forbid one transfomation.

- C11 forbids load hoisting, allows CSE over lock().
- LLVM allows load hoisting, forbids CSE over lock().

The C11 memory model

- Introduced by the C and C++ standards in 2011.
- First formalized by Batty et al., POPL 2011.
- Several subsequent fixes.

The C11 memory model

Two types of locations: ordinary and atomic

- Races on ordinary accesses $\rightsquigarrow$ error

A spectrum of atomic accesses:

- Relaxed $\rightsquigarrow$ no fence
- Consume reads $\rightsquigarrow$ no fence, but preserve deps
- Release writes $\rightsquigarrow$ no fence (x86); lwsync (PPC)
- Acquire reads $\rightsquigarrow$ no fence (x86); isync (PPC)
- Seq. consistent $\rightsquigarrow$ full memory fence

Primitives for explicit fences

Roach motel example in C11

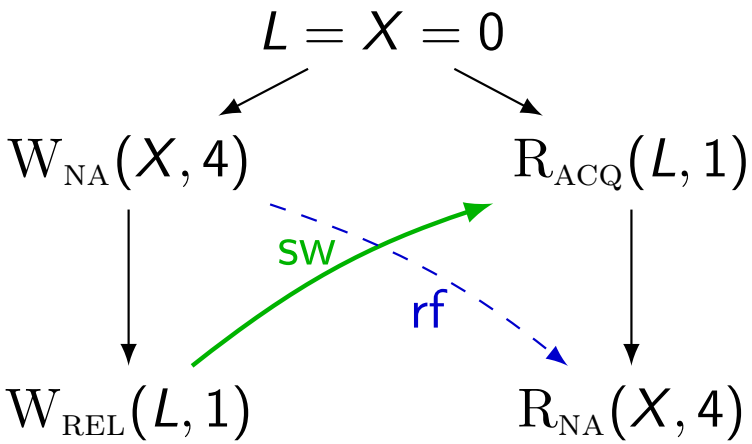
$$\texttt{atomic\_int}\ L = 0;\ \texttt{int}\ X = 0;$$

$$\begin{array}{l|l} X_{\mathrm{NA}} = 4; & \texttt{if}(L_{\mathrm{ACQ}}) \\ L_{\mathrm{REL}} = 1; & \quad r = X_{\mathrm{NA}}; \end{array}$$

Roach motel example in C11

$$\texttt{atomic\_int}\ L = 0;\ \texttt{int}\ X = 0;$$

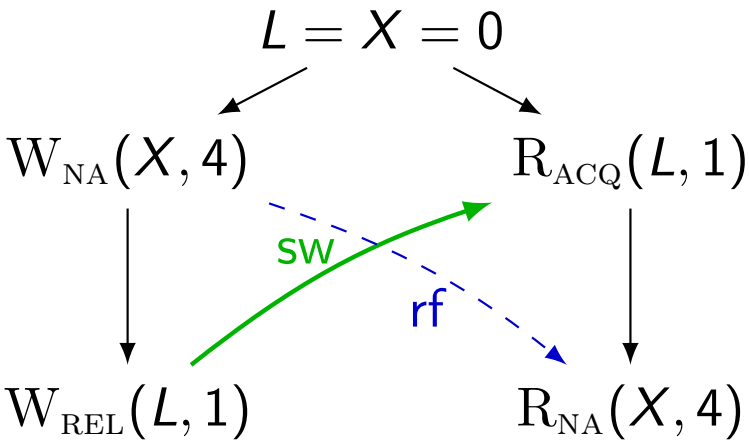
$$\begin{array}{l|l} X_{\mathrm{NA}} = 4; & \mathsf{if}(L_{\mathrm{ACQ}}) \\ L_{\mathrm{REL}} = 1; & \quad r = X_{\mathrm{NA}}; \end{array}$$

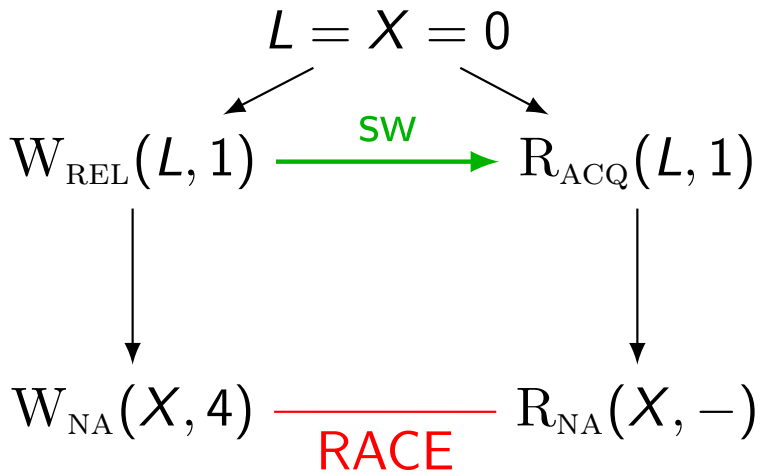


Roach motel example in C11

$$\texttt{atomic\_int } L = 0;\ \texttt{int } X = 0;$$

$$\begin{array}{l|l} X_{\mathrm{NA}} = 4; & \mathsf{if}(L_{\mathrm{ACQ}}) \\ L_{\mathrm{REL}} = 1; & \quad r = X_{\mathrm{NA}}; \end{array} \quad \rightsquigarrow \quad \begin{array}{l|l} L_{\mathrm{REL}} = 1; & \mathsf{if}(L_{\mathrm{ACQ}}) \\ X_{\mathrm{NA}} = 4; & \quad r = X_{\mathrm{NA}}; \end{array}$$





Reorderings

- ✓ $unlock(); X = 1; \quad \rightsquigarrow \quad X = 1; unlock();$
- ✗ $X = 1; unlock(); \quad \rightsquigarrow \quad unlock(); X = 1;$

Eliminations

- ✓ $t = X; t' = X; \quad \rightsquigarrow \quad t = X; t' = t;$
- ✓ $t = X; lock(); t' = X; \quad \rightsquigarrow \quad t = X; lock(); t' = t;$
- ✓ $t = X; unlock(); t' = X; \quad \rightsquigarrow \quad t = X; unlock(); t' = t;$
- ✗ $t = X; unlock(); lock(); t' = X;$
  $\rightsquigarrow \quad t = X; unlock(); lock(); t' = t;$

Valid instruction reorderings $a\,;b \rightsquigarrow b\,;a$ (POPL'15)

$\downarrow a \setminus b \rightarrow$	$\mathsf{R}_{\neq\mathrm{sc}}$	R_{sc}	W_{na}	$\mathsf{W}_{\mathrm{rlx}}$	$\mathsf{W}_{\sqsupseteq\mathrm{rel}}$	$\mathsf{C}_{\mathrm{rlx}\mid\mathrm{acq}}$	$\mathsf{C}_{\sqsupseteq\mathrm{rel}}$	$\mathsf{F}_{\mathrm{acq}}$	$\mathsf{F}_{\mathrm{rel}}$
R_{na}	✓	✓	✓	✓	✗	✓	✗	✓	✗
$\mathsf{R}_{\mathrm{rlx}}$	✓	✓	✓	(✓)	✗	(✓)	✗	✗	✗
$\mathsf{R}_{\sqsupseteq\mathrm{acq}}$	✗	✗	✗	✗	✗	✗	✗	✓	✗
$\mathsf{W}_{\neq\mathrm{sc}}$	✓	✓	✓	✓	✗	✓	✗	✓	✗
W_{sc}	✓	✗	✓	✓	✗	✓	✗	✓	✗
$\mathsf{C}_{\mathrm{rlx}\mid\mathrm{rel}}$	✓	✓	✓	(✓)	✗	(✓)	✗	✗	✗
$\mathsf{C}_{\sqsupseteq\mathrm{acq}}$	✗	✗	✗	✗	✗	✗	✗	✓	✗
$\mathsf{F}_{\mathrm{acq}}$	✗	✗	✗	✗	✗	✗	✗	$=$	✗
$\mathsf{F}_{\mathrm{rel}}$	✓	✓	✓	✗	✓	✗	✓	✓	$=$

Redundant instruction eliminations (POPL'15)

Overwritten write:

$$\begin{array}{rll} & x.\mathsf{store}(v, M)\,;\, C\,;\, x.\mathsf{store}(v', M) & C \text{ has no rel} \\ \rightsquigarrow & C\,;\, x.\mathsf{store}(v', M) & \& \text{ no } x \text{ accesses} \end{array}$$

Read after write:

$$\begin{array}{rll} & x.\mathsf{store}(v, M)\,;\, C\,;\, t = x.\mathsf{load}(M') & C \text{ has no acq} \\ \rightsquigarrow & x.\mathsf{store}(v, M)\,;\, C\,;\, t = v & \& \text{ no } x \text{ accesses} \end{array}$$

Read after read:

$$\begin{array}{rll} & t = x.\mathsf{load}(M)\,;\, C\,;\, t' = x.\mathsf{load}(M) & C \text{ has no acq} \\ \rightsquigarrow & t = x.\mathsf{load}(M)\,;\, C\,;\, t' = t & \& \text{ no } x \text{ accesses} \end{array}$$

The LLVM memory model

- Described in prose in the LLVM documentation.
- Formalization in progress.

C11 vs LLVM

Overview

- LLVM concurrency constructs match those of C11.
- But they give different semantics to racy programs.

C11 semantics

- In C11, racy programs have undefined behavior.

LLVM semantics

- *Write-write* races result in undefined behavior.
- *Write-read* races have defined behavior.
- The racy reads return undef ($\approx$ any value).

The semantics of "undef" in LLVM

The semantics of "undef" in LLVM is very weak.

- Arithmetic on "undef" returns "undef".
- Branching on "undef" could go either way.
- The following program, starting with $X = Y = Z = 0$,

$$Z_{\mathrm{NA}} = 5; \;\Big\|\; \begin{array}{l} t = Z_{\mathrm{NA}}; \\ \mathrm{if}(t == 1)\ X_{\mathrm{NA}} = 1; \\ \mathrm{if}(t == 7)\ Y_{\mathrm{NA}} = 1; \end{array}$$

  is allowed to return $X = Y = 1$.
- This semantics allows some very aggressive optimisations. For example, $X = \mathit{undef}; \;\rightsquigarrow\; \mathit{skip};$

Impact on optimization: C11 vs LLVM

Non-atomic read introduction:

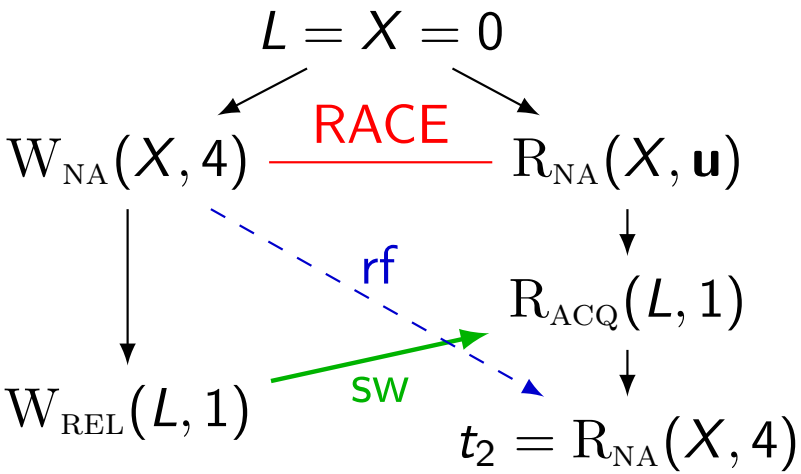
- Unsafe in C11
- Allowed in LLVM

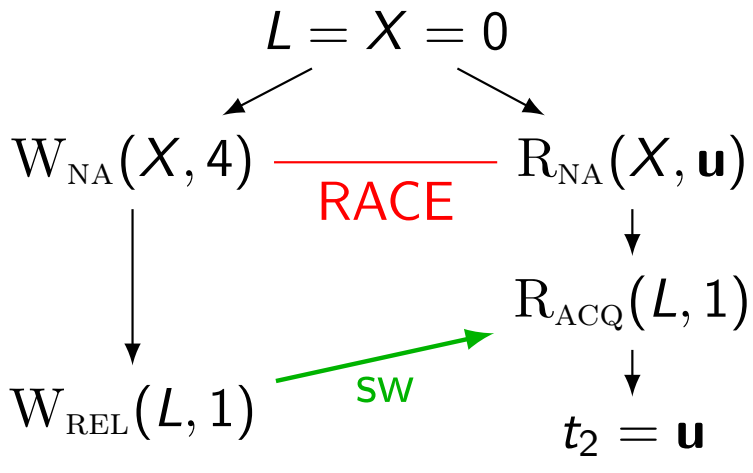
$$X_{\mathrm{NA}} = 1; \;\Big\|\; \begin{array}{l} r = 0; \\ \text{if } (f) \\ \quad r = X_{\mathrm{NA}}; \end{array} \quad \rightsquigarrow \quad X_{\mathrm{NA}} = 1; \;\Big\|\; \begin{array}{l} r = 0; \\ t = X_{\mathrm{NA}}; \\ r = f \;?\; t : 0; \end{array}$$

Read elimination over an acquire action

Allowed in C11 but unsafe in LLVM.

$$\begin{array}{l} X_{\mathrm{NA}} = 4; \\ L_{\mathrm{REL}} = 1; \end{array} \;\Big\|\; \begin{array}{l} t_1 = X_{\mathrm{NA}}; \\ \mathrm{if}(L_{\mathrm{ACQ}}) \\ \quad t_2 = X_{\mathrm{NA}}; \end{array} \quad \rightsquigarrow \quad \begin{array}{l} X_{\mathrm{NA}} = 4; \\ L_{\mathrm{REL}} = 1; \end{array} \;\Big\|\; \begin{array}{l} t_1 = X_{\mathrm{NA}}; \\ \mathrm{if}(L_{\mathrm{ACQ}}) \\ \quad t_2 = t_1; \end{array}$$





Compiler validation

- Translation validation is a generic approach.
- We apply it to the LLVM compiler.
- Specifically to check for concurrecy errors.

Compiler validation

$$(P_{src} \xRightarrow{comp.} P_{tgt})\ ?\ \textbf{Correct}\ :\ \textbf{Error}$$

$$\Downarrow$$

1. Identify safe reorderings(R) & safe eliminations(E)
2. $(P_{src} \xRightarrow{(R \cup E)^*} P_{tgt})\ ?\ \textbf{Correct}\ :\ \textbf{Error}$

Validation approaches

Metadata-based validation (MD)

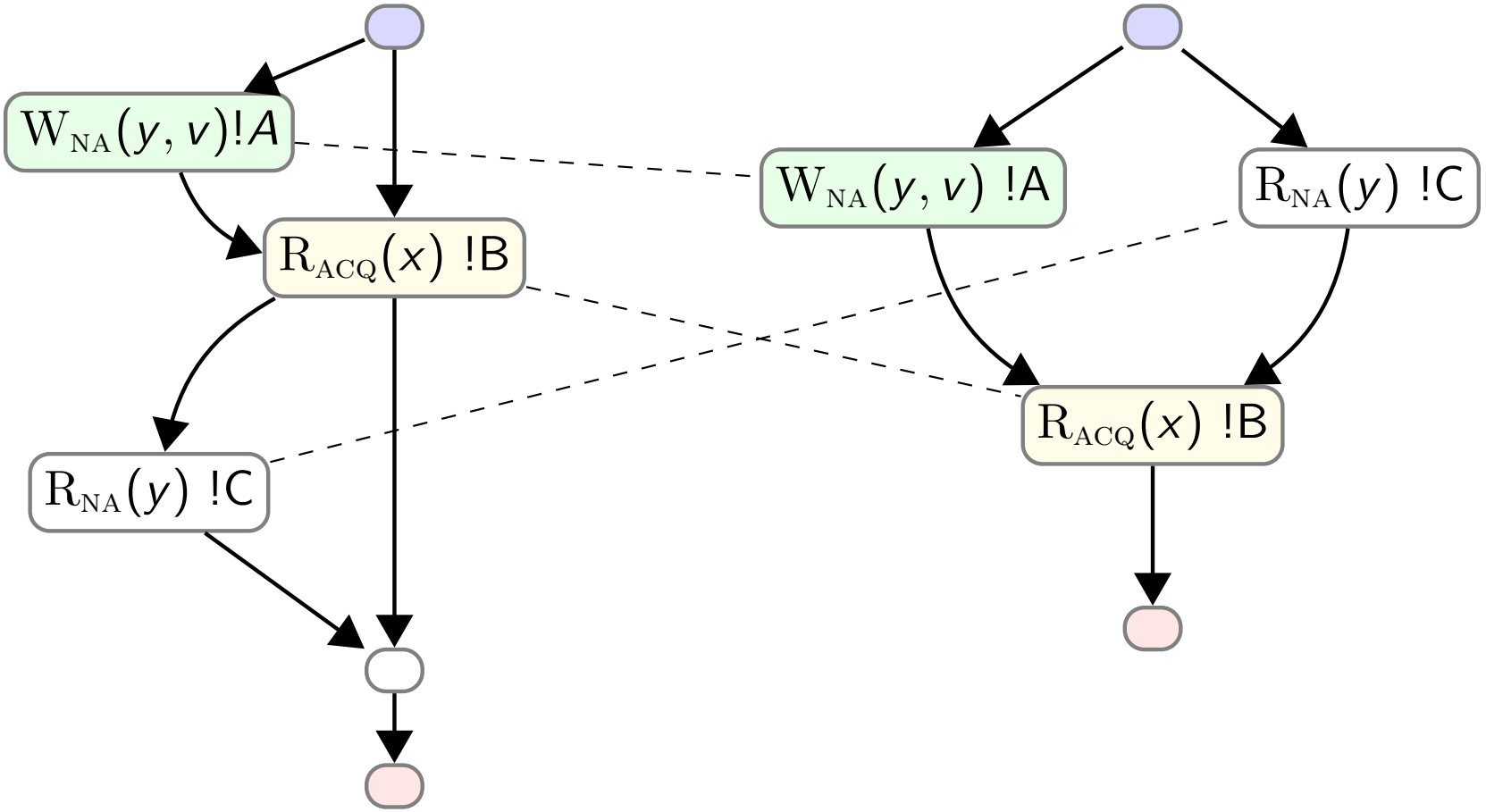
- Use LLVM metadata capability.
- Modify compiler to generate required metadata.

Compiler-independent matching (CIM)

- Independent of compiler internals.
- Applicable to any compiler.

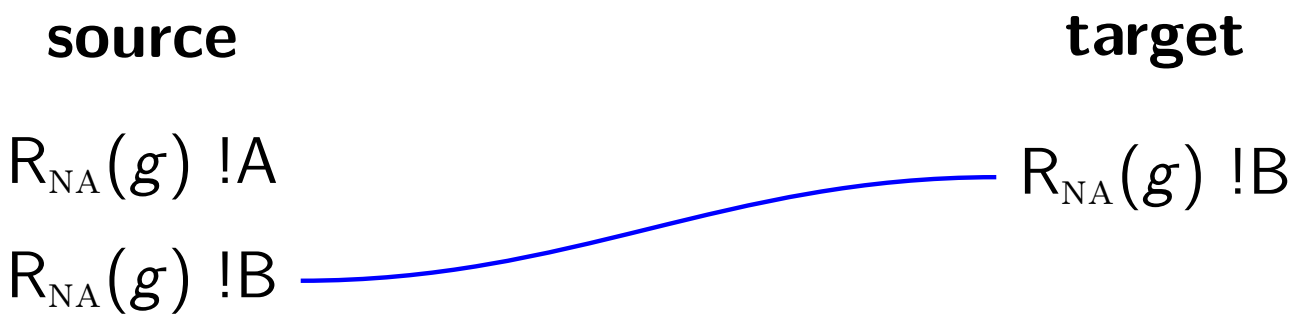
Metadata-based validation (MD)

- Match accesses based on attached metadata.
- Analyze the matching on source and target CFGs.



Metadata-based validation (MD)

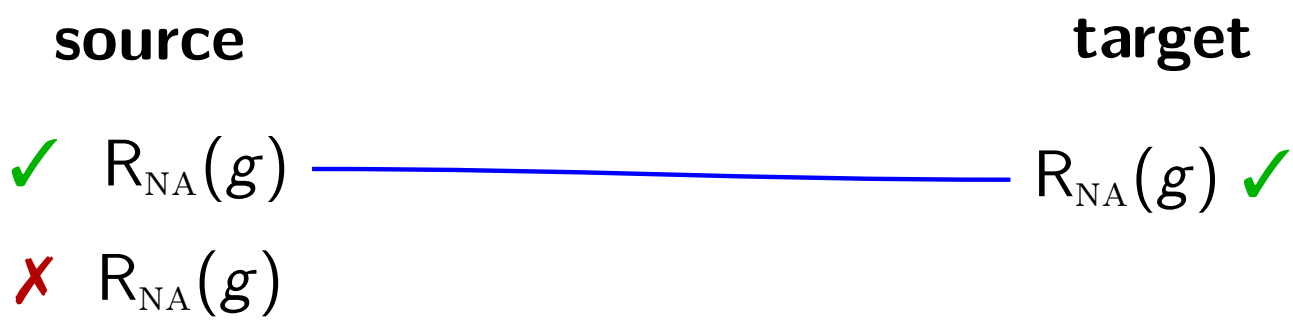
Can give false positives!



MD Report: **Error**

Compiler-independent matching (CIM)

- Mark action deletability/non-deletability
- Match access sequences and analyze



CIM reports: Correct

CIM example

source

$(C) \otimes a : \mathrm{W}_{\mathrm{RLX}}(X, v_x)$

✓ $b : \mathrm{W}_{\mathrm{REL}}(Y, v_y)$

✓ $c : \mathrm{R}_{\mathrm{NA}}(g)$

✓ $d : \mathrm{W}_{\mathrm{RLX}}(X, v'_x)$

✓ $e : \mathrm{R}_{\mathrm{ACQ}}(Z)$

✗ $f : \mathrm{R}_{\mathrm{NA}}(g)$

$C = [a; \mathrm{W}_{\mathrm{RLX}}(X, -); b]$

target

$a' : \mathrm{W}_{\mathrm{RLX}}(X, v'_x)$ ✓

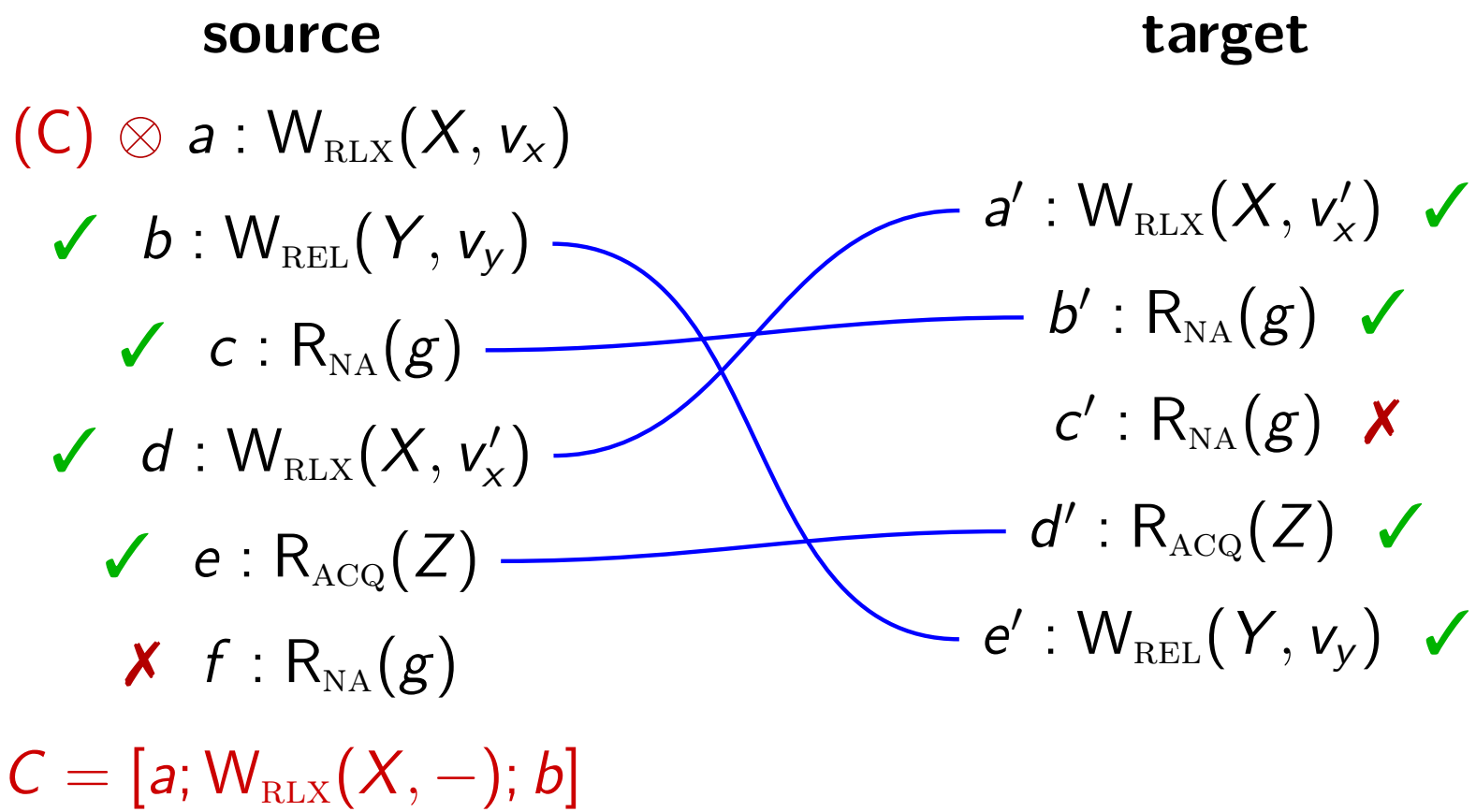
$b' : \mathrm{R}_{\mathrm{NA}}(g)$ ✓

$c' : \mathrm{R}_{\mathrm{NA}}(g)$ ✗

$d' : \mathrm{R}_{\mathrm{ACQ}}(Z)$ ✓

$e' : \mathrm{W}_{\mathrm{REL}}(Y, v_y)$ ✓

CIM example

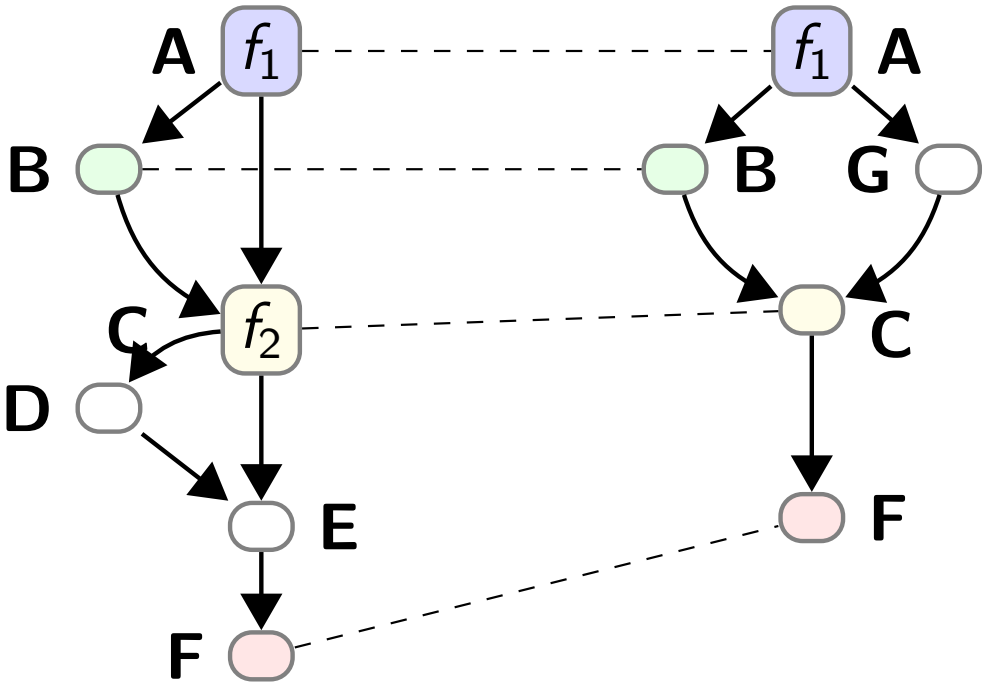


$C = [a; \mathrm{W}_{\mathrm{RLX}}(X, -); b]$

Report: **Correct**

Control flow matching

- Match equivalent paths between source and target



$\text{ABCF}(f_1) \rightarrow \{\text{ABCDEF}(f_1 \wedge f_2), \text{ABCEF}(f_1 \wedge \neg f_2)\}$

$\text{AGCF}(\neg f_1) \rightarrow \{\text{ACDEF}(\neg f_1 \wedge f_2), \text{ACEF}(\neg f_1 \wedge \neg f_2)\}$

Experimental results

Tests	Model	End-to-end				Stepwise			
		llvm 3.6		llvm 3.7rc2		llvm 3.6		llvm 3.7rc2	
		CIM	MD	CIM	MD	CIM	MD	CIM	MD
straight	LLVM	95	95	0	0	95 †	95 †	0	0
	C11	0	0	0	0	0	0	0	0
& cond	LLVM	66	74	0	6	64 †	77 †‡	0	0
	C11	12	39	0	27	15 †‡	43 †‡	1 †	31 †‡
& dead	LLVM	58	74	0	3	57 †	79 †‡	0	0
	C11	6	40	0	25	11 †	49 †‡	0	26 †‡
& loop	LLVM	48	72	0	27	48 †	90 †‡	0	0
	C11	6	48	0	32	7 †	54 †‡	0	44 †‡
small	LLVM	32	38	0	8	32 †	40 †‡	0	0
	C11	7	20	5	19	7 †‡	24 †‡	6 †	23 †‡

- In total, 500 tests and about 7000 non-trivial steps.
- Report # failed validations.
- Erroneous passes: † GVN and ‡ SimplifyCFG

Experimental results

Tests	Model	End-to-end				Stepwise			
		llvm 3.6		llvm 3.7rc2		llvm 3.6		llvm 3.7rc2	
		CIM	MD	CIM	MD	CIM	MD	CIM	MD
straight	LLVM	95	95	0	0	95 †	95 †	0	0
	C11	0	0	0	0	0	0	0	0
& cond	LLVM	66	74	0	6	64 †	77 †‡	0	0
	C11	12	39	0	27	15 †‡	43 †‡	1 †	31 †‡
& dead	LLVM	58	74	0	3	57 †	79 †‡	0	0
	C11	6	40	0	25	11 †	49 †‡	0	26 †‡
& loop	LLVM	48	72	0	27	48 †	90 †‡	0	0
	C11	6	48	0	32	7 †	54 †‡	0	44 †‡
small	LLVM	32	38	0	8	32 †	40 †‡	0	0
	C11	7	20	5	19	7 †‡	24 †‡	6 †	23 †‡

Compiler independent maching (CIM) finds:

- plenty of errors on LLVM 3.6;
- no errors on LLVM 3.7rc2 (i.e., also no false positives);
- relatively few errors on LLVM 3.7rc2 w.r.t. the C11 model.

Experimental results

Tests	Model	End-to-end				Stepwise			
		llvm 3.6		llvm 3.7rc2		llvm 3.6		llvm 3.7rc2	
		CIM	MD	CIM	MD	CIM	MD	CIM	MD
straight	LLVM	95	95	0	0	95 †	95 †	0	0
	C11	0	0	0	0	0	0	0	0
& cond	LLVM	66	74	0	6	64 †	77 †‡	0	0
	C11	12	39	0	27	15 †‡	43 †‡	1 †	31 †‡
& dead	LLVM	58	74	0	3	57 †	79 †‡	0	0
	C11	6	40	0	25	11 †	49 †‡	0	26 †‡
& loop	LLVM	48	72	0	27	48 †	90 †‡	0	0
	C11	6	48	0	32	7 †	54 †‡	0	44 †‡
small	LLVM	32	38	0	8	32 †	40 †‡	0	0
	C11	7	20	5	19	7 †‡	24 †‡	6 †	23 †‡

Some stepwise errors are masked by other transformations:

$$r = X_{\text{ACQ}};\quad \text{if(a) } r' = X_{\text{NA}}; \xrightarrow{(1)} r = X_{\text{ACQ}};\quad t = X_{\text{NA}};\quad r' = a\ ?\ t : r'; \xrightarrow{(2)} r = X_{\text{ACQ}};\quad t = r;\quad r' = a\ ?\ t : r';$$

Experimental results

Tests	Model	End-to-end				Stepwise			
		llvm 3.6		llvm 3.7rc2		llvm 3.6		llvm 3.7rc2	
		CIM	MD	CIM	MD	CIM	MD	CIM	MD
straight	LLVM	95	95	0	0	95 †	95 †	0	0
	C11	0	0	0	0	0	0	0	0
& cond	LLVM	66	74	0	6	64 †	77 †‡	0	0
	C11	12	39	0	27	15 †‡	43 †‡	1 †	31 †‡
& dead	LLVM	58	74	0	3	57 †	79 †‡	0	0
	C11	6	40	0	25	11 †	49 †‡	0	26 †‡
& loop	LLVM	48	72	0	27	48 †	90 †‡	0	0
	C11	6	48	0	32	7 †	54 †‡	0	44 †‡
small	LLVM	32	38	0	8	32 †	40 †‡	0	0
	C11	7	20	5	19	7 †‡	24 †‡	6 †	23 †‡

- Stepwise errors are also masked by adjacent accesses.
- Use MD or smaller programs.

Experimental results

Tests	Model	End-to-end				Stepwise			
		llvm 3.6		llvm 3.7rc2		llvm 3.6		llvm 3.7rc2	
		CIM	MD	CIM	MD	CIM	MD	CIM	MD
straight	LLVM	95	95	0	0	95 †	95 †	0	0
	C11	0	0	0	0	0	0	0	0
& cond	LLVM	66	74	0	6	64 †	77 †‡	0	0
	C11	12	39	0	27	15 †‡	43 †‡	1 †	31 †‡
& dead	LLVM	58	74	0	3	57 †	79 †‡	0	0
	C11	6	40	0	25	11 †	49 †‡	0	26 †‡
& loop	LLVM	48	72	0	27	48 †	90 †‡	0	0
	C11	6	48	0	32	7 †	54 †‡	0	44 †‡
small	LLVM	32	38	0	8	32 †	40 †‡	0	0
	C11	7	20	5	19	7 †‡	24 †‡	6 †	23 †‡

MD is good stepwise, but not end-to-end.

$$\text{if}(*)\{\ \checkmark \mathrm{R}_{\mathrm{NA}}(g)\ !\mathrm{A}\}\quad \checkmark \mathrm{R}_{\mathrm{NA}}(g)\ !\mathrm{B} \quad \overset{(1)}{\rightsquigarrow} \quad \checkmark\ \mathrm{R}_{\mathrm{NA}}(g)\ !\mathrm{A}\quad \times\ \mathrm{R}_{\mathrm{NA}}(g)\ !\mathrm{B} \quad \overset{(2)}{\rightsquigarrow} \quad \checkmark\ \mathrm{R}_{\mathrm{NA}}(g)\ !\mathrm{A}$$

Conclusion

Summary

- C11 and LLVM models differ.
- This affects allowed compiler optimisations.
- Validator reveals compilation bugs.

Future directions

- Extend validation to loop optimisations.
- Properly formalize LLVM semantics.