Formal Validation of Intra-Procedural Transformations by Defensive Symbolic Simulation

PhD Defense of Léo Gourdin — 12/12/2023

leo.gourdin@univ-grenoble-alpes.fr

Advisors:
Sylvain Boulmé (Verimag)
Frédéric Pétrot (TIMA)

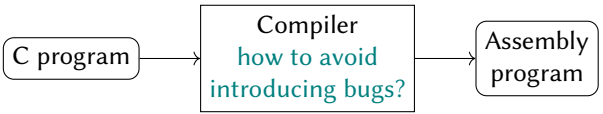
Committee:
Delphine Demange — Examiner
Jean-Christophe Filliâtre — Rapporteur
Jens Knoop — Rapporteur
Marc Pouzet — Examiner
Gwen Salaün — Examiner

Contents

Motivations: compilation bugs

[Yang et al. 2011; Sun et al. 2016; Zhou et al. 2021]

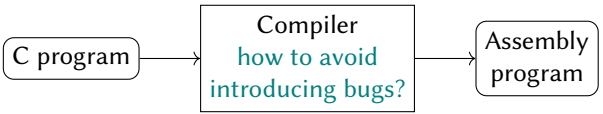
Compilers: **translate & optimize** programs (source language → target language).



Motivations: compilation bugs

[Yang et al. 2011; Sun et al. 2016; Zhou et al. 2021]

Compilers: **translate & optimize** programs (source language → target language).

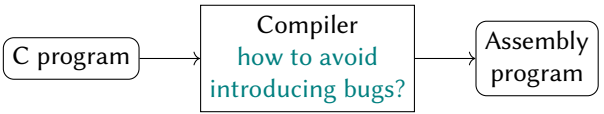


Optimizations: **buggiest component**
(except for the C++ component)
in both GCC and LLVM.

Motivations: compilation bugs

[Yang et al. 2011; Sun et al. 2016; Zhou et al. 2021]

Compilers: **translate** & **optimize** programs (source language → target language).



Optimizations: **buggiest component**
(except for the C++ component)
in both GCC and LLVM.

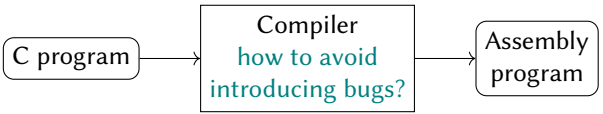
Bugs may alter program semantics, and thus program **behavior**.

Avoiding bugs in **safety-critical systems** (planes, trains, elevators, ...) is essential.

Motivations: compilation bugs

[Yang et al. 2011; Sun et al. 2016; Zhou et al. 2021]

Compilers: **translate & optimize** programs (source language → target language).



Optimizations: **buggiest component**
(except for the C++ component)
in both GCC and LLVM.

Bugs may alter program semantics, and thus program **behavior**.

Avoiding bugs in **safety-critical systems** (planes, trains, elevators, ...) is essential.

- $> 50\%$ optimizations bugs result in **incorrect generated code**
- Last 20 years: > 8700 optimization bugs identified in GCC (vs. > 1500 for LLVM)
- Bugs that crash compiler are easier to trace than optimization bugs

The CompCert compiler, verified in Coq

[Blazy et al. 2006; Leroy 2009]

CompCert (ACM Software System & ACM Programming Languages Software Awards):

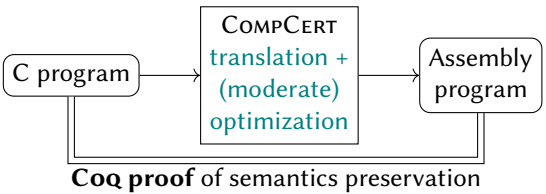
is the 1^{st} formally verified C compiler

The CompCert compiler, verified in Coq

[Blazy et al. 2006; Leroy 2009]

CompCert (ACM Software System & ACM Programming Languages Software Awards):

is the 1st formally verified C compiler



Formal correctness of CompCert:

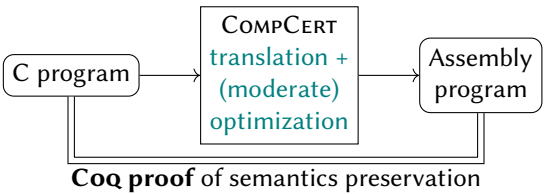
For any source program S in C language,
if S has no undefined behavior, and if
the compiler returns some assembly program T,
then any behavior of T is also a behavior of S.

The CompCert compiler, verified in Coq

[Blazy et al. 2006; Leroy 2009]

CompCert (ACM Software System & ACM Programming Languages Software Awards):

is the 1st formally verified C compiler



Formal correctness of CompCert:

For any source program S in C language,
if S has no undefined behavior, and if
the compiler returns some assembly program T,
then any behavior of T is also a behavior of S.

However... it is still less optimizing than "*trusted*" (non-proven) compilers (e.g. GCC)

Embedded/safe often means simple: compiler optimizations are then even more important.

Goal: correct & efficient code for embedded cores

Predictability, security, or safety norms often require [França et al. 2012]:

- **no dynamic reordering** inside processors (instruction scheduling)
- **no speculative execution** (guessing conditions)
- **simpler instruction sets**, such as RISC-V

Goal: correct & efficient code for embedded cores

Predictability, security, or safety norms often require [França et al. 2012]:

- **no dynamic reordering** inside processors (instruction scheduling)
- **no speculative execution** (guessing conditions)
- **simpler instruction sets**, such as RISC-V

$\rightarrow$ efficiency is therefore the compiler's job

Goal: correct & efficient code for embedded cores

Predictability, security, or safety norms often require [França et al. 2012]:

- **no dynamic reordering** inside processors (instruction scheduling)
- **no speculative execution** (guessing conditions)
- **simpler instruction sets**, such as RISC-V

$\rightarrow$ efficiency is therefore the compiler's job

Many optimizations of GCC/LLVM are still missing:

Code Motion:	moving instructions at better places, e.g. out of loops
Strength-reduction:	replacing costly instructions (e.g. multiplications) by simpler ones (e.g. additions)
Software pipelining:	optimization of loop bodies (e.g. by scheduling instructions above/below conditions)

Formally verified translation validation

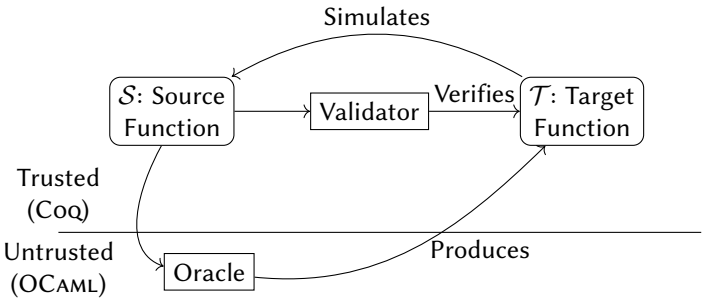
Proving such complex optimizations is difficult, like solving a sudoku...
...but checking a sudoku solution for correctness is much easier!

Formally verified translation validation

Proving such complex optimizations is difficult, like solving a sudoku...
...but checking a sudoku solution for correctness is much easier!

We call this idea
Translation Validation

Used for Register Allocation
[Rideau and Leroy 2010]

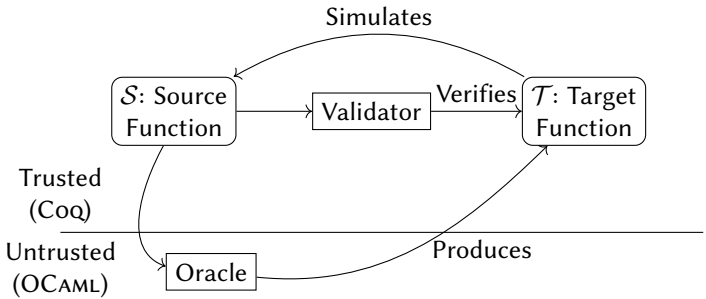


Formally verified translation validation

Proving such complex optimizations is difficult, like solving a sudoku...
...but checking a sudoku solution for correctness is much easier!

We call this idea
Translation Validation

Used for Register Allocation
[Rideau and Leroy 2010]



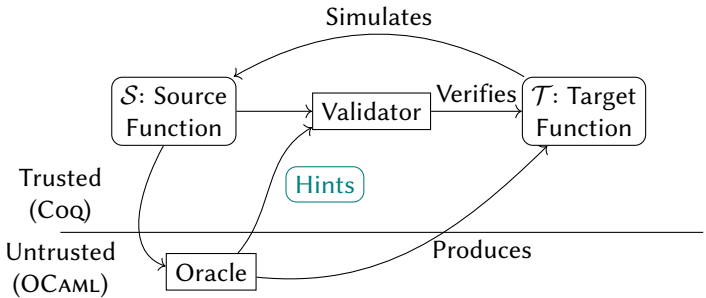
Complex computations by **efficient functions, called oracles,** with an untrusted and hidden implementation for the formal proof.
→ only a dynamic **defensive test** of their result is formally verified

Formally verified translation validation

Proving such complex optimizations is difficult, like solving a sudoku...
...but checking a sudoku solution for correctness is much easier!

We call this idea Defensive Translation Validation

Used for Register Allocation [Rideau and Leroy 2010]



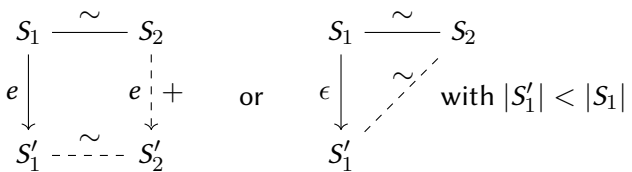
Complex computations by **efficient functions, called oracles,** with an untrusted and hidden implementation for the formal proof.
→ only a dynamic **defensive test** of their result is formally verified

A few details on CompCert's formalism

Program behavior $\triangleq$ sequence of observable events

Undefined behavior $\triangleq$ "errors" in the C semantics

Theorem of correctness by composing **forward simulations** between deterministic languages.



Each **source** step $S_1 \to^e S_1'$ is simulated by **target** steps
without infinite successive stutterings;
absence of step represents **Undefined Behavior**

Motivating RISC-V example (1/2)

```
1  double foo(double *a, long *v, long n) {
2    long k = 7; long i = 0;
3    double r = 2;
4    if (a[0] < 2) return 2;
5    for(; i < n; i += 4) {
6      if (r >= a[1]) r -= a[0];
7      else r *= 3;
8      r += v[i] - k * i;
9    }
10   return r;
11 }
```

Motivating RISC-V example (1/2)

```
1  double foo(double *a, long *v, long n) {
2    long k = 7; long i = 0;
3    double r = 2;
4    if (a[0] < 2) return 2;
5    for(; i < n; i += 4) {
6      if (r >= a[1]) r -= a[0];
7      else r *= 3;
8      r += v[i] - k * i;
9    }
10   return r;
11 }
```

CompCert optimizations are applied on **register transfer language (RTL)**

Motivating RISC-V example (1/2)

```
1  foo(a, v, n) {
2    k = 7; i = 0; r = 2f
3    x17 = float64[a+0] // previous occurrence
4    if (x17 <f r) { goto Exit }
5  Loop:
6    if (i >=ls n) { goto Exit }
7    x16 = float64[a+8] // a[1] (unsafe)
8    if (r >=f x16) {
9      x14 = float64[a+0]; // safe to eliminate
10     r = r -f x14 }
11   else { x15 = 3f; r = r *f x15 } // PRE
12   x13 = i <<l 3 // SR (addressing)
13   x12 = v +l x13 // SR (in sequence)
14   x10 = int64[x12+0]
15   x11 = i *l k // SR
16   x9 = x10 -l x11
17   x8 = floatoflong(x9)
18   r = r +f x8
19   i = i +l 4
20   goto Loop
21 Exit: return r }
```

```
1  double foo(double *a, long *v, long n) {
2    long k = 7; long i = 0;
3    double r = 2;
4    if (a[0] < 2) return 2;
5    for(; i < n; i += 4) {
6      if (r >= a[1]) r -= a[0];
7      else r *= 3;
8      r += v[i] - k * i;
9    }
10   return r;
11 }
```

CompCert optimizations are applied on **register transfer language (RTL)**

Left frame: naive RISC-V (pseudo)code (mainline CompCert)!

Motivating RISC-V example (2/2)

```
1  foo(a, v, n) {
2    k = 7; i = 0; r = 2f
3    x17 = float64[a+0] // previous occurrence
4    if (x17 <f r) { goto Exit }
5  Loop:
6    if (i >=ls n) { goto Exit }
7    x16 = float64[a+8] // a[1] (unsafe)
8    if (r >=f x16) {
9      x14 = float64[a+0]; // safe to eliminate
10     r = r -f x14 }
11   else { x15 = 3f; r = r *f x15 } // PRE
12   x13 = i <<l 3 // SR (addressing)
13   x12 = v +l x13 // SR (in sequence)
14   x10 = int64[x12+0]
15   x11 = i *l k // SR
16   x9 = x10 -l x11
17   x8 = floatoflong(x9)
18   r = r +f x8
19   i = i +l 4
20   goto Loop
21 Exit: return r }
```

What is needed?

Motivating RISC-V example (2/2)

```
1  foo(a, v, n) {
2    k = 7; i = 0; r = 2f
3    x17 = float64[a+0] // previous occurrence
4    if (x17 <f r) { goto Exit }
5  Loop:
6    if (i >=ls n) { goto Exit }
7    x16 = float64[a+8] // a[1] (unsafe)
8    if (r >=f x16) {
9      x14 = float64[a+0]; // safe to eliminate
10     r = r -f x14 }
11   else { x15 = 3f; r = r *f x15 } // PRE
12   x13 = i <<l 3 // SR (addressing)
13   x12 = v +l x13 // SR (in sequence)
14   x10 = int64[x12+0]
15   x11 = i *l k // SR
16   x9 = x10 -l x11
17   x8 = floatoflong(x9)
18   r = r +f x8
19   i = i +l 4
20   goto Loop
21 Exit: return r }
```

What is needed?

- partial loop invariant redundancy: load of constant 3

Motivating RISC-V example (2/2)

```
1  foo(a, v, n) {
2    k = 7; i = 0; r = 2f
3    x17 = float64[a+0] // previous occurrence
4    if (x17 <f r) { goto Exit }
5  Loop:
6    if (i >=ls n) { goto Exit }
7    x16 = float64[a+8] // a[1] (unsafe)
8    if (r >=f x16) {
9      x14 = float64[a+0]; // safe to eliminate
10     r = r -f x14 }
11   else { x15 = 3f; r = r *f x15 } // PRE
12   x13 = i <<l 3 // SR (addressing)
13   x12 = v +l x13 // SR (in sequence)
14   x10 = int64[x12+0]
15   x11 = i *l k // SR
16   x9 = x10 -l x11
17   x8 = floatoflong(x9)
18   r = r +f x8
19   i = i +l 4
20   goto Loop
21 Exit: return r }
```

What is needed?

- partial loop invariant redundancy: load of constant 3
- redundant load of
  - a[0] (available before the loop)
  - a[1] (only in the loop)

Motivating RISC-V example (2/2)

```
1  foo(a, v, n) {
2    k = 7; i = 0; r = 2f
3    x17 = float64[a+0] // previous occurrence
4    if (x17 <f r) { goto Exit }
5  Loop:
6    if (i >=ls n) { goto Exit }
7    x16 = float64[a+8] // a[1] (unsafe)
8    if (r >=f x16) {
9      x14 = float64[a+0]; // safe to eliminate
10     r = r -f x14 }
11   else { x15 = 3f; r = r *f x15 } // PRE
12   x13 = i <<l 3 // SR (addressing)
13   x12 = v +l x13 // SR (in sequence)
14   x10 = int64[x12+0]
15   x11 = i *l k // SR
16   x9 = x10 -l x11
17   x8 = floatoflong(x9)
18   r = r +f x8
19   i = i +l 4
20   goto Loop
21 Exit: return r }
```

What is needed?

- partial loop invariant redundancy: load of constant 3
- redundant load of
  - `a[0]` (available before the loop)
  - `a[1]` (only in the loop)
- Strength reduction of
  - source multiplication `k * i`
  - addressing calculation for `v[i]`
    $\rightarrow$ was **decomposed by instruction selection!**

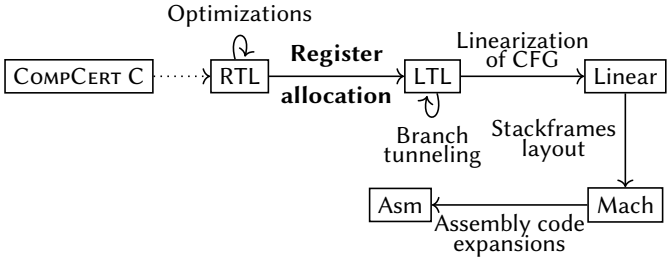
Motivating RISC-V example (2/2)

```
1  foo(a, v, n) {
2    k = 7; i = 0; r = 2f
3    x17 = float64[a+0] // previous occurrence
4    if (x17 <f r) { goto Exit }
5  Loop:
6    if (i >=ls n) { goto Exit }
7    x16 = float64[a+8] // a[1] (unsafe)
8    if (r >=f x16) {
9      x14 = float64[a+0]; // safe to eliminate
10     r = r -f x14 }
11   else { x15 = 3f; r = r *f x15 } // PRE
12   x13 = i <<l 3 // SR (addressing)
13   x12 = v +l x13 // SR (in sequence)
14   x10 = int64[x12+0]
15   x11 = i *l k // SR
16   x9 = x10 -l x11
17   x8 = floatoflong(x9)
18   r = r +f x8
19   i = i +l 4
20   goto Loop
21 Exit: return r }
```

What is needed?

- partial loop invariant redundancy: load of constant 3
- redundant load of
  - `a[0]` (available before the loop)
  - `a[1]` (only in the loop)
- Strength reduction of
  - source multiplication `k * i`
  - addressing calculation for `v[i]`
    $\rightarrow$ was **decomposed by instruction selection!**
- + possibility to schedule some instructions in a better way
  (not explained in this presentation)

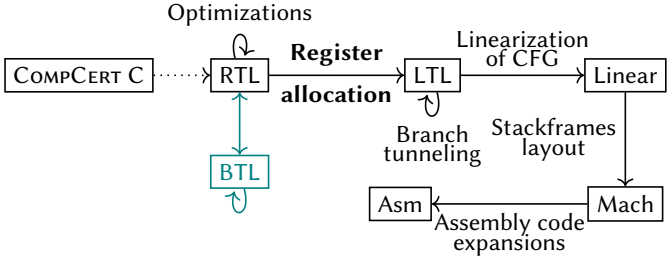
Overview of my contributions in Chamois CompCert



Black: original CompCert passes

Overview of my contributions in Chamois  CompCert

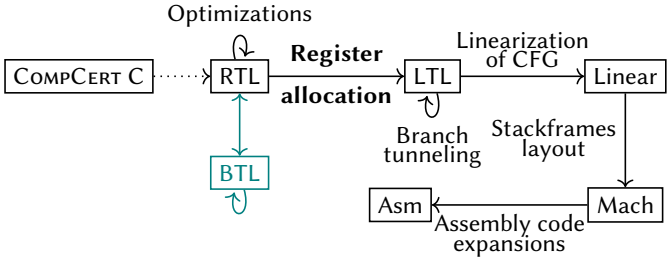
- Block Transfer Language IR



Black: original CompCert passes
Teal: All (AArch64+ARMv7+RISC-V+KVX+PPC+x86)

Overview of my contributions in Chamois  CompCert

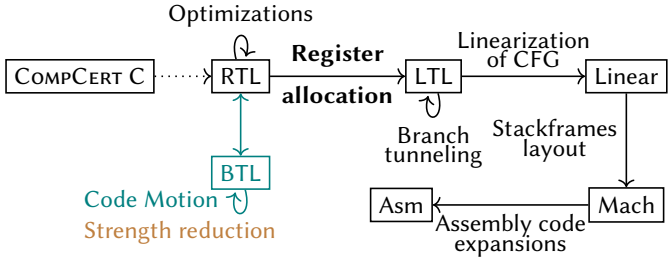
- Block Transfer Language IR
- Intra-procedural, defensive Symbolic Execution framework



Black: original CompCert passes
Teal: All (AArch64+ARMv7+RISC-V+KVX+PPC+x86)

Overview of my contributions in Chamois CompCert

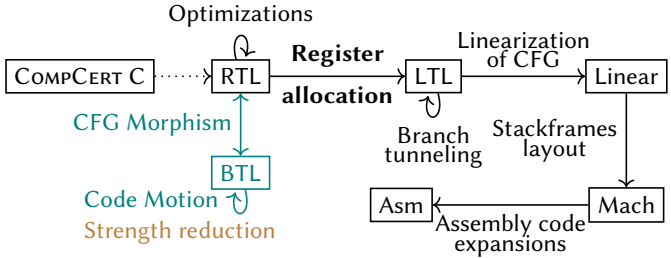
- Block Transfer Language IR
- Intra-procedural, defensive Symbolic Execution framework
- CM + SR: Lazy Code Transformations algorithm



Black: original CompCert passes
Teal: All (AArch64+ARMv7+RISC-V+KVX+PPC+x86)
Brown:RISC-V only

Overview of my contributions in Chamois CompCert

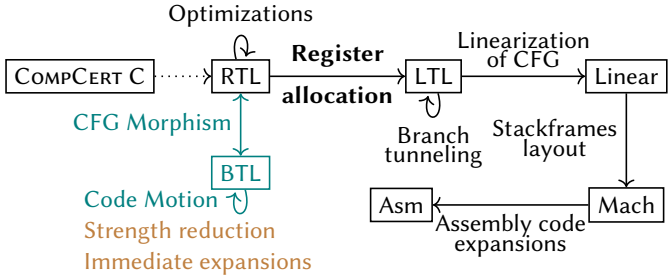
- Block Transfer Language IR
- Intra-procedural, defensive Symbolic Execution framework
- CM + SR: Lazy Code Transformations algorithm
- Control Flow Graph Morphism validator



Black: original CompCert passes
Teal: All (AArch64+ARMv7+RISC-V+KVX+PPC+x86)
Brown:RISC-V only

Overview of my contributions in Chamois  CompCert

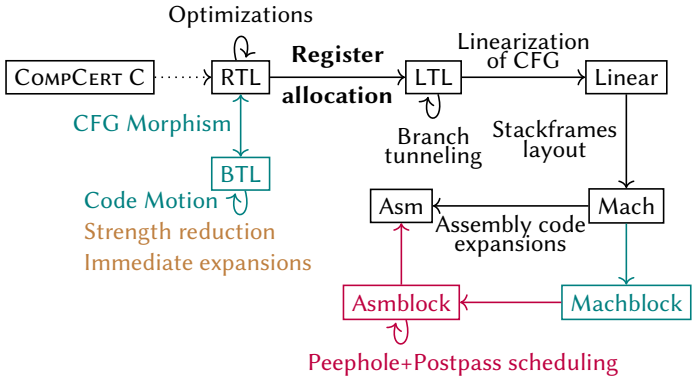
- Block Transfer Language IR
- Intra-procedural, defensive Symbolic Execution framework
- CM + SR: Lazy Code Transformations algorithm
- Control Flow Graph Morphism validator
- RISC-V expansion engine



Black: original CompCert passes
Teal: All (AArch64+ARMv7+RISC-V+KVX+PPC+x86)
Brown:RISC-V only

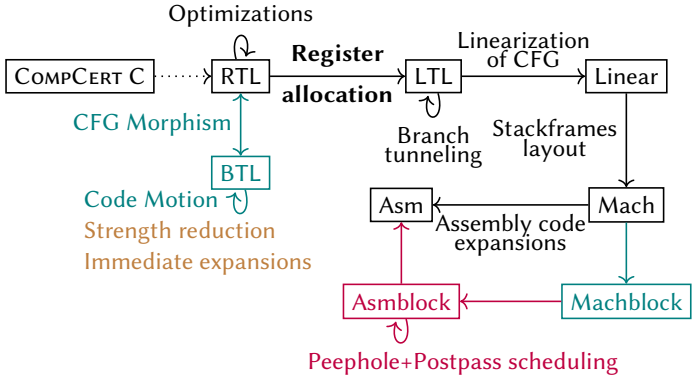
Overview of my contributions in Chamois  CompCert

- Block Transfer Language IR
- Intra-procedural, defensive Symbolic Execution framework
- CM + SR: Lazy Code Transformations algorithm
- Control Flow Graph Morphism validator
- RISC-V expansion engine
- Port of the KVX postpass scheduler to AArch64 + Peephole optimizer



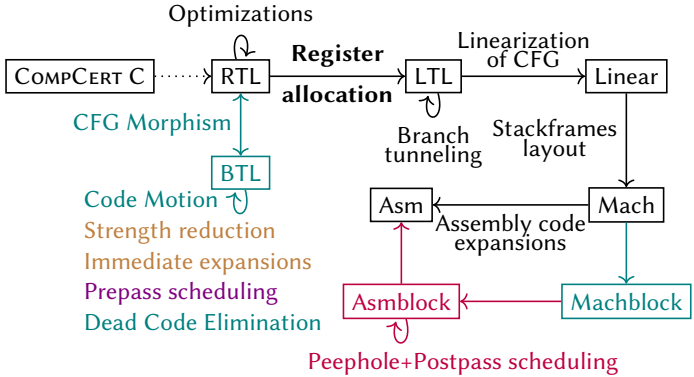
Overview of my contributions in Chamois CompCert

- Block Transfer Language IR
- Intra-procedural, defensive Symbolic Execution framework
- CM + SR: Lazy Code Transformations algorithm
- Control Flow Graph Morphism validator
- RISC-V expansion engine
- Port of the KVX postpass scheduler to AArch64 + Peephole optimizer
- Experimental evaluation framework



Overview of my contributions in Chamois CompCert

- Block Transfer Language IR
- Intra-procedural, defensive Symbolic Execution framework
- CM + SR: Lazy Code Transformations algorithm
- Control Flow Graph Morphism validator
- RISC-V expansion engine
- Port of the KVX postpass scheduler to AArch64 + Peephole optimizer
- Experimental evaluation framework
- Extension of [Six et al. 2022]'s superblock prepass scheduling



Lazy Code Motion (LCM) & Lazy Strength Reduction (LSR)

[Knoop, Rüthing and Steffen 1992-1995]

Lazy Code Motion (LCM) & Lazy Strength Reduction (LSR)

[Knoop, Rüthing and Steffen 1992-1995]

- **Intra-procedural**, data-flow algorithms:
  aim at **computational optimality** with minimal impact on **register pressure** (liverange)

Lazy Code Motion (LCM) & Lazy Strength Reduction (LSR)

[Knoop, Rüthing and Steffen 1992-1995]

- **Intra-procedural**, data-flow algorithms:
  aim at **computational optimality** with minimal impact on **register pressure** (liverange)
- **LCM:** moves operations and loads (all platforms)

Lazy Code Motion (LCM) & Lazy Strength Reduction (LSR)

[Knoop, Rüthing and Steffen 1992-1995]

- **Intra-procedural**, data-flow algorithms:
  aim at **computational optimality** with minimal impact on **register pressure** (liverange)
- **LCM:** moves operations and loads (all platforms)
- **LSR:** reduces multiplications with a constant (RISC-V only)

Lazy Code Motion (LCM) & Lazy Strength Reduction (LSR)

[Knoop, Rüthing and Steffen 1992-1995]

- **Intra-procedural**, data-flow algorithms:
  aim at **computational optimality** with minimal impact on **register pressure** (liverange)
- **LCM:** moves operations and loads (all platforms)
- **LSR:** reduces multiplications with a constant (RISC-V only)

Goals

- A "**Lazy Code Transformations**" (LCT) algorithm combining LCM & LSR
- Producing **hints** to guide the symbolic execution validator

Lazy Code Motion (LCM) & Lazy Strength Reduction (LSR)

[Knoop, Rüthing and Steffen 1992-1995]

- **Intra-procedural**, data-flow algorithms:
  aim at **computational optimality** with minimal impact on **register pressure** (liverange)
- **LCM:** moves operations and loads (all platforms)
- **LSR:** reduces multiplications with a constant (RISC-V only)

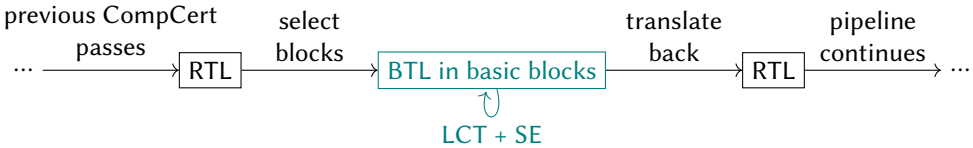
Goals

- A "**Lazy Code Transformations**" (LCT) algorithm combining LCM & LSR
- Producing **hints** to guide the symbolic execution validator
- An efficient OCaml implementation operating over BTL in **basic blocks** (1 entry, 1 exit)

Why LCM & LSR? Data-flow algorithms fit well with
block structure and invariant inference

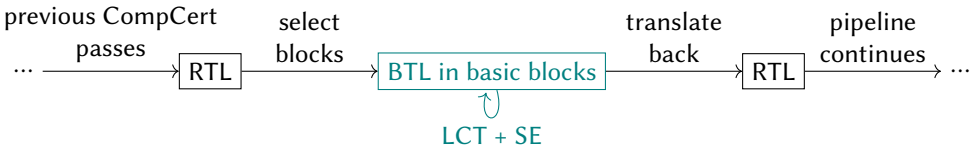
How the “Lazy Code Transformations” algorithm is applied

LCT is **untrusted** → it was co-designed with defensive validation by Symbolic Execution:



How the “Lazy Code Transformations” algorithm is applied

LCT is **untrusted** → it was co-designed with defensive validation by Symbolic Execution:

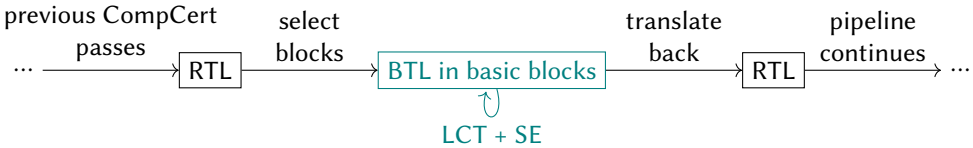


LCT step-by-step

1. **Joining (and critical) edges** are split with synthetic (empty) nodes
   → this is needed for data-flow fixed points + code motion opportunities in BTL

How the “Lazy Code Transformations” algorithm is applied

LCT is **untrusted** → it was co-designed with defensive validation by Symbolic Execution:

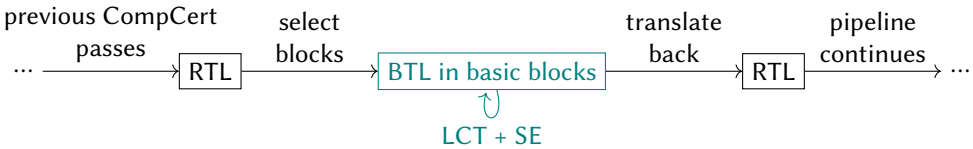


LCT step-by-step

1. **Joining (and critical) edges** are split with synthetic (empty) nodes
   → this is needed for data-flow fixed points + code motion opportunities in BTL
2. **Equation systems are solved** (data-flow: 4 for code motion + 3 for strength reduction; and a few non data-flow computations)

How the “Lazy Code Transformations” algorithm is applied

LCT is **untrusted** → it was co-designed with defensive validation by Symbolic Execution:

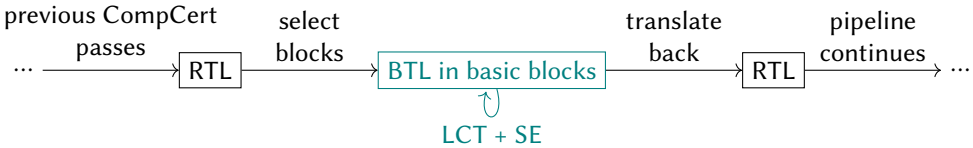


LCT step-by-step

1. **Joining (and critical) edges** are split with synthetic (empty) nodes
   → this is needed for data-flow fixed points + code motion opportunities in BTL
2. **Equation systems are solved** (data-flow: 4 for code motion + 3 for strength reduction; and a few non data-flow computations)
3. Control Flow Graph is **rewritten**

How the “Lazy Code Transformations” algorithm is applied

LCT is **untrusted** → it was co-designed with defensive validation by Symbolic Execution:



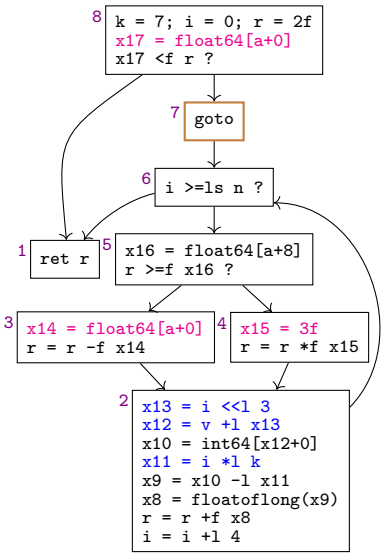
LCT step-by-step

1. **Joining (and critical) edges** are split with synthetic (empty) nodes
   → this is needed for data-flow fixed points + code motion opportunities in BTL
2. **Equation systems are solved** (data-flow: 4 for code motion + 3 for strength reduction; and a few non data-flow computations)
3. Control Flow Graph is **rewritten**
4. Invariant are **inferred** from equation results

On our example: partitioning, synthetic nodes, and candidates

```
1  double foo(double *a, long *v, long n) {
2    long k = 7; long i = 0;
3    double r = 2;
4    if (a[0] < 2) return 2;
5    for(; i < n; i += 4) {
6      if (r >= a[1]) r -= a[0];
7      else r *= 3;
8      r += v[i] - k * i;
9    }
10   return r;
11 }
```

On our example: partitioning, synthetic nodes, and candidates



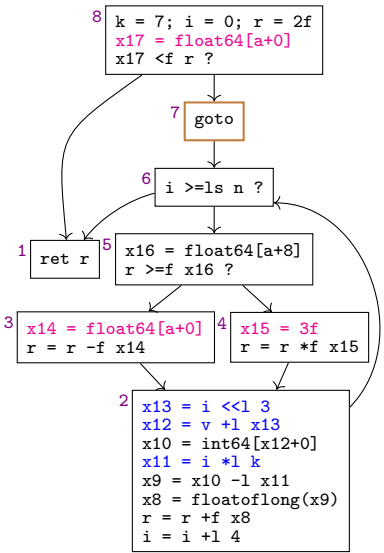
Legend:

Numbering (post-order)

Synthetic nodes

```
1  double foo(double *a, long *v, long n) {
2    long k = 7; long i = 0;
3    double r = 2;
4    if (a[0] < 2) return 2;
5    for(; i < n; i += 4) {
6      if (r >= a[1]) r -= a[0];
7      else r *= 3;
8      r += v[i] - k * i;
9    }
10   return r;
11 }
```

On our example: partitioning, synthetic nodes, and candidates



Legend:

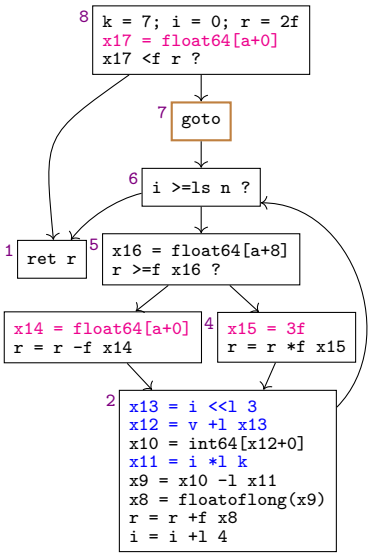
Numbering
(post-order)

Synthetic nodes

Candidates:
- Code Motion
- Strength Reduction

```
1  double foo(double *a, long *v, long n) {
2    long k = 7; long i = 0;
3    double r = 2;
4    if (a[0] < 2) return 2;
5    for(; i < n; i += 4) {
6      if (r >= a[1]) r -= a[0];
7      else r *= 3;
8      r += v[i] - k * i;
9    }
10   return r;
11 }
```

On our example: partitioning, synthetic nodes, and candidates



Legend:

Numbering (post-order)

Synthetic nodes

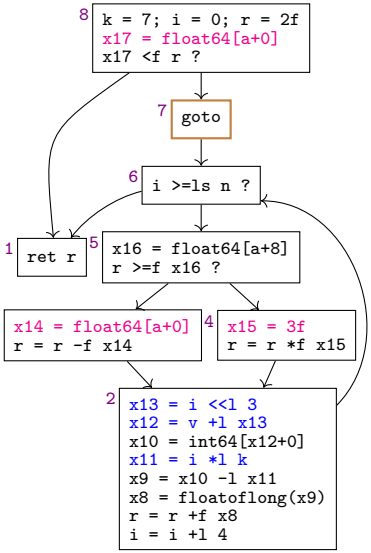
Candidates:
- Code Motion
- Strength Reduction

```
1  double foo(double *a, long *v, long n) {
2    long k = 7; long i = 0;
3    double r = 2;
4    if (a[0] < 2) return 2;
5    for(; i < n; i += 4) {
6      if (r >= a[1]) r -= a[0];
7      else r *= 3;
8      r += v[i] - k * i;
9    }
10   return r;
11 }
```

Desirable adjustments

- Restriction: our validator cannot **anticipate** potentially trapping instructions (e.g. load `a[1]`)

On our example: partitioning, synthetic nodes, and candidates



Legend:

Numbering (post-order)

Synthetic nodes

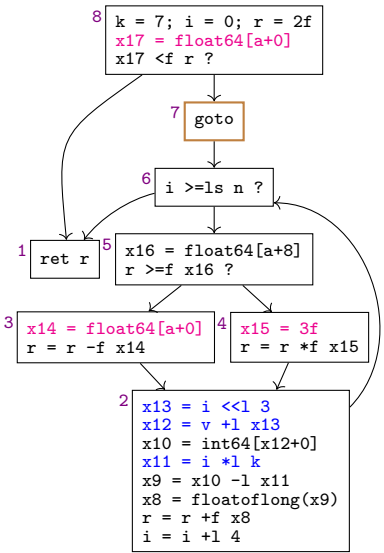
Candidates:
- Code Motion
- Strength Reduction

```
1  double foo(double *a, long *v, long n) {
2    long k = 7; long i = 0;
3    double r = 2;
4    if (a[0] < 2) return 2;
5    for(; i < n; i += 4) {
6      if (r >= a[1]) r -= a[0];
7      else r *= 3;
8      r += v[i] - k * i;
9    }
10   return r;
11 }
```

Desirable adjustments

- Restriction: our validator cannot **anticipate** potentially trapping instructions (e.g. load `a[1]`)
- Extension: the original algorithms would be unable to reduce nested sequences

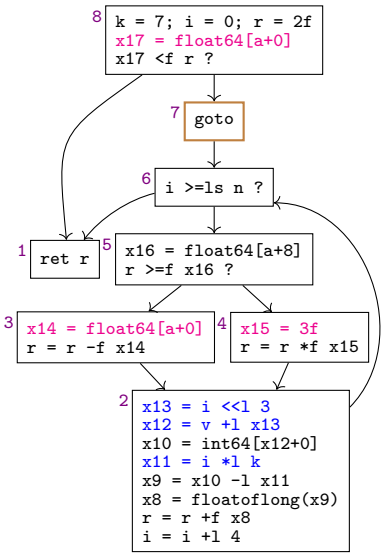
Optimizing candidates (1/2)



Candidates treated **topologically**

1st candidate: the load of `a[0]`

Optimizing candidates (1/2)



Candidates treated **topologically**

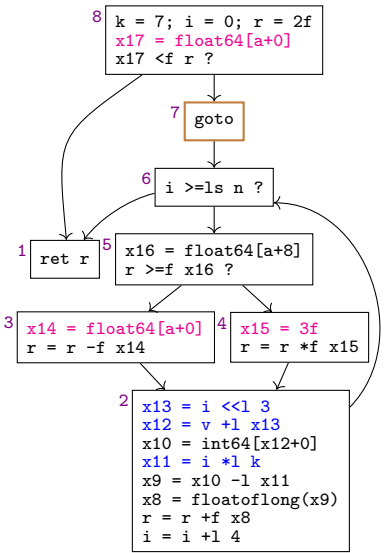
1st candidate: the load of `a[0]`

Restriction for potentially trapping instruction

As loads may trap, LCT ensures **two important conditions:**

1. a previous occurrence exists
2. the previous occurrence is available on every path leading to the target redundancy

Optimizing candidates (1/2)



Candidates treated **topologically**

1st candidate: the load of `a[0]`

Restriction for potentially trapping instruction

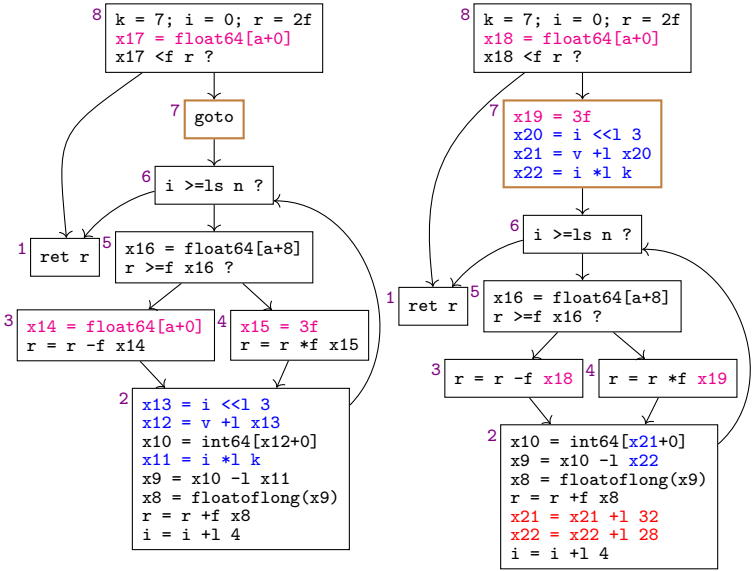
As loads may trap, LCT ensures **two important conditions:**

1. a previous occurrence exists
2. the previous occurrence is available on every path leading to the target redundancy

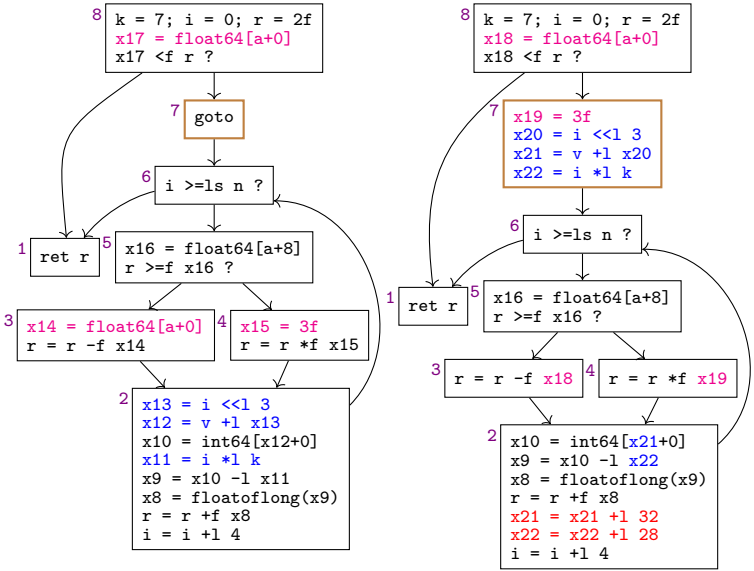
Extension for nested sequences

1. Introduction of **fresh pseudoregister**
2. Local substitution of pseudoregisters
3. Insertion of a move at block exits
   this move is then removed by dead code elimination if useless

Optimizing candidates (2/2)

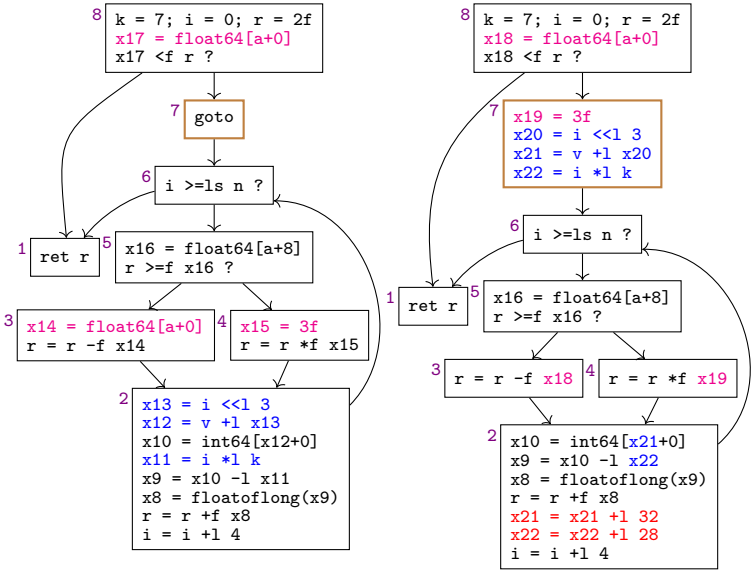


Optimizing candidates (2/2)



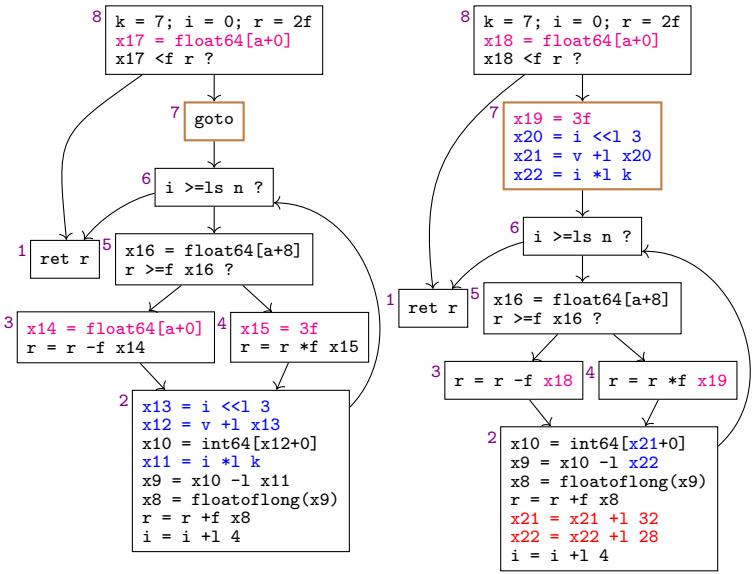
- Redundant load of `a[0]` **eliminated**

Optimizing candidates (2/2)



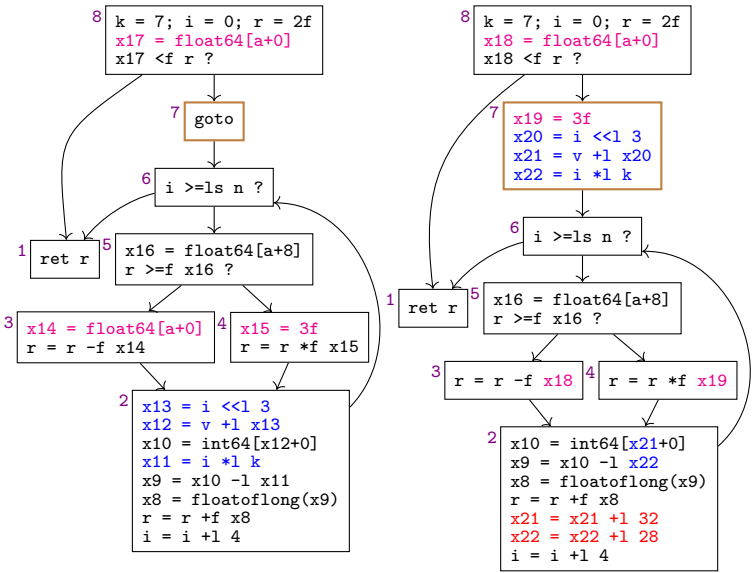
- Redundant load of `a[0]` **eliminated**
- Load of immediate constant `3f` **anticipated**

Optimizing candidates (2/2)



- Redundant load of `a[0]` **eliminated**
- Load of immediate constant `3f` **anticipated**
- Array addressing sequence for `v[i]` **reduced** with compensation $i << 3 = 4 \times 8 = 32$
- Multiplication `i * k` **reduced** with compensation $i \times k = 4 \times 7 = 28$

Optimizing candidates (2/2)



- Redundant load of `a[0]` **eliminated**
- Load of immediate constant `3f` **anticipated**
- Array addressing sequence for `v[i]` **reduced** with compensation $i << 3 = 4 \times 8 = 32$
- Multiplication `i * k` **reduced** with compensation $i \times k = 4 \times 7 = 28$

Gain $\sim$ **8 cycles/iteration** on U74 RISC-V!
(49 to 41 cycles, 16% reduction)

Quick summary on Lazy Code Transformations

- For **potentially trapping** instructions (e.g. loads + arch specific operations);
  we adapted Lazy Code Motion to restrict it with stronger conditions
- To support **instruction sequences**;
  we proposed a rewriting procedure by substitution of fresh variables

Quick summary on Lazy Code Transformations

- For **potentially trapping** instructions (e.g. loads + arch specific operations);
  we adapted Lazy Code Motion to restrict it with stronger conditions
- To support **instruction sequences**;
  we proposed a rewriting procedure by substitution of fresh variables
- To generalize Lazy Strength Reduction on **basic blocks**;
  we had to adapt data-flow equations of [Knoop et al. 1993]

Quick summary on Lazy Code Transformations

- For **potentially trapping** instructions (e.g. loads + arch specific operations);
  we adapted Lazy Code Motion to restrict it with stronger conditions
- To support **instruction sequences**;
  we proposed a rewriting procedure by substitution of fresh variables
- To generalize Lazy Strength Reduction on **basic blocks**;
  we had to adapt data-flow equations of [Knoop et al. 1993]
- Lastly, LCT features an **invariant inference** procedure reusing existing analyses

Quick summary on Lazy Code Transformations

- For **potentially trapping** instructions (e.g. loads + arch specific operations);
  we adapted Lazy Code Motion to restrict it with stronger conditions
- To support **instruction sequences**;
  we proposed a rewriting procedure by substitution of fresh variables
- To generalize Lazy Strength Reduction on **basic blocks**;
  we had to adapt data-flow equations of [Knoop et al. 1993]
- Lastly, LCT features an **invariant inference** procedure reusing existing analyses
- Now, two questions arise:
  1. How to **defensively validate** LCT by Symbolic Execution + Invariants?
  2. How can we eliminate **non-available loads** like `a[1]` in the example?

Block Transfer Language & Blockstep semantics

Partitioning the code into **loop-free blocks** (with a single entry point from the outside):

- Avoids loops in symbolic execution
- Allows for block scoped optimizations (e.g. instruction scheduling)
- Stays compatible with (basic) block based algorithms

Block Transfer Language & Blockstep semantics

Partitioning the code into **loop-free blocks** (with a single entry point from the outside):

- Avoids loops in symbolic execution
- Allows for block scoped optimizations (e.g. instruction scheduling)
- Stays compatible with (basic) block based algorithms

Block Transfer Language: Control flow graph of syntactically defined blocks

Block Transfer Language & Blockstep semantics

Partitioning the code into **loop-free blocks** (with a single entry point from the outside):

- Avoids loops in symbolic execution
- Allows for block scoped optimizations (e.g. instruction scheduling)
- Stays compatible with (basic) block based algorithms

Block Transfer Language: Control flow graph of syntactically defined blocks

Blockstep $\triangleq$ execution from the entry point to one exit point (at most one non-silent event)

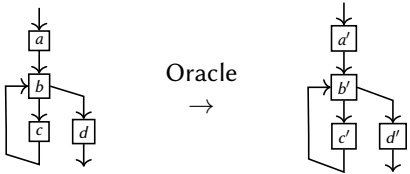
To relate the BTL blockstep semantics with the RTL **smallstep** semantics,
we want **"local" blockstep simulations** to ensure a "global" simulation!

It suffices that blockstep semantics bisimulates the standard smallstep semantics.

Principle of symbolic execution

[King 1976; Samet 1976]

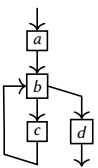
Control flow graph of blocks:



Principle of symbolic execution

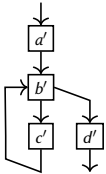
[King 1976; Samet 1976]

Control flow graph of blocks:



Oracle

$\rightarrow$



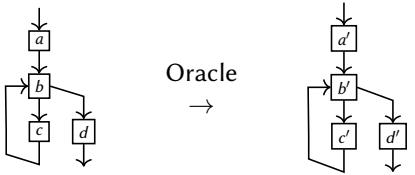
For each pair of block $(B_{\mathcal{S}}, B_{\mathcal{T}})$ in $[(a, a'), (b, b'), ...]$,
compare symbolic states $(\delta_{\mathcal{S}}, \delta_{\mathcal{T}})$
from their symbolic execution with $\xi : \mathit{block} \rightarrow \delta$.

With $\xi(B_{\mathcal{S}}) = \delta_{\mathcal{S}}$ and $\xi(B_{\mathcal{T}}) = \delta_{\mathcal{T}}$,
does $\delta_{\mathcal{S}} \equiv \delta_{\mathcal{T}}$ hold?

Principle of symbolic execution

[King 1976; Samet 1976]

Control flow graph of blocks:



For each pair of block $(B_\mathcal{S}, B_\mathcal{T})$ in $[(a, a'), (b, b'), ...]$,
compare symbolic states $(\delta_\mathcal{S}, \delta_\mathcal{T})$
from their symbolic execution with $\xi : \mathit{block} \to \delta$.

With $\xi(B_\mathcal{S}) = \delta_\mathcal{S}$ and $\xi(B_\mathcal{T}) = \delta_\mathcal{T}$,
does $\delta_\mathcal{S} \equiv \delta_\mathcal{T}$ hold?

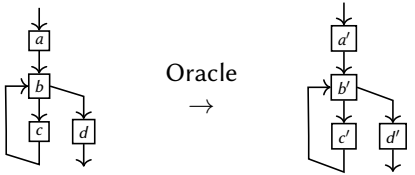
In CompCert: Formally verified superblock scheduling [Six et al. 2022]

Validated by **Symbolic Execution (SE)**, but limited to superblock scope;
→ no support for intra-procedural transformations

Principle of symbolic execution

[King 1976; Samet 1976]

Control flow graph of blocks:



For each pair of block (B_S, B_T) in $[(a, a'), (b, b'), ...]$,
compare symbolic states (δ_S, δ_T)
from their symbolic execution with $\xi : block \to \delta$.

With $\xi(B_S) = \delta_S$ and $\xi(B_T) = \delta_T$,
does $\delta_S \equiv \delta_T$ hold?

In CompCert: Formally verified superblock scheduling [Six et al. 2022]

Validated by **Symbolic Execution (SE)**, but limited to superblock scope;
→ no support for intra-procedural transformations

Advantages: **generic** validation method + **scales** well + supports **normalized rewrites**

Intra-Block simulation: basic block example

Assume a proven **rewriting rule** $\forall x, x \times 2 = x + x$.

Intra-Block simulation: basic block example

Assume a proven **rewriting rule** $\forall x, x \times 2 = x + x$.

$$
\begin{array}{ll}
(B_1) & r_3 \coloneqq r_1 + r_2; \\
& r_3 \coloneqq r_3 \times 2; \\
& r_4 \coloneqq \texttt{load}[m, r_3]; \\
& r_4 \coloneqq r_2 \times r_2;
\end{array}
\qquad
\begin{array}{ll}
(B_2) & r_4 \coloneqq r_2 \times r_2; \\
& r_3 \coloneqq r_1 + r_2; \\
& r_3 \coloneqq r_3 + r_3;
\end{array}
$$

Intra-Block simulation: basic block example

Assume a proven **rewriting rule** $\forall x, x \times 2 = x + x$.

$$
\begin{array}{lll}
(B_1) & r_3 \coloneqq r_1 + r_2; \\
& r_3 \coloneqq r_3 \times 2; \\
& r_4 \coloneqq \texttt{load}[m, r_3]; \\
& r_4 \coloneqq r_2 \times r_2;
\end{array}
\qquad
\begin{array}{lll}
(B_2) & r_4 \coloneqq r_2 \times r_2; \\
& r_3 \coloneqq r_1 + r_2; \\
& r_3 \coloneqq r_3 + r_3;
\end{array}
$$

Both B_1 and B_2 lead to the **same parallel assignment** (of live registers):

$$r_3 \coloneqq (r_1 + r_2) + (r_1 + r_2) \parallel r_4 \coloneqq r_2 \times r_2$$

Intra-Block simulation: basic block example

Assume a proven **rewriting rule** $\forall x, x \times 2 = x + x$.

$$
\begin{array}{rl|rl}
(B_1) & r_3 \coloneqq r_1 + r_2; & (B_2) & r_4 \coloneqq r_2 \times r_2; \\
& r_3 \coloneqq r_3 \times 2; & & r_3 \coloneqq r_1 + r_2; \\
& r_4 \coloneqq \texttt{load}[m, r_3]; & & r_3 \coloneqq r_3 + r_3; \\
& r_4 \coloneqq r_2 \times r_2; & &
\end{array}
$$

Both B_1 and B_2 lead to the **same parallel assignment** (of live registers):

$$r_3 \coloneqq (r_1 + r_2) + (r_1 + r_2) \parallel r_4 \coloneqq r_2 \times r_2$$

B_2 simulates B_1, **but** B_1 simulates B_2 iff "$OK(\texttt{load}[m, r_3])$"
$\rightarrow B_1 \sim B_2$ precondition is stronger as we must not add any potential trap

Intra-Block simulation: basic block example

Assume a proven **rewriting rule** $\forall x, x \times 2 = x + x$.

$$
\begin{array}{lll}
(B_1) & r_3 \coloneqq r_1 + r_2; & \\
& r_3 \coloneqq r_3 \times 2; & \\
& r_4 \coloneqq \texttt{load}[m, r_3]; & \\
& r_4 \coloneqq r_2 \times r_2; &
\end{array}
\qquad
\begin{array}{lll}
(B_2) & r_4 \coloneqq r_2 \times r_2; & \\
& r_3 \coloneqq r_1 + r_2; & \\
& r_3 \coloneqq r_3 + r_3; &
\end{array}
$$

Both B_1 and B_2 lead to the **same parallel assignment** (of live registers):

$$r_3 \coloneqq (r_1 + r_2) + (r_1 + r_2) \parallel r_4 \coloneqq r_2 \times r_2$$

B_2 simulates B_1, **but** B_1 simulates B_2 iff "$OK(\texttt{load}[m, r_3])$"
$\rightarrow B_1 \sim B_2$ precondition is stronger as we must not add any potential trap

However... **term duplication** makes structural comparison **exponential** (e.g. "$r_1 + r_2$")!
Solution of [Six et al. 2020]: **hash-consing**, i.e. memoize subterms + pointer equalities

Aggregated block-by-block simulations, in practice

Symbolic states: $\delta \triangleq (\mu, \vec{\sigma}, \mathcal{R})$ (memory, precondition, registers state)

Aggregated block-by-block simulations, in practice

Symbolic states: $\delta \triangleq (\mu, \vec{\sigma}, \mathcal{R})$ (memory, precondition, registers state)

Memory $\mu ::= \texttt{Sinit} \mid \texttt{Sstore}(\mu_{old}, chk, addr, \vec{\sigma}, src)$

Aggregated block-by-block simulations, in practice

Symbolic states: $\delta \triangleq (\mu, \vec{\sigma}, \mathcal{R})$ (memory, precondition, registers state)

Memory $\quad \mu ::= \texttt{Sinit} \mid \texttt{Sstore}(\mu_{old}, \mathit{chk}, \mathit{addr}, \vec{\sigma}, \mathit{src})$

Values $\quad \sigma ::= \texttt{Sinput}(r) \mid \texttt{Sop}(\mathit{op}, \vec{\sigma}) \mid \texttt{Sload}(\mu, \mathit{trap}, \mathit{chk}, \mathit{addr}, \vec{\sigma}) \mid \ldots$

Aggregated block-by-block simulations, in practice

Symbolic states: $\delta \triangleq (\mu, \vec{\sigma}, \mathcal{R})$ (memory, precondition, registers state)

Memory	$\mu ::= \texttt{Sinit} \mid \texttt{Sstore}(\mu_{old}, \mathit{chk}, \mathit{addr}, \vec{\sigma}, \mathit{src})$
Values	$\sigma ::= \texttt{Sinput}(r) \mid \texttt{Sop}(\mathit{op}, \vec{\sigma}) \mid \texttt{Sload}(\mu, \mathit{trap}, \mathit{chk}, \mathit{addr}, \vec{\sigma}) \mid \ldots$
Regset	$\mathcal{R} \triangleq r \mapsto \sigma$ a **finite map "register $\mapsto$ terms"** (parallel assignment)

Aggregated block-by-block simulations, in practice

Block shapes:: basic-blocks (1 entry, 1 exit), superblocks (1 entry, side-exits), extended (basic) blocks (trees without internal joins), loop-free blocks (directed acyclic graphs)

Symbolic states: $\delta \triangleq (\mu, \vec{\sigma}, \mathcal{R})$ (memory, precondition, registers state)

Memory	$\mu ::= \texttt{Sinit} \mid \texttt{Sstore}(\mu_{old}, chk, addr, \vec{\sigma}, src)$
Values	$\sigma ::= \texttt{Sinput}(r) \mid \texttt{Sop}(op, \vec{\sigma}) \mid \texttt{Sload}(\mu, trap, chk, addr, \vec{\sigma}) \mid \ldots$
Regset	$\mathcal{R} \triangleq r \mapsto \sigma$ a **finite map "register $\mapsto$ terms"** (parallel assignment)

According to the block shape, the resulting state is:

- a **single triplet** for basic-blocks;

Aggregated block-by-block simulations, in practice

Block shapes:: basic-blocks (1 entry, 1 exit), superblocks (1 entry, side-exits), extended (basic) blocks (trees without internal joins), loop-free blocks (directed acyclic graphs)

Symbolic states: $\delta \triangleq (\mu, \vec{\sigma}, \mathcal{R})$ (memory, precondition, registers state)

Memory	$\mu ::= \texttt{Sinit} \mid \texttt{Sstore}(\mu_{old}, chk, addr, \vec{\sigma}, src)$
Values	$\sigma ::= \texttt{Sinput}(r) \mid \texttt{Sop}(op, \vec{\sigma}) \mid \texttt{Sload}(\mu, trap, chk, addr, \vec{\sigma}) \mid \ldots$
Regset	$\mathcal{R} \triangleq r \mapsto \sigma$ a **finite map "register $\mapsto$ terms"** (parallel assignment)

According to the block shape, the resulting state is:

- a **single triplet** for basic-blocks;
- a **Binary Decision Diagram (BDD)** with triplets on leafs (=exits) in the general case.

The “basic” simulation test

Independently, for each pair of blocks,

Comparing symbolic states

The “basic” simulation test

Independently, for each pair of blocks,

Comparing symbolic states

Let $\delta_1 = (\mu_1, \vec{\sigma_1}, \mathcal{R}_1)$ and $\delta_2 = (\mu_2, \vec{\sigma_2}, \mathcal{R}_2)$, δ_2 simulates δ_1 iff:

$$\delta_1 \succeq \delta_2 \triangleq \mu_1 = \mu_2 \wedge \vec{\sigma_2} \subseteq \vec{\sigma_1} \wedge \mathcal{R}_1 \subseteq \mathcal{R}_2$$

The “basic” simulation test

Independently, for each pair of blocks,

Comparing symbolic states

Let $\delta_1 = (\mu_1, \vec{\sigma_1}, \mathcal{R}_1)$ and $\delta_2 = (\mu_2, \vec{\sigma_2}, \mathcal{R}_2)$, δ_2 simulates δ_1 iff:

$$\delta_1 \succeq \delta_2 \triangleq \mu_1 = \mu_2 \wedge \vec{\sigma_2} \subseteq \vec{\sigma_1} \wedge \mathcal{R}_1 \subseteq \mathcal{R}_2$$

$\rightarrow \delta_1$ *is at least as trapping as* δ_2 (we do not add any potential trap)

The “basic” simulation test

Independently, for each pair of blocks,

Comparing symbolic states

Let $\delta_1 = (\mu_1, \vec{\sigma_1}, \mathcal{R}_1)$ and $\delta_2 = (\mu_2, \vec{\sigma_2}, \mathcal{R}_2)$, δ_2 simulates δ_1 iff:

$$\delta_1 \succeq \delta_2 \triangleq \mu_1 = \mu_2 \wedge \vec{\sigma_2} \subseteq \vec{\sigma_1} \wedge \mathcal{R}_1 \subseteq \mathcal{R}_2$$

$\rightarrow \delta_1$ *is at least as trapping as* δ_2 (we do not add any potential trap)
$\rightarrow \delta_2$ *may define fresh variables*

The “basic” simulation test

Independently, for each pair of blocks,

Comparing symbolic states

Let $\delta_1 = (\mu_1, \vec{\sigma_1}, \mathcal{R}_1)$ and $\delta_2 = (\mu_2, \vec{\sigma_2}, \mathcal{R}_2)$, δ_2 simulates δ_1 iff:

$$\delta_1 \succeq \delta_2 \triangleq \mu_1 = \mu_2 \wedge \vec{\sigma_2} \subseteq \vec{\sigma_1} \wedge \mathcal{R}_1 \subseteq \mathcal{R}_2$$

$\rightarrow$ δ_1 *is at least as trapping as* δ_2 (we do not add any potential trap)
$\rightarrow$ δ_2 *may define fresh variables*

Limitation: cannot **anticipate** potentially trapping instructions (e.g. loads)

The “basic” simulation test

Independently, for each pair of blocks,

Comparing symbolic states

Let $\delta_1 = (\mu_1, \vec{\sigma_1}, \mathcal{R}_1)$ and $\delta_2 = (\mu_2, \vec{\sigma_2}, \mathcal{R}_2)$, δ_2 simulates δ_1 iff:

$$\delta_1 \succeq \delta_2 \triangleq \mu_1 = \mu_2 \wedge \vec{\sigma_2} \subseteq \vec{\sigma_1} \wedge \mathcal{R}_1 \subseteq \mathcal{R}_2$$

$\rightarrow$ δ_1 *is at least as trapping as* δ_2 (we do not add any potential trap)
$\rightarrow$ δ_2 *may define fresh variables*

Limitation: cannot **anticipate** potentially trapping instructions (e.g. loads)
$\rightarrow$ solution: a prior **loop-peeling pass**.

The “basic” simulation test

Independently, for each pair of blocks,

Comparing symbolic states

Let $\delta_1 = (\mu_1, \vec{\sigma_1}, \mathcal{R}_1)$ and $\delta_2 = (\mu_2, \vec{\sigma_2}, \mathcal{R}_2)$, δ_2 simulates δ_1 iff:

$$\delta_1 \succeq \delta_2 \triangleq \mu_1 = \mu_2 \wedge \vec{\sigma_2} \subseteq \vec{\sigma_1} \wedge \mathcal{R}_1 \subseteq \mathcal{R}_2$$

$\rightarrow$ δ_1 *is at least as trapping as* δ_2 (we do not add any potential trap)
$\rightarrow$ δ_2 *may define fresh variables*

Limitation: cannot **anticipate** potentially trapping instructions (e.g. loads)
$\rightarrow$ solution: a prior **loop-peeling pass**.

Main problematic: extending the approach for **inter-block** (intra-procedural) transformations.

Generalizing this principle for inter-block transformations (1/2)

Idea:

1. Oracles infer and add **invariant annotations** to the target program
2. Symbolic simulation **defensively** validate invariants

$\rightarrow$ information propagation + consistency at global level

Generalizing this principle for inter-block transformations (1/2)

Idea:

1. Oracles infer and add **invariant annotations** to the target program
2. Symbolic simulation **defensively** validate invariants

$\rightarrow$ information propagation + consistency at global level

High-level overview

Each block is annotated with two types of invariants:

1. Gluing invariant ($\mathcal{G}$): **assigns target variables** by expressions of source variables
2. History invariant ($\mathcal{H}$): **assigns source variables** by expressions of source variables

Generalizing this principle for inter-block transformations (1/2)

Idea:

1. Oracles infer and add **invariant annotations** to the target program
2. Symbolic simulation **defensively** validate invariants

$\rightarrow$ information propagation + consistency at global level

High-level overview

Each block is annotated with two types of invariants:

1. Gluing invariant ($\mathcal{G}$): **assigns target variables** by expressions of source variables
2. History invariant ($\mathcal{H}$): **assigns source variables** by expressions of source variables

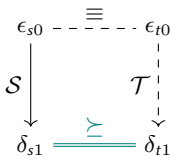
Each invariant is composed of:

- A sequence of **assignments**
- A set of **live variables** in the block (i.e. as trivial assignments "`x:=x`")

Generalizing this principle for inter-block transformations (2/2)

- $\epsilon \triangleq$ **empty** symbolic state

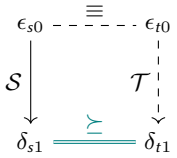
In [Six et al. 2022],
no relation between local simulations:
no anticipation possible!



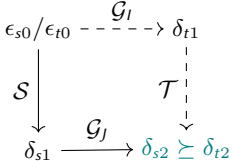
Generalizing this principle for inter-block transformations (2/2)

- $\epsilon \triangleq$ **empty** symbolic state
- $V_{\mathcal{S}}$, $V_{\mathcal{T}}$: sets of source/target **variables**; $\sigma[V]$: **symbolic expressions** of variables of V
- I, J subscripts: invariant of the **current/successors** blocks

In [Six et al. 2022],
no relation between local simulations:
no anticipation possible!



Anticipation of
(non-trapping) computations:
Gluing Invariants
($\mathcal{G}$: $V_{\mathcal{T}} \mapsto \sigma[V_{\mathcal{S}}]$).

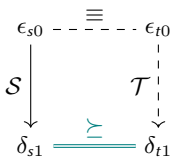


Still using the "$\succeq$" **comparison** on the target's output liveness (e.g. "$\succeq_{dom(\mathcal{G}_J)}$")

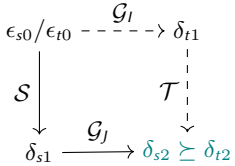
Generalizing this principle for inter-block transformations (2/2)

- $\epsilon \triangleq$ **empty** symbolic state
- $V_{\mathcal{S}}$, $V_{\mathcal{T}}$: sets of source/target **variables**; $\sigma[V]$: **symbolic expressions** of variables of V
- I, J subscripts: invariant of the **current/successors** blocks

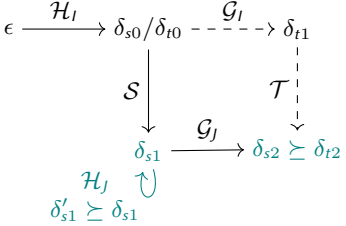
In [Six et al. 2022], **no relation** between local simulations: no anticipation possible!



Anticipation of (non-trapping) computations: **Gluing Invariants** ($\mathcal{G}$: $V_{\mathcal{T}} \mapsto \sigma[V_{\mathcal{S}}]$).

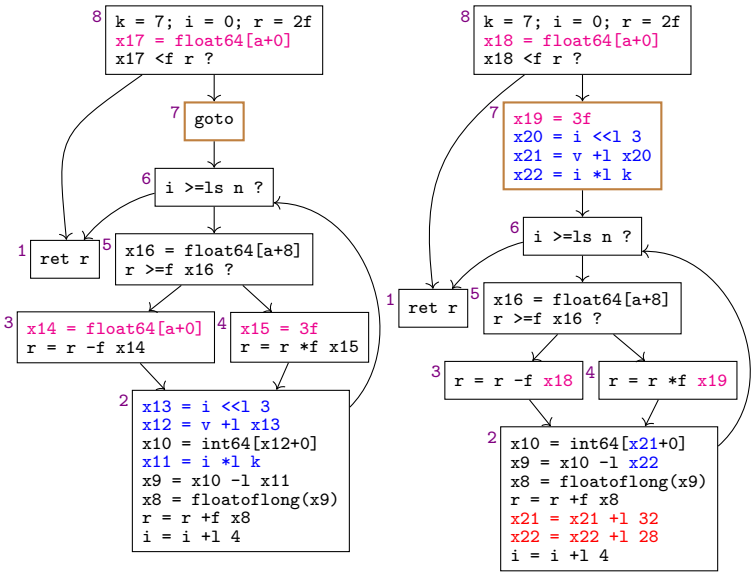


Sharing a common execution past: **History Invariants** ($\mathcal{H}$: $V_{\mathcal{S}} \mapsto \sigma[V_{\mathcal{S}}]$).

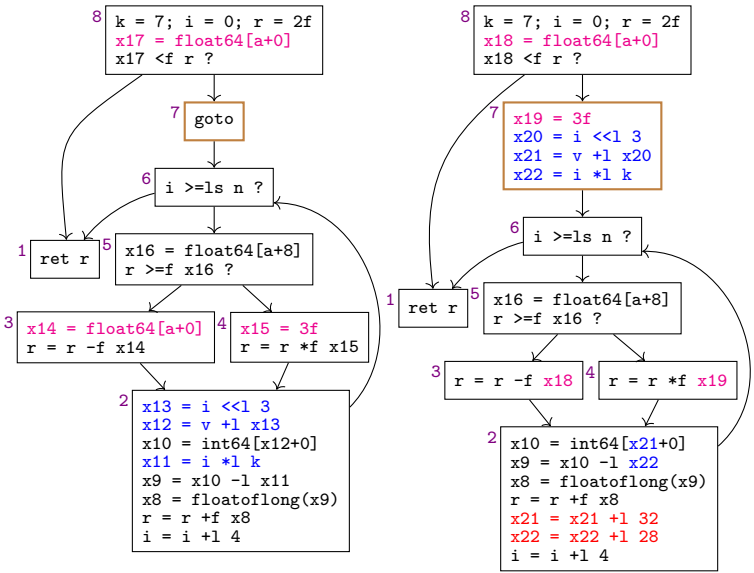


Still using the "$\succeq$" **comparison** on the target's output liveness (e.g. "$\succeq_{dom(\mathcal{G}_J)}$")

Validating Lazy Code Transformations on our example (1/3)



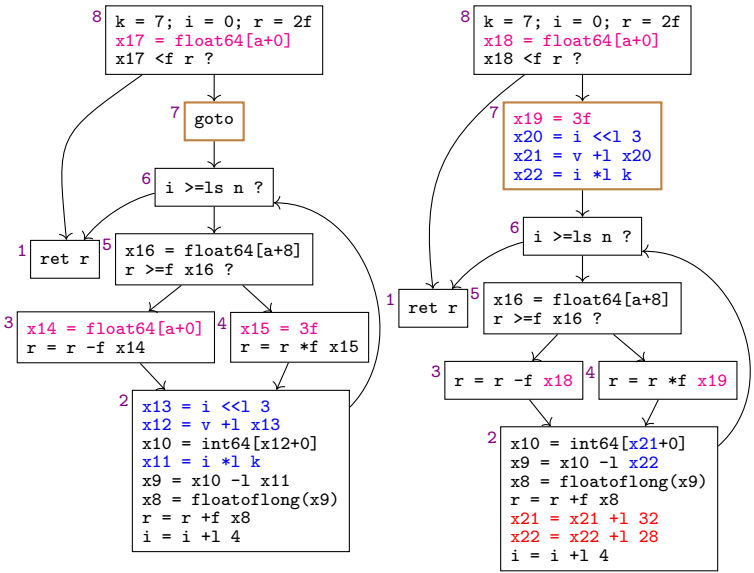
Validating Lazy Code Transformations on our example (1/3)



Entry with live variables only (block 8):

$\mathcal{G}$: `[ALIVE={a, v, n}]`

Validating Lazy Code Transformations on our example (1/3)



Entry with live variables only (block 8):

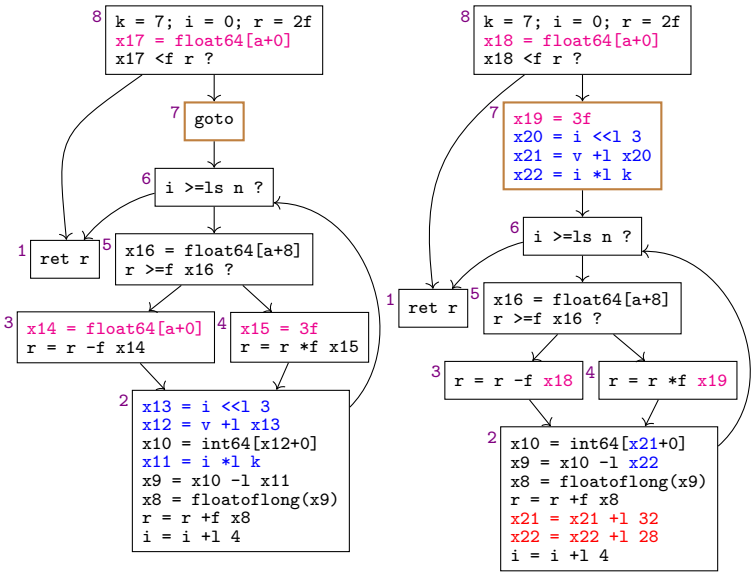
$\mathcal{G}$: `[ALIVE={a, v, n}]`

Old synthetic node (block 7):

$\mathcal{G}$: `[ALIVE={a, v, n, k, i, r}; x18:=float64[a+0]]`

$\mathcal{H}$: `[k:=7]`

Validating Lazy Code Transformations on our example (1/3)



Entry with live variables only (block 8):

$\mathcal{G}$: `[ALIVE={a, v, n}]`

Old synthetic node (block 7):

$\mathcal{G}$: `[ALIVE={a, v, n, k, i, r};`
`x18:=float64[a+0]]`

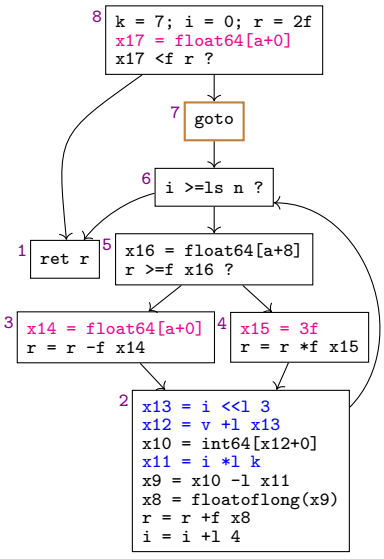
$\mathcal{H}$: `[k:=7]`

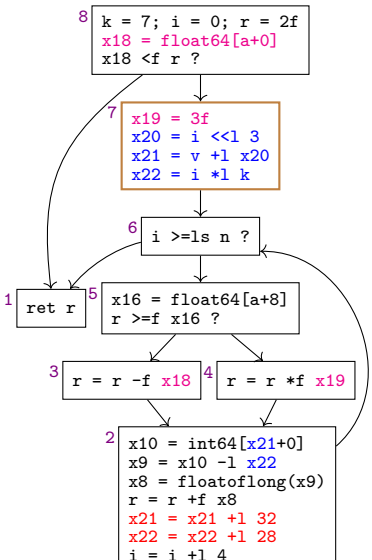
For all **loop blocks** (in $\{2, 3, 4, 5, 6\}$):

$\mathcal{G}$: `[ALIVE={a, n, i, r};`
`x18:=float64[a+0]; x19:=3f;`
`AUX/x20:=i <<l 3;`
`x21:=v +l x20; x22:=i *l k]`

$\mathcal{H}$: `[k:=7]`

Validating Lazy Code Transformations on our example (1/3)





Entry with live variables only (block 8):

$\mathcal{G}$: `[ALIVE={a, v, n}]`

Old synthetic node (block 7):

$\mathcal{G}$: `[ALIVE={a, v, n, k, i, r}; x18:=float64[a+0]]`

$\mathcal{H}$: `[k:=7]`

For all **loop blocks** (in $\{2, 3, 4, 5, 6\}$):

$\mathcal{G}$: `[ALIVE={a, n, i, r}; x18:=float64[a+0]; x19:=3f; AUX/x20:=i <<l 3; x21:=v +l x20; x22:=i *l k]`

$\mathcal{H}$: `[k:=7]`

Exit (block 1):

$\mathcal{G}$: `[ALIVE={r}]`

Validating Lazy Code Transformations on our example (2/3)

$\mathcal{G}_I$=input / $\mathcal{G}_J$=output, we have:

1. apply $\mathcal{H}$ (same for input/output here);
2. compare with $(\mathcal{S} \triangleright \mathcal{G}_J) \succeq_{dom(\mathcal{G}_J)} (\mathcal{G}_I \triangleright \mathcal{T})$

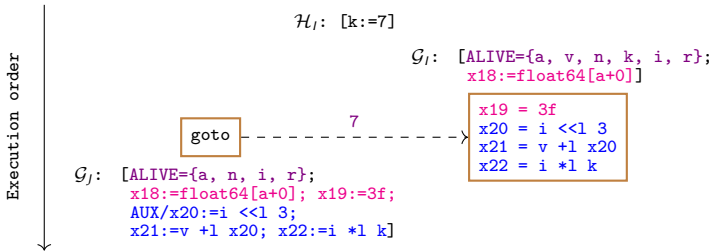
Example 1: synthetic node 7 (**anticipate** reduced operations)

Validating Lazy Code Transformations on our example (2/3)

$\mathcal{G}_I$=input / $\mathcal{G}_J$=output, we have:

1. apply $\mathcal{H}$ (same for input/output here);
2. compare with $(\mathcal{S} \triangleright \mathcal{G}_J) \succeq_{dom(\mathcal{G}_J)} (\mathcal{G}_I \triangleright \mathcal{T})$

Example 1: synthetic node 7 (**anticipate** reduced operations)

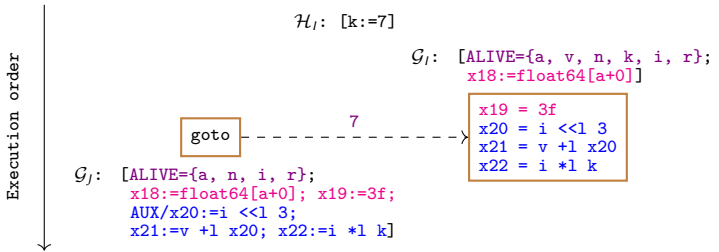


Validating Lazy Code Transformations on our example (2/3)

$\mathcal{G}_I$=input / $\mathcal{G}_J$=output, we have:

1. apply $\mathcal{H}$ (same for input/output here);
2. compare with $(\mathcal{S} \triangleright \mathcal{G}_J) \succeq_{dom(\mathcal{G}_J)} (\mathcal{G}_I \triangleright \mathcal{T})$

Example 1: synthetic node 7 (**anticipate** reduced operations)



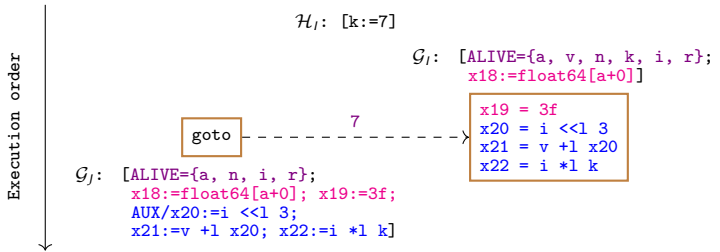
(1) $\mu_1 = \mu_2 = \texttt{Sinit}$ (2) $\vec{\sigma_2} = \vec{\sigma_1} = \texttt{float64}[a+0]$

Validating Lazy Code Transformations on our example (2/3)

$\mathcal{G}_I$=input / $\mathcal{G}_J$=output, we have:

1. apply $\mathcal{H}$ (same for input/output here);
2. compare with $(\mathcal{S} \triangleright \mathcal{G}_J) \succeq_{dom(\mathcal{G}_J)} (\mathcal{G}_I \triangleright \mathcal{T})$

Example 1: synthetic node 7 (**anticipate** reduced operations)



(1) $\mu_1 = \mu_2 = \texttt{Sinit}$ (2) $\vec{\sigma_2} = \vec{\sigma_1} = \texttt{float64}[a+0]$

(3) $\mathcal{R}_1 = \mathcal{R}_2 = a \coloneqq a \parallel n \coloneqq n \parallel i \coloneqq i \parallel r \coloneqq r \parallel$

$x_{18} \coloneqq \texttt{float64}[a+0] \parallel x_{19} \coloneqq 3\texttt{f} \parallel x_{21} \coloneqq 8 \cdot i + v \parallel x_{22} \coloneqq 7 \cdot i$

$\implies \delta_1 \succeq \delta_2$

Validating Lazy Code Transformations on our example (3/3)

$\mathcal{G}_I$=input / $\mathcal{G}_J$=output, we have:

1. apply $\mathcal{H}$ (same for input/output here);
2. compare with $(\mathcal{S} \triangleright \mathcal{G}_J) \succeq_{dom(\mathcal{G}_J)} (\mathcal{G}_I \triangleright \mathcal{T})$

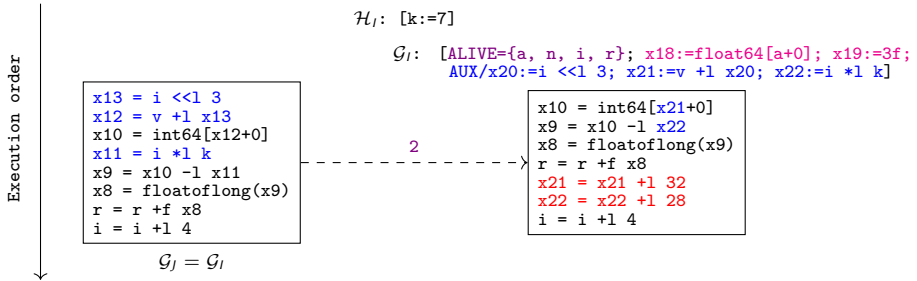
Example 2: loop block 2 (**remember** reduced operations)

Validating Lazy Code Transformations on our example (3/3)

$\mathcal{G}_I$=input / $\mathcal{G}_J$=output, we have:

1. apply $\mathcal{H}$ (same for input/output here);
2. compare with $(\mathcal{S} \triangleright \mathcal{G}_J) \succeq_{dom(\mathcal{G}_J)} (\mathcal{G}_I \triangleright \mathcal{T})$

Example 2: loop block 2 (**remember** reduced operations)

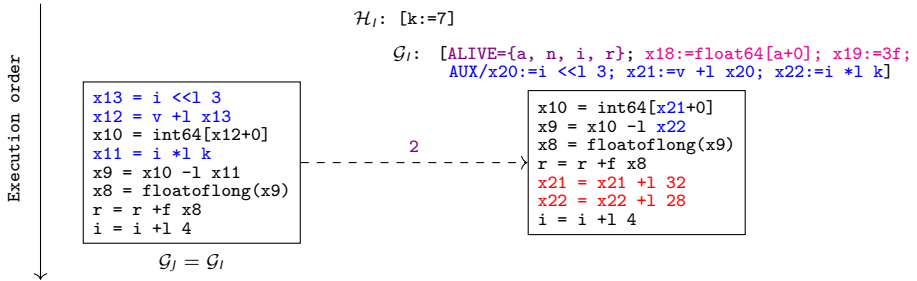


Validating Lazy Code Transformations on our example (3/3)

$\mathcal{G}_I$=input / $\mathcal{G}_J$=output, we have:

1. apply $\mathcal{H}$ (same for input/output here);
2. compare with $(\mathcal{S} \triangleright \mathcal{G}_J) \succeq_{dom(\mathcal{G}_J)} (\mathcal{G}_I \triangleright \mathcal{T})$

Example 2: loop block 2 (**remember** reduced operations)



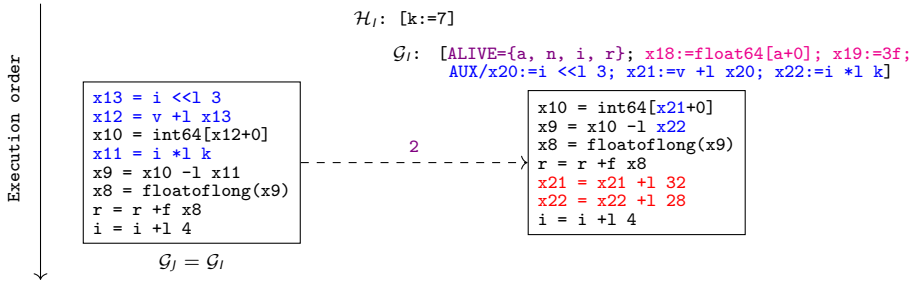
(1) $\mu_1 = \mu_2 = \texttt{Sinit}$ (2) $\vec{\sigma_2} = \vec{\sigma_1} = \texttt{float64}[a + 0];\ \texttt{int64}[8 \cdot i + v]$

Validating Lazy Code Transformations on our example (3/3)

$\mathcal{G}_I$=input / $\mathcal{G}_J$=output, we have:

1. apply $\mathcal{H}$ (same for input/output here);
2. compare with $(\mathcal{S} \triangleright \mathcal{G}_J) \succeq_{dom(\mathcal{G}_J)} (\mathcal{G}_I \triangleright \mathcal{T})$

Example 2: loop block 2 (**remember** reduced operations)



(1) $\mu_1 = \mu_2 = \texttt{Sinit}$ (2) $\vec{\sigma_2} = \vec{\sigma_1} = \texttt{float64}[a+0];\ \texttt{int64}[8 \cdot i + v]$

(3) $\mathcal{R}_1 = \mathcal{R}_2 = a \coloneqq a \parallel n \coloneqq n \parallel i = i + 4 \parallel x_{18} \coloneqq \texttt{float64}[a+0] \parallel x_{19} \coloneqq 3\texttt{f} \parallel$
$x_{21} \coloneqq 8 \cdot i + v + 32 \parallel x_{22} \coloneqq 7 \cdot i + 28 \parallel r \coloneqq r + \texttt{fofl}(\texttt{int64}[8 \cdot i + v] - 7 \cdot i)$

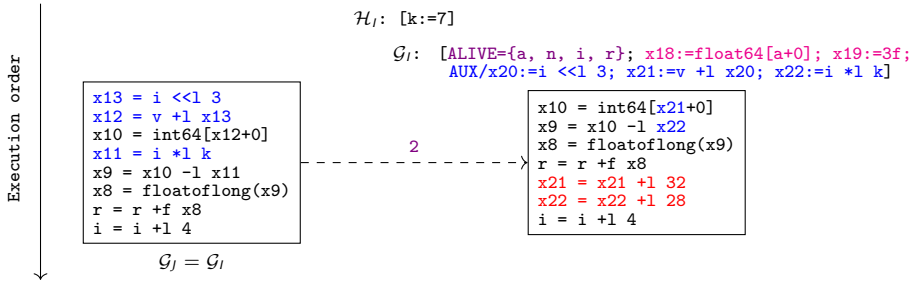
$\implies \delta_1 \succeq \delta_2$

Validating Lazy Code Transformations on our example (3/3)

$\mathcal{G}_I$=input / $\mathcal{G}_J$=output, we have:

1. apply $\mathcal{H}$ (same for input/output here);
2. compare with $(\mathcal{S} \triangleright \mathcal{G}_J) \succeq_{dom(\mathcal{G}_J)} (\mathcal{G}_I \triangleright \mathcal{T})$

Example 2: loop block 2 (**remember** reduced operations)

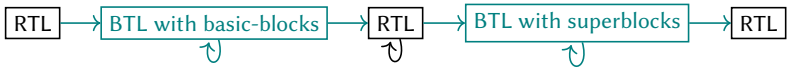


Some symbolic values were rewritten to a normal form, e.g.

$$x_{21} := 8 \cdot i + v + 32 \parallel x_{22} := 7 \cdot i + 28 \implies \delta_1 \succeq \delta_2$$

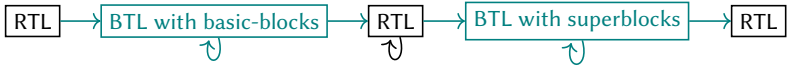
...using a restricted **affine theory**.

A step back: summary on Block Transfer Language & CFG morphisms



† = my contributions

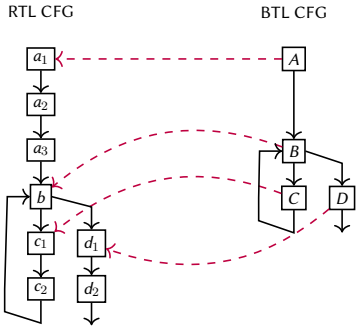
A step back: summary on Block Transfer Language & CFG morphisms



- Code expansions†
- Lazy Code Transformations†
- Store Motion

- Liveness analysis
- Simple Dead-Code Elimination†
- Renaming & If-lifting
- Prepass scheduling

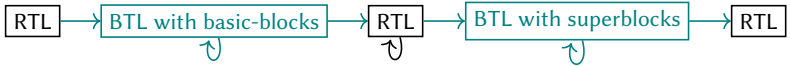
† = my contributions



Other contribution: a control flow graph morphism validator

Parametrized according to the type of morphism, used to validate:

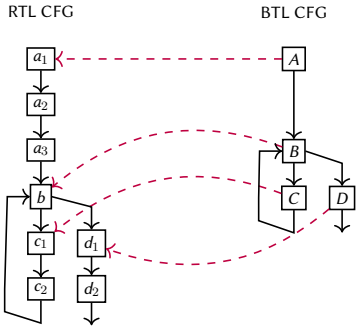
A step back: summary on Block Transfer Language & CFG morphisms



- Code expansions†
- Lazy Code Transformations†
- Store Motion

- Liveness analysis
- Simple Dead-Code Elimination†
- Renaming & If-lifting
- Prepass scheduling

† = my contributions

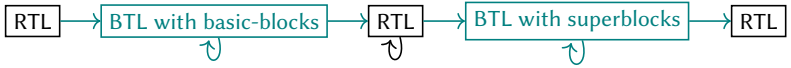


Other contribution: a control flow graph morphism validator

Parametrized according to the type of morphism, used to validate:

- the RTL↔BTL **translation**

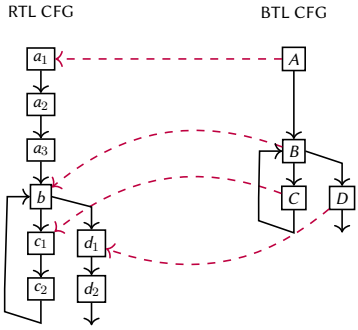
A step back: summary on Block Transfer Language & CFG morphisms



- Code expansions†
- Lazy Code Transformations†
- Store Motion

- Liveness analysis
- Simple Dead-Code Elimination†
- Renaming & If-lifting
- Prepass scheduling

† = my contributions



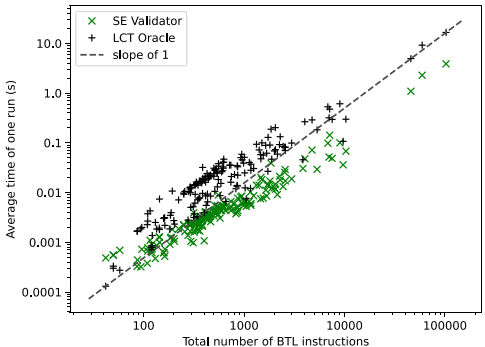
Other contribution: a control flow graph morphism validator

Parametrized according to the type of morphism, used to validate:

- the RTL↔BTL **translation**
- code **duplication** (loop unrollings) & **factorization** (DFA minimization)
- the **insertion of synthetic nodes** for data-flow analyses

Experimental evaluation

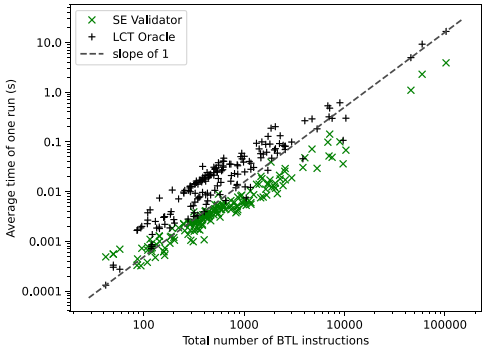
Compile times that scale
(thanks to formally verified hash-consing)



Benchmarks: LLVMtests, MiBench, PolyBench, TACLeBench, Verimag

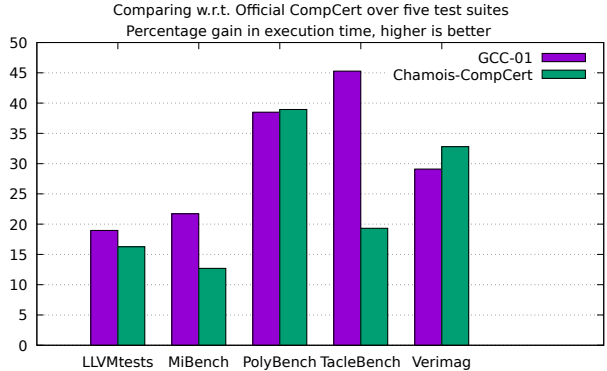
Experimental evaluation

Compile times that scale
(thanks to formally verified hash-consing)



Benchmarks: LLVMtests, MiBench, PolyBench, TACLeBench, Verimag

Closing the gap with"`GCC -O1`"



Measured on a RISC-V U74 Core
(SiFive HiFive Unmatched board)
Median gain w.r.t. Official CompCert
with relative standard deviation $\leq 2\%$

Conclusion

Insights

Formally verified defensive programming **helps** in validating advanced compiler optimizations:

Conclusion

Insights

Formally verified defensive programming **helps** in validating advanced compiler optimizations:

- A formally verified interpreter only does **simple** computations;
- Oracles generate hints that are **simple for them to yield,** but that would be **hard to have the validators reconstruct.**

→ Defensive, hash-consed symbolic execution is an efficient way of validating **a class** of intra-procedural transformations!

Conclusion

Insights

Formally verified defensive programming **helps** in validating advanced compiler optimizations:

- A formally verified interpreter only does **simple** computations;
- Oracles generate hints that are **simple for them to yield,** but that would be **hard to have the validators reconstruct.**

→ Defensive, hash-consed symbolic execution is an efficient way of validating **a class** of intra-procedural transformations!

Future work

Can we extend this principle for **security** (in contrast to safety) applications?

Conclusion

Insights

Formally verified defensive programming **helps** in validating advanced compiler optimizations:

- A formally verified interpreter only does **simple** computations;
- Oracles generate hints that are **simple for them to yield,** but that would be **hard to have the validators reconstruct.**

→ Defensive, hash-consed symbolic execution is an efficient way of validating **a class** of intra-procedural transformations!

Future work

Can we extend this principle for **security** (in contrast to safety) applications?

1. To **prove the insertion** of security countermeasures (correctness)
2. To provide some **security guarantees** w.r.t. an abstract attacker model

Thank You! Questions?

Online code:  -CompCert version at:
`https://gricad-gitlab.univ-grenoble-alpes.fr/certicompil/Chamois-CompCert`

Manuscript (frozen) CompCert version at:
`https://framagit.org/yukit/compcert-chamois-gl-thesis`

Main publications:

- Cyril Six, Léo Gourdin, Sylvain Boulmé, David Monniaux, Justus Fasse, and Nicolas Nardino. “Formally Verified Superblock Scheduling.”, CPP 2022.
- Léo Gourdin. “Lazy Code Transformations in a Formally Verified Compiler.”, ICOOOLPS 2023.
- David Monniaux, Léo Gourdin, Sylvain Boulmé, and Olivier Lebeltel. “Testing a Formally Verified Compiler.”, TAP 2023.
- Léo Gourdin, Benjamin Bonneau, Sylvain Boulmé, David Monniaux, and Alexandre Bérard. “Formally Verifying Optimizations with Block Simulations.”, OOPSLA 2023.

Appendices

- Peephole & Postpass on AArch64
- If-lifting
- Loop Unrollings
- CompCert's Trusted Computing Base
- Safe translation validation in Coq
- Hash-consing
- Why on RISC-V?
- BTL syntax & semantics
- RISC-V macros expansions & mini-CSE
- Predicates for Lazy Code Transformations
- Diagrammatic proof of blockstep simulation
- Development size
- More benchmark results

Peephole pairing load (and store) instructions on AArch64

[Gourdin 2021; Six et al. 2022]

```
w1 := ldr [x6, #0]
w2 := add w4, w3
w4 := ldr [x6, #4]   // WAR w4
str w2, [x1, #4]
w5 := ldr [x3, #4]
w6 := add w5, w3  // RAW w5
w7 := ldr [x3, #0]
```

Source

```
w2 := add w4, w3
w1, w4 := ldp [x6, #0] // WAR w4
str w2, [x1, #4]
w7, w5 := ldp [x3, #0]
w6 := add w5, w3    // RAW w5
```

Target

Rewriting rule before symbolic simulation:

under guard $r_1 \neq r_2$

$$r_1, r_2 := \mathbf{ldp}[r_3, \#n] \quad \rightarrow \quad \begin{array}{l} r_2 := r_3; \\ r_1 := \mathbf{ldr}[r_3, \#n]; \\ r_2 := \mathbf{ldr}[r_2, \#n + 4] \end{array}$$

Proving the correctness of this rewriting rule is **much easier** than a direct proof on the peephole optimization.

Example: the finer capabilities of postpass (on AArch64)

Reordering an instruction expanded at the Asm level

```
1 int main(int x, int y) {
2   int z = x << 32;
3   y = y - z;
4   return x + y;
5 }
```

```
I1 orr w2, wzr, #32
I2 lsl w2, w0, w2
I3 sub w3, w1, w2
I4 add w0, w0, w3
I5 ldr x30, [sp, #8]
I6 add sp, sp, #16
I7 ret x30
```

Before postpass

```
I1 orr w2, wzr, #32
I5 ldr x30, [sp, #8]
I2 lsl w2, w0, w2
I6 add sp, sp, #16
I3 sub w3, w1, w2
I4 add w0, w0, w3
I7 ret x30
```

After postpass

Main difference: the load of the return address is lifted.

Latencies

LSL=2; LDR=3; others=1

Stalls info

1. w2 is not ready!
2. sp is not ready!

bad scheduling

EXEC1	EXEC2
I_1	
I_2	
$stall_1$	I_2
I_3	
I_4	I_5
$stall_2$	I_5
$stall_2$	I_5
I_6	

running time ↓

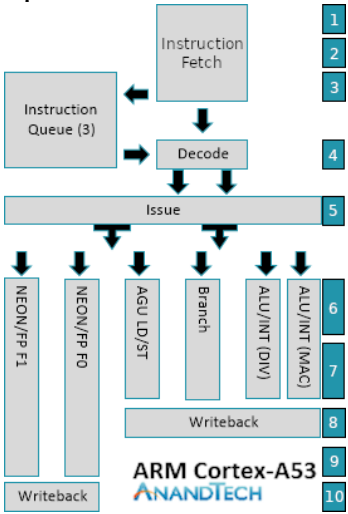
good scheduling

EXEC1	EXEC2
I_1	I_5
I_2	I_5
I_2	I_5
I_6	I_3
I_4	

8 versus 5 cycles,
3 cycles are won!

Instruction Level Parallelism

Pipeline of the ARMv8 Cortex A53



Two dimensions of parallelism

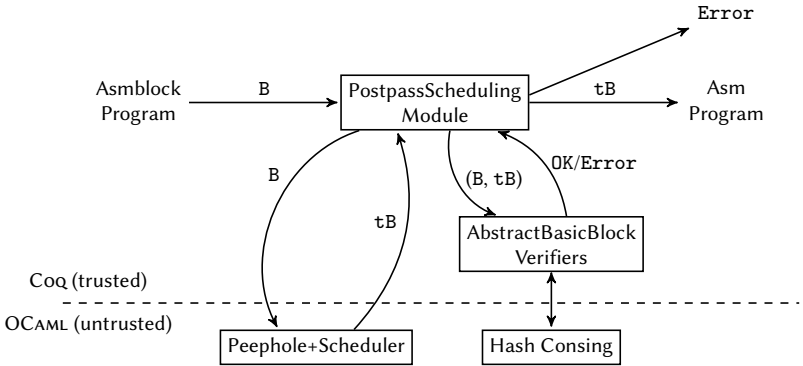
- vertical: several stages of computing units
- horizontal: several units at the same stage

Usually **interlocked** pipeline: observationnally, assembly semantics is sequential!
(with dynamically inserted **stalls**)

On VLIW processors:
horizontal parallelism specified by the assembly program
(i.e. “tiny-scope” parallelism).

Certifying Peephole & Postpass by translation validation

How it works?



- Adapted from [Six et al. 2020]
- Generic verifier backend, specialized Domain Specification Language
- The verifier proof is independent of the transformations

Asmblock implementation & basic blocks structure

Basic block: A block with at most one branching instruction, in final position. The sequence is only reachable at its first instruction.

```
Inductive basic: Type :=   (* basic instructions *)
Inductive control: Type := (* control-flow instructions *)
Record bblock := {
  header: list label;  body: list basic;  exit: option control;
  correct: Is_true(non_empty_body body || non_empty_exit exit)
}
```

State (rs,m): A tuple of a register state *rs* (mapping registers to values) and a memory state *m* (mapping addresses to values).

The basic block is executed from $\binom{rs_0}{m_0}$ to $\binom{rs_n}{m_n}$:

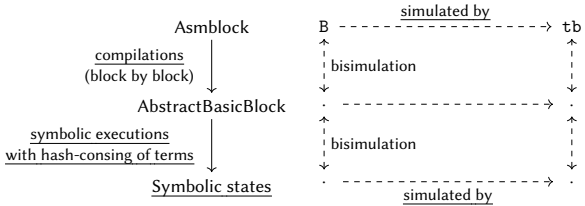
$$\binom{rs_0}{m_0} \xrightarrow{\texttt{bstep}} \binom{rs_1}{m_1} \xrightarrow{\texttt{bstep}} \cdots \xrightarrow{\texttt{bstep}} \binom{rs_{n-1}}{m_{n-1}} \xrightarrow{\texttt{estep}} \binom{rs_n}{m_n}$$

$$\texttt{bbstep}\ f\ [b_1; b_2; \cdots; b_n]\ rs_0\ m_0$$

A Domain Specific Language for symbolic execution of assembly code

Simulation test correctness

1. Code is translated in the generic AbstractBasicBlock DSL
2. A symbolic execution is run to compute “symbolic states”
3. Simulation is deduced from syntactical equalities on “symbolic states”



Assembly level framework: proof effort and benefits

Overall implementation: three man·months of development.

- Machblock to Asmblock: A difficult star simulation
- Peephole/postpass proof in Asmblock: a simple lockstep simulation
- Asmblock to Asm: a plus simulation

Simulation property of the verifier :

```
Definition bblock_simu (lk: aarch64_linker)
   (ge: Genv.t fundef unit) (f: function) (bb bb': bblock) :=
   ∀ rs m rs' m' t,
  exec_bblock lk ge f bb rs m t rs' m' →
   exec_bblock lk ge f bb' rs m t rs' m'
```

Bug found while implementing the verifier

- Difference between the formal specification of Asm and the “printer”
- Concerns `Pfmovimmd` and `Pfmovimms` macro-instructions
- Instruction behavior was not fully specified

Go back to slide 31.

Interleaving of rotated & unrolled loop-bodies on Cortex A-53 (AArch64)

```
double sumsq(double *x, int len){
  double s = 0.0; for (int i=0; i < len; i++) s += x[i]*x[i];
  return s;
}
```

```
1  .L101: // DO-WHILE loop
2    ldr  d2,[x0,w2,sxtw #3]
3    fmul  d1, d2, d2
4    fadd  d0, d0, d1 // d0 += x[w2]^2
5    add   w2, w2, #1
6    cmp   w2, w1
7    b.ge  .L100 // end body 1
8    ldr   d2,[x0,w2,sxtw #3]
9    fmul d1, d2, d2
10   fadd d0, d0, d1
11   add   w2, w2, #1
12   cmp   w2, w1
13   b.lt .L101 // end body 2
14 .L100: // loop exit
15   // only d0 is live here
```

```
.L101:
  ldr   d2,[x0,w2,sxtw #3]
  add   w2, w2, #1
  cmp   w2, w1
  b.ge  .L102
  ldr   d3,[x0,w2,sxtw #3]
  add   w2, w2, #1
  fmul  d1, d2, d2
  cmp   w2, w1
  fmul d4, d3, d3
  fadd  d0, d0, d1
  fadd d0, d0, d4
  b.lt .L101
  b .L100
.L102:
  fmul d1, d2, d2
  fadd d0, d0, d1
.L100:
```

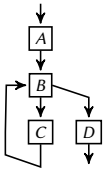
Gain of right hand-side schedule $\simeq$ 30% wrt the (above) source order.

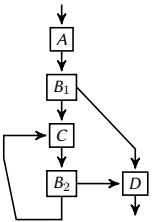
Go back to slide 31.

Validating loop-unrollings through CFG-projections

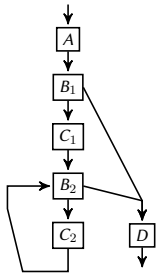
Various loop-unrollings (below)
from the source "while-do" loop on the right

A = before the loop B = loop-condition
C = loop-body D = after the loop

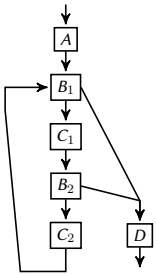




rotate
(i.e. if-do-while)



unroll 1st iteration



unroll body

Go back to slide 31.

The main parts of CompCert Trusted Computing Base (TCB)

- formal semantics of the CompCert C language (in Coq);
- formal semantics of the assembly languages (in Coq);
- option parsing and filename handling (in OCaml);
- preprocessor (partly external, partly in OCaml), which turns regular C into CompCert C;
- "assembly expansions" (in OCaml) dealing with "pseudo-instructions" for stack (de)allocation & memory copy;
- formal axiomatization (in Coq) of these pseudo-instructions;
- assembly pretty-printer (in OCaml);
- compatibility of the ABI used by CompCert with other libraries (e.g. standard C library) compiled on the system with GCC;
- external assembler and linker;
- Coq TCB (+ "purity of oracles is not used in the Coq proof")

Go back to slide 31.

Translation validation in Coq

Declaring a foreign function in Coq using an axiom is not totally safe:
⇒ OCaml "function" are not functions in a mathematical pov, but "relations", as they are nondeterministics.

Existing oracles in CompCert are declared as "pure" functions:
Example of register allocation:

```
Axiom regalloc: RTL.func → option LTL.func
```

implemented by imperative OCaml code using hash-tables.
⇒ not a real issue, as their purity is not used in the formal proof;

Successfully applied in the VPL (Verified Polyhedra Library)
[Boulmé, Fouilhé, Maréchal, Monniaux, Périn, etc'2013-2018]
And partially applied in our version of CompCert
[Boulmé, Gourdin, Fasse, Monniaux, Six'2018-2023]

The Impure library

1. We rely on the Impure library [Boulmé 2021] to model OCaml foreign functions as nondeterministic ones;
2. Based on *may-return monads* of [Fouilhé and Boulmé 2014] to make determinism unprovable

Impure computation $\triangleq$ Coq code embedding OCaml code

- **Axiomatize** (in Coq) "$A \to \mathrm{Prop}$" as type "$??A$"
  to represent "impure computations of type A"
  with "$(k\ a)$" as proposition "$k \rightsquigarrow a$"
  with formal type $\rightsquigarrow_A : ??A \to A \to \mathrm{Prop}$
  read "computation k may return value a"
  and usual monad operators
- "$??A$" extracted like "A".

Features of this approach

Summary of our approach:

- Almost any OCaml function embeddable into Coq.
  (e.g. mutable data-structures with aliasing in Coq)
- No formal reasoning on *effects*, only on results:
  foreign functions could have bugs, only their type is ensured.
  $\Rightarrow$ Considered as nondeterministic.
  e.g. for I/O reasoning, use FreeSpec or InteractionTrees instead.
- OCaml polymorphism provides "*theorems-for-free*"
  (i.e. a form of unary parametricity through Coq extraction)
- Exceptionally: additional axioms on results (e.g. pointer equality)
  In this case, the foreign function must be trusted!

Go back to slide 31.

Verified defensive hash-consing factory from pointer equality

Hash-consing of inductive type `T` consists in memoizing its constructors through a dedicated factory.

[Six et al. 2020] gives a verified defensive variant of [Filliâtre and Conchon 2006]:

- a polymorphic oracle provides—for any `T`—an untrusted hash-consing factory of type $\texttt{T} \to \texttt{??T}$;
- this factory is wrapped into a certified factory dynamically enforcing that each returned term is structurally equals to its inputs...
- ...through a **constant-time** checking that,
  on input $(c\ t_1 \ldots t_n)$ and output $(c'\ t'_1 \ldots t'_m)$,
  we have $c = c'$ and that forall i, $t_i == t'_i$

works in practice because of (the non-formalized) invariant:
all t_i are already "hash-consed" terms

Go back to slide 31.

Why targetting RISC-V for Strength Reduction?

CompCert is particularly slow on RISC-V.

1. Less work went on this backend;
2. Instruction Set Architecture (ISA) is simpler;
3. Addressing modes are very limited;
   e.g. consider a load in C "`x = a[i]`*", CompCert produces:*

On AArch64:

```
ldr  x0, [x0,w1,sxtw#3]
```

On RISC-V:

```
slli  x6, x11, 3
add   x6, x10, x6
ld    x6, 0(x6)
```

4. RISC-V is a good candidate for the future of embedded (and critical) systems.
   e.g. NOEL-V for space; openness of hardware; modularity

Porting the LCT's strength reduction to other backends should be straightforward (∼140 LoC).

Go back to slide 31.

The BTL IR: A syntax-based block representation

$$
\begin{aligned}
fi ::= {} & \texttt{Bgoto}(l) \\
\mid {} & \texttt{Breturn}([r]) \\
\mid {} & \texttt{Bcall}(sig, (r|id), \vec{r}, r, l) \\
\mid {} & \texttt{Btailcall}(sig, (r|id), \vec{r}) \\
\mid {} & \texttt{Bbuiltin}(ef, \vec{br}, br, l) \\
\mid {} & \texttt{Bjumptable}(r, \vec{l})
\end{aligned}
$$

$$
\begin{aligned}
blk ::= {} & \texttt{BF}(fi, iinfo) \\
\mid {} & \texttt{Bnop}([iinfo]) \\
\mid {} & \texttt{Bop}(op, \vec{r}, r, iinfo) \\
\mid {} & \texttt{Bload}(trap, chk, addr, \vec{r}, r, iinfo) \\
\mid {} & \texttt{Bstore}(chk, addr, \vec{r}, r, iinfo) \\
\mid {} & \texttt{Bseq}(blk_1, blk_2) \\
\mid {} & \texttt{Bcond}(cond, \vec{r}, blk_{so}, blk_{not}, iinfo)
\end{aligned}
$$

Keeping a block structure is interesting for at least two reasons:

1. Invariants are checked for blocks instead of every instruction;
2. Block-scoped optimizations (e.g. scheduling) are still compatible.

Two shades of BTL Invariants

$\Rightarrow$ To avoid redundancies in invariants and facilitate their generation by oracles.

An abstract (theorical) representation

Assignments of invariant values (into reg).

```
(** FPASV: "Finite Parallel Assignment of Symbolic Values" *)
Record fpasv :=
  { fpa_ok: list sval; fpa_reg:> PTree.tree sval;
    fpa_wf: ∀ r sv, fpa_reg!r = Some sv → ~(is_input sv) → List.In sv fpa_ok }
```

A more compact representation

In the set of output registers, we distinguish those not defined in `aseq` (which satisfy `[r:=Sinput r]`).

```
(** CSASV: "Compact Sequence Assignments of Symbolic Values" *)
Record csasv := {
  aseq: list (reg * ival);
  outputs: Regset.t;
}
```

Go back to slide 31.

Rewritings & mini-CSE over superblocks on RISC-V (1/3)

```
1  long foo(int x, char y, long *t) {
2    int z = x / 4096;
3    y = x / 256;
4    t[0] = t[1] * t[2];
5    if (x + z < 7) {
6      if (y < 7)
7        return 421 + t[0];
8    }
9    y = y - z;
10   return x + y - t[0];
11 }
```

Colors delimit superblocks.

- Sub-optimal ordering
- Macros (in **pink**) are not expansed

```
Bop: x4 = x3 >> 12 # 1
Bop: x15 = x3 >> 8 # 2
Bop: x2 = x15 \& 255
Bload: x13 = int64[x1 + 8]
Bload: x14 = int64[x1 + 16]
Bop: x12 = x13 *l x14
Bstore: int64[x1 + 0] = x12
Bop: x11 = x3 + x4
Bcond: (x11 >=s 7) # 3
  ifso = [ Bgoto: 7 ]
Bcond: (x2 <s 7) # 4
  ifso = [ Bgoto: 10 ]
Bgoto: 7
```

Non-optimized RISC-V CompCert code (uncolored is **orange**)

Rewritings & mini-CSE over superblocks on RISC-V (2/3)

```
1  long foo(int x, char y, long *t) {
2    int z = x / 4096;
3    y = x / 256;
4    t[0] = t[1] * t[2];
5    if (x + z < 7) {
6      if (y < 7)
7        return 421 + t[0];
8    }
9    y = y - z;
10   return x + y - t[0];
11 }
```

- No duplications thks to mini-CSE on the expansion of #3 and #4
- Bad ordering
- Makespan is 14 on U74

```
Bop: x16 = x3 >> 31 # 1
Bop: x17 = x16 >> 20 # 1
Bop: x18 = x3 + x17 # 1
Bop: x4 = x18 >> 12 # 1
Bop: x20 = x16 >> 24 # 2
Bop: x21 = x3 + x20 # 2
Bop: x15 = x21 >> 8 # 2
Bop: x2 = x15 \& 255
Bload: x13 = int64[x1 + 8]
Bload: x14 = int64[x1 + 16]
Bop: x12 = x13 *l x14
Bstore: int64[x1 + 0] = x12
Bop: x11 = x3 + x4
Bop: x22 = OEaddiw(X0,7) # 3,4
Bcond: (CEbgew(x11 >= x22)) # 3
  ifso = [ Bgoto: 7 ]
Bcond: (CEbltw(x2 < x22)) # 4
  ifso = [ Bgoto: 10 ]
Bgoto: 7
```

Pre-processed RISC-V
CompCert code (uncolored is **orange**)

Rewritings & mini-CSE over superblocks on RISC-V (3/3)

```
1  long foo(int x, char y, long *t) {
2    int z = x / 4096;
3    y = x / 256;
4    t[0] = t[1] * t[2];
5    if (x + z < 7) {
6      if (y < 7)
7        return 421 + t[0];
8    }
9    y = y - z;
10   return x + y - t[0];
11 }
```

We won 5 cycles!

- Better ordering
- Makespan is reduced to 9 thanks to avoided stalls

```
Bop: x16 = x3 >>s 31
Bload: x13 = int64[x1 + 8]
Bop: x17 = x16 >>u 20
Bload: x14 = int64[x1 + 16]
Bop: x18 = x3 + x17
Bop: x20 = x16 >> 24
Bop: x4 = x18 >>s 12
Bop: x21 = x3 + x20
Bop: x15 = x21 >>s 8
Bop: x12 = x13 *l x14
Bop: x2 = x15 \& 255
Bop: x11 = x3 + x4
Bop: x22 = OEaddiw(X0,7)
Bstore: int64[x1 + 0] = x12
Bcond: (CEbgew(x11 >= x22))
  ifso = [ Bgoto: 7 ]
Bcond: (CEbltw(x2 < x22))
  ifso = [ Bgoto: 10 ]
Bgoto: 7
```

Optimized RISC-V CompCert code (uncolored is orange)

Go back to slide 31.

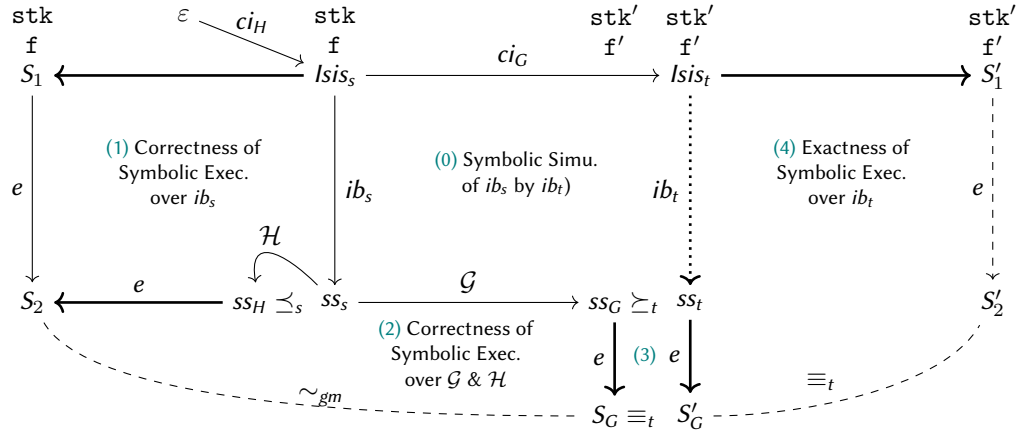
Bit vector predicates for LCT (non-exhaustive list)

Candidates (of the form $n \equiv v := t$ at node n, writing term t in variable v) are operations or loads.
Boolean equation systems to solve for each node, and for each candidate:

- **Transparency:** the node does not alter the candidate expr.;
- **Comp:** the node contains a computation of the candidate;
- **Down-safety:** a computation t at n does not introduce a new value on a terminating path starting at n;
- **Up-safety:** same for every path leading at n;
- **Earliestness:** can't be placed earlier without breaking the safety property;
- **Delayability:** possibility to move the inserted value from its earliest down-safe point as far as possible in the direction of the control-flow;
- **Latestness:** optimality of delayability (maximum delay);
- **Isolatedness:** the inserted computation would be isolated in its block;
- **Insert:** Candidate should be inserted at this node;
- **Replace:** Candidate should be replaced at this node.

Go back to slide 31.

Diagrammatic proof of blockstep simulation



(3): Correctness of the modulo liveness relation

Go back to slide 31.

An idea of the development size

In number of significant lines of code (sloc)...

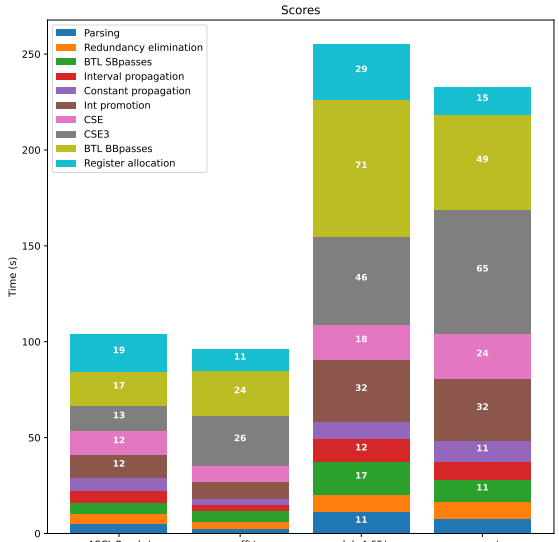
Project	Defs	Proofs
BTL IR	252	20
BTL projection checker	296	121
RTL $\rightarrow$ BTL	313	377
BTL $\rightarrow$ RTL	146	249
BTL SE theory	1844	1862
BTL SE refinement	1612	1411
BTL rewriting engine (RISC-V only)	1209	1038
BTL passes module	122	60
Total	5794	5138

Project	Ocaml	Coq
BTL oracles & framework	3332	10 932
AArch64 scheduling & peephole	1157	11 171
Total	4489	22 103

LCT oracle combining code motion & strength reduction: 2000 sloc

Go back to slide 31.

Compilation time of slowest CompCert passes



Results zooming on the LCT impact

GCC, Base=(scheduling + CSE3 + unroll single), and Base+LCT versus mainline CompCert on RISC-V U74, higher is better

Setup	GCC -O1	Base	**Base + LCT**
LLVMtest/fpconvert	+24.22%	+7.9%	+17.15%
LLVMtest/matmul	+15.9%	+115.05%	+144.11%
LLVMtest/nbench_bf	+74.58%	+11.84%	+24.51%
MiBench/jpeg	+27.75%	+20.62%	+24.75%
MiBench/sha	+92.43%	+45.68%	+51.73%
MiBench/stringsearch	+133.34%	+40.28%	-10.15%
PolyBench/*	+64.05%	+38.06%	+46.23%
TACLeBench/bsort	+49.04%	+9%	+33.16%
TACLeBench/deg2rad	+56.75%	+41.5%	+50.28%
TACLeBench/md5	+42.18%	+18.59%	+47.93%

Go back to slide 31.