Certifying assembly optimizations in Coq by symbolic execution with hash-consing

Léo Gourdin and Sylvain Boulmé [1]

{Leo.Gourdin,Sylvain.Boulme}@univ-grenoble-alpes.fr
Joint work with David Monniaux, Cyril Six and Justus Fasse.

02/07/2021

Université Grenoble Alpes

Verimag

[1] This work has been partially supported by the LabEx PERSYVAL-Lab (ANR-11-LABX-0025-01) funded by the French program Investissement d'avenir.

Contents

1 Lightweight reasoning on pointer equalities and hash-consing in Coq

2 Realistic applications to formally-verified compilation

3 Experimental results and Conclusion

Checking compiler optimizations with pointer equalities

Question: Given two sequences of assignments (B_1) and (B_2) such as
$(B_1)\ \ r_1 := r_1 + r_2;\ r_3 := r_1;\ r_1 := r_1 + r_3$
$(B_2)\ \ r_3 := r_1 + r_2;\ r_1 := r_3 + r_3$
How to implement a **linear** check of their *observational* equivalence?

Checking compiler optimizations with pointer equalities

Question: Given two sequences of assignments (B_1) and (B_2) such as
$(B_1)\ \ r_1 := r_1 + r_2;\ r_3 := r_1;\ r_1 := r_1 + r_3$
$(B_2)\ \ r_3 := r_1 + r_2;\ r_1 := r_3 + r_3$
How to implement a **linear** check of their *observational* equivalence?

Old idea of [King, 1976]: check that their *symbolic execution* leads to the same parallel assignment: $r_1 := (r_1 + r_2) + (r_1 + r_2) \parallel r_3 := r_1 + r_2$.

Checking compiler optimizations with pointer equalities

Question: Given two sequences of assignments (B_1) and (B_2) such as
(B_1) $r_1 := r_1 + r_2;\ r_3 := r_1;\ r_1 := r_1 + r_3$
(B_2) $r_3 := r_1 + r_2;\ r_1 := r_3 + r_3$
How to implement a **linear** check of their *observational* equivalence?

Old idea of [King, 1976]: check that their *symbolic execution* leads to the same parallel assignment: $r_1 := (r_1 + r_2) + (r_1 + r_2) \parallel r_3 := r_1 + r_2$.

Issue: symbolic execution is linear, but *structural equality* of generated terms is exponential due to duplications (such as "$r_1 + r_2$")

Checking compiler optimizations with pointer equalities

Question: Given two sequences of assignments (B_1) and (B_2) such as
(B_1) $r_1 := r_1 + r_2;\ r_3 := r_1;\ r_1 := r_1 + r_3$
(B_2) $r_3 := r_1 + r_2;\ r_1 := r_3 + r_3$
How to implement a **linear** check of their *observational* equivalence?

Old idea of [King, 1976]: check that their *symbolic execution* leads to the same parallel assignment: $r_1 := (r_1 + r_2) + (r_1 + r_2) \parallel r_3 := r_1 + r_2$.

Issue: symbolic execution is linear, but *structural equality* of generated terms is exponential due to duplications (such as "$r_1 + r_2$")

Solution: (constant-time) *pointer equalities* instead of *structural* & *hash-consing* that binds the 2 symbolic executions to the same pointers.

Is there a Coq model of OCaml pointer equality (==) ?

```
Parameter phys_eq: nat → nat → bool (* only [nat] for simplicity.*)
Extract Constant phys_eq ⇒ "(==)"
```

where (in OCaml) "`let n = (S O) in n == n`" returns `true`
and "`(S O) == (S O)`" returns `false`.

Is there a Coq model of OCaml pointer equality (==) ?

```
Parameter phys_eq: nat → nat → bool (* only [nat] for simplicity.*)
Extract Constant phys_eq ⇒ "(==)"
```

where (in OCaml) "`let n = (S O) in n == n`" returns `true`
and "`(S O) == (S O)`" returns `false`.

Idea `phys_eq` returns `true` when applied to the "same physical object" `x`.

```
Hypothesis phys_eq_axiom: ∀ x, phys_eq x x = true
```

Wrong idea because each "`x`" above may correspond to a distinct pointer!

```
Lemma wrong_property: ∀ x y, x = y → phys_eq x y <> false
```

Is there a Coq model of OCaml pointer equality (==) ?

```
Parameter phys_eq: nat → nat → bool (* only [nat] for simplicity.*)
Extract Constant phys_eq ⇒ "(==)"
```

where (in OCaml) "`let n = (S O) in n == n`" returns `true`
and "`(S O) == (S O)`" returns `false`.

Idea `phys_eq` returns `true` when applied to the "same physical object" `x`.

```
Hypothesis phys_eq_axiom: ∀ x, phys_eq x x = true
```

Wrong idea because each "`x`" above may correspond to a distinct pointer!

```
Lemma wrong_property: ∀ x y, x = y → phys_eq x y <> false
```

Weaker alternatives of "`phys_eq_axiom`" cannot solve the issue:

```
Lemma congr: ∀ x y, x = y → phys_eq x x = phys_eq x y
Proof. congruence. Qed
```

Safe (and lightweight) abstraction OCaml functions in Coq

Issue of embedding OCaml untrusted imperative oracles in Coq:

```
Parameter oracle: nat → bool
Lemma congr: ∀ n, oracle n = oracle n
```

Unsound for `let oracle = (let b = ref false in fun _ -> b:=not !b; !b)`

Safe (and lightweight) abstraction OCaml functions in Coq

Issue of embedding OCaml untrusted imperative oracles in Coq:

```
Parameter oracle: nat → bool
Lemma congr: ∀ n, oracle n = oracle n
```

Unsound for `let oracle = (let b = ref false in fun _ -> b:=not !b; !b)`

Lightweight solution: May-return monads of [Fouilhé and Boulmé, 2014].

- represent each OCaml function of $A \to B$ as a "relation" $A \to ??B$ where $??A$ abstracts ($A \to$ `Prop`) through $\leadsto_A : ??A \to (A \to \mathrm{Prop})$ such that "$k \leadsto a$" means "$(k\ a)$".

Safe (and lightweight) abstraction OCaml functions in Coq

Issue of embedding OCaml untrusted imperative oracles in Coq:

```
Parameter oracle: nat → bool
Lemma congr: ∀ n, oracle n = oracle n
```

Unsound for `let oracle = (let b = ref false in fun _ -> b:=not !b; !b)`

Lightweight solution: May-return monads of [Fouilhé and Boulmé, 2014].

- represent each OCaml function of $A \to B$ as a "relation" $A \to ??B$ where $??A$ abstracts ($A \to$ `Prop`) through $\leadsto_A : ??A \to (A \to \mathrm{Prop})$ such that "$k \leadsto a$" means "$(k\ a)$".
- ??. provides usual monad operators and their axioms w.r.t. $\leadsto$.

Safe (and lightweight) abstraction OCaml functions in Coq

Issue of embedding OCaml untrusted imperative oracles in Coq:

```
Parameter oracle: nat → bool
Lemma congr: ∀ n, oracle n = oracle n
```

Unsound for `let oracle = (let b = ref false in fun _ -> b:=not !b; !b)`

Lightweight solution: May-return monads of [Fouilhé and Boulmé, 2014].

- represent each OCaml function of $A \to B$ as a "relation" $A \to ??B$ where $??A$ abstracts $(A \to \texttt{Prop})$ through $\leadsto_A : ??A \to (A \to \mathrm{Prop})$ such that "$k \leadsto a$" means "$(k\ a)$".
- $??.$ provides usual monad operators and their axioms w.r.t. $\leadsto$.
- $??A$ is extracted like A.

Safe (and lightweight) abstraction OCaml functions in Coq

Issue of embedding OCaml untrusted imperative oracles in Coq:

```
Parameter oracle: nat → bool
Lemma congr: ∀ n, oracle n = oracle n
```

Unsound for `let oracle = (let b = ref false in fun _ -> b:=not !b; !b)`

Lightweight solution: May-return monads of [Fouilhé and Boulmé, 2014].

- represent each OCaml function of $A \to B$ as a "relation" $A \to ??B$ where $??A$ abstracts $(A \to \texttt{Prop})$ through $\leadsto_A : ??A \to (A \to \mathrm{Prop})$ such that "$k \leadsto a$" means "$(k\ a)$".
- $??.$ provides usual monad operators and their axioms w.r.t. $\leadsto$.
- $??A$ is extracted like A.

NB: The "$??.$" monad only restricts Coq congruence w.r.t. to usual functions. "$A \to ??B$" can be viewed as a type of "non-deterministic" function which may not terminate normally: reasoning on $\leadsto$ in partial correctness.

Model of OCaml "==" in Coq using may-return monads

Proposed in [Six et al., 2020].

```
Axiom phys_eq: ∀ {A}, A → A → ?? bool
Extract Constant phys_eq ⇒ "(==)"
Axiom phys_eq_true: ∀ A (x y: A), phys_eq x y ⇝ true →  x=y
```

Model of OCaml "==" in Coq using may-return monads

Proposed in [Six et al., 2020].

```
Axiom phys_eq: ∀ {A}, A → A → ?? bool
Extract Constant phys_eq ⇒ "(==)"
Axiom phys_eq_true: ∀ A (x y: A), phys_eq x y ⇝ true →  x=y
```

NB: "`phys_eq x y ⇝ b`" means something like
"if `b=true` then it has existed an allocated object `o` such that `x=o=y`"
since the lemma below is still provable in Coq

```
Lemma ok: ∀ x y, x = y →  phys_eq x x ⇝ true →  phys_eq x y ⇝ true
```

Model of OCaml "==" in Coq using may-return monads

Proposed in [Six et al., 2020].

```
Axiom phys_eq: ∀ {A}, A → A → ?? bool
Extract Constant phys_eq ⇒ "(==)"
Axiom phys_eq_true: ∀ A (x y: A), phys_eq x y ⇝ true →  x=y
```

NB: "`phys_eq x y ⇝ b`" means something like
"if `b=true` then it has existed an allocated object `o` such that `x=o=y`"
since the lemma below is still provable in Coq

```
Lemma ok: ∀ x y, x = y →  phys_eq x x ⇝ true →  phys_eq x y ⇝ true
```

However, this wrong property cannot be proved (except on empty types):

```
∀ x y, x=y → phys_eq x x ⇝ true → phys_eq x y ⇝ false → False
```

because, if "`phys_eq x y ⇝ b`" interpreted as "`b=true → x=y`", then
the property reduces to "`∀ x y, x=y → False`"

Lightweight certified hash-consing from pointer equality

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

Lightweight certified hash-consing from pointer equality

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

[Six et al., 2020] proposes a variant of [Filliâtre and Conchon, 2006] with *a posteriori* certification in Coq:

Lightweight certified hash-consing from pointer equality

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

[Six et al., 2020] proposes a variant of [Filliâtre and Conchon, 2006] with *a posteriori* certification in Coq:

- for any `T`, an oracle provides an *untrusted* hash-consing factory of type `T` $\rightarrow$ `??T`;

Lightweight certified hash-consing from pointer equality

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

[Six et al., 2020] proposes a variant of [Filliâtre and Conchon, 2006] with *a posteriori* certification in Coq:

- for any `T`, an oracle provides an *untrusted* hash-consing factory of type $\texttt{T} \rightarrow \texttt{??T}$;
- this factory is wrapped into a *certified* factory *dynamically enforcing* that each returned term is *structurally equals* to its inputs...

Lightweight certified hash-consing from pointer equality

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

[Six et al., 2020] proposes a variant of [Filliâtre and Conchon, 2006] with *a posteriori* certification in Coq:

- for any `T`, an oracle provides an *untrusted* hash-consing factory of type $\mathtt{T} \to\ \mathtt{??T}$;
- 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, $t_i == t'_i$

Lightweight certified hash-consing from pointer equality

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

[Six et al., 2020] proposes a variant of [Filliâtre and Conchon, 2006] with *a posteriori* certification in Coq:

- for any `T`, an oracle provides an *untrusted* hash-consing factory of type $\mathtt{T} \rightarrow \mathtt{??T}$;
- 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

See `https://github.com/boulme/Impure/`

Contents

1 Lightweight reasoning on pointer equalities and hash-consing in Coq

2 Realistic applications to formally-verified compilation

3 Experimental results and Conclusion

Translation validation of untrusted optimizations

Translation validation by symbolic execution

Transforming an input program P_1 into an output program P_2, and using *symbolic execution* to assert *semantic preservation* between P_1 and P_2.

Translation validation of untrusted optimizations

Background and history of the method

Translation validation by symbolic execution

Transforming an input program P_1 into an output program P_2, and using *symbolic execution* to assert *semantic preservation* between P_1 and P_2.

1. [Necula, 2000] has established its effectiveness over 20 years ago. *However, the mechanism itself was not proved correct.*

Translation validation of untrusted optimizations

Background and history of the method

Translation validation by symbolic execution

Transforming an input program P_1 into an output program P_2, and using *symbolic execution* to assert *semantic preservation* between P_1 and P_2.

1. [Necula, 2000] has established its effectiveness over 20 years ago.
   However, the mechanism itself was not proved correct.
2. [Tristan and Leroy, 2008] introduced a formally-verified symbolic execution.
   Alas, their verifier has exponential complexity.

Translation validation of untrusted optimizations

Background and history of the method

Translation validation by symbolic execution

Transforming an input program P_1 into an output program P_2, and using *symbolic execution* to assert *semantic preservation* between P_1 and P_2.

1. [Necula, 2000] has established its effectiveness over 20 years ago.
   However, the mechanism itself was not proved correct.
2. [Tristan and Leroy, 2008] introduced a formally-verified symbolic execution.
   Alas, their verifier has exponential complexity.
3. Our team [Six et al., 2020] solved this issue with formally-verified *hash-consing* within the symbolic execution.
   Applied to certify a scheduler for a VLIW processor (Kalray KVX).

Translation validation of untrusted optimizations

Background and history of the method

Translation validation by symbolic execution

Transforming an input program P_1 into an output program P_2, and using *symbolic execution* to assert *semantic preservation* between P_1 and P_2.

1. [Necula, 2000] has established its effectiveness over 20 years ago.
   However, the mechanism itself was not proved correct.
2. [Tristan and Leroy, 2008] introduced a formally-verified symbolic execution.
   Alas, their verifier has exponential complexity.
3. Our team [Six et al., 2020] solved this issue with formally-verified *hash-consing* within the symbolic execution.
   Applied to certify a scheduler for a VLIW processor (Kalray KVX).

We reuse their verifier to extend the AArch64 backend.

Our contributions on CompCert backend for AArch64

Verified optimizations

1. Peephole optimizer performing **instruction compaction**
2. Basic blocks **postpass scheduling**

Our contributions on CompCert backend for AArch64

Verified optimizations

1. Peephole optimizer performing **instruction compaction**
2. Basic blocks **postpass scheduling**

Adapting a two-tier design from [Six et al., 2020]

1. An **untrusted oracle** performs the peephole then the scheduling
2. The result is verified by a **generic checker** using a Domain Specific Language (AbstractBasicBlock)

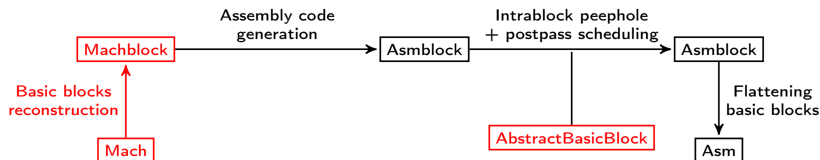
Our contributions on CompCert backend for AArch64

Verified optimizations

1. Peephole optimizer performing **instruction compaction**
2. Basic blocks **postpass scheduling**

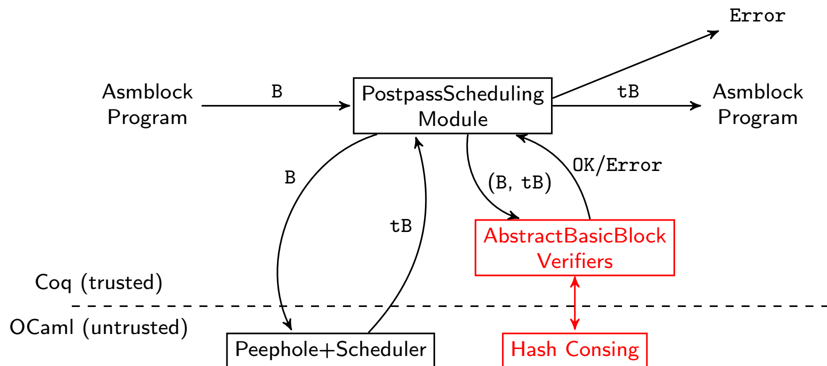
Adapting a two-tier design from [Six et al., 2020]

1. An **untrusted oracle** performs the peephole then the scheduling
2. The result is verified by a **generic checker** using a Domain Specific Language (AbstractBasicBlock)



Architecture of our translation validation

How it works?

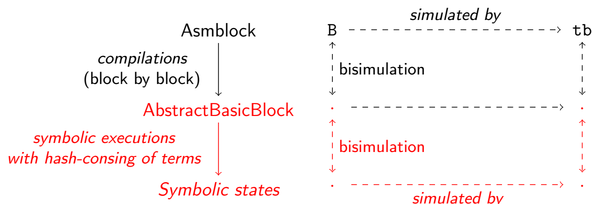


- Generic verifier backend in a Domain Specific Language
- Verified frontend adapted from [Six et al., 2020]
- The verifier proof is independent of the optimization oracles.

DSL for translation validation of basic-block optimizations

Simulation test correctness

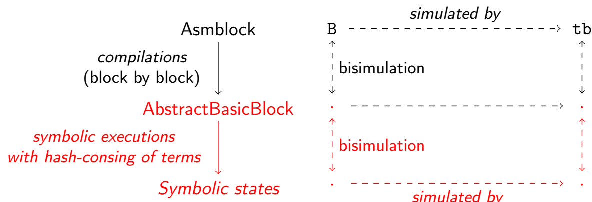
1. Code is translated in the generic AbstractBasicBlock DSL
2. Symbolic execution computes "symbolic states" (aka parallel assign.)
3. Simulation is deduced from syntactical equalities on "symbolic states"



DSL for translation validation of basic-block optimizations

Simulation test correctness

1. Code is translated in the generic AbstractBasicBlock DSL
2. Symbolic execution computes "symbolic states" (aka parallel assign.)
3. Simulation is deduced from syntactical equalities on "symbolic states"



Example: Consider two "abstract" basic blocks B_1 and B_2

(B_1) $r_1 := r_1 + r_2;\ r_3 := \texttt{load}[m, r_1];\ r_3 := r_1;\ r_1 := r_1 + r_3$

(B_2) $r_3 := r_1 + r_2;\ r_1 := r_3 + r_3$

B_1 and B_2 do similar assignments: $r_1 := (r_1 + r_2) + (r_1 + r_2) \parallel r_3 := r_1 + r_2$,

but B_1 **may fail** on $\texttt{load}[m, r_1]$: B_2 *simulates* B_1, but not vice-versa!

Peephole: using load and store pair instructions

Aim: replace two load/store instructions by a single one.
Benefit: more compact code size.

Peephole: using load and store pair instructions

Aim: replace two load/store instructions by a single one.
Benefit: more compact code size.

Briefly

- AArch64 ISA has pair memory transfer instructions
- Those were not modeled nor generated in CompCert
- We provide a semantics, along with a peephole oracle to merge consecutive or not load and store

Peephole: using load and store pair instructions

Aim: replace two load/store instructions by a single one.
Benefit: more compact code size.

Briefly

- AArch64 ISA has pair memory transfer instructions
- Those were not modeled nor generated in CompCert
- We provide a semantics, along with a peephole oracle to merge consecutive or not load and store

Requirements

1. Offsets must be in a defined range
2. Base addressing registers must be equal
3. For load only, destinations must be different

Currently, we only support the classical base+offset addressing mode.

Example 1: merging load and store instructions

```
movz x6, #0, lsl #0
ldr w4, [x6, #0]
sxtw  x3, w0
ldr w1, [x6, #4]
ldr w5, [x3, #0]
ldr w7, [x3, #4]
add w2, w4, w1
adrp  x16, a
```

Example1 before

```
movz x6, #0, lsl #0
ldp w4, w1, [x6, #0]
sxtw  x3, w0
ldp w5, w7, [x3, #0]
add w2, w4, w1
adrp  x16, a
```

Example1 after

1. In **orange** color: backward load pairing, with increasing offset (the offset of the second load is *greater* than that of the first one)
2. In **lime** color: consecutive load pairing, with increasing offset

Example 2: merging load and store instructions

```
mov x0, x19
ldr x19, [sp, #16]
ldr x30, [sp, #8]
movz  x1, #0, lsl #0
str w2, [x1, #0]
movz  w0, #0, lsl #0
str w2, [x1, #4]
sub w0, w0, w2
```

Example2 before

```
mov x0, x19
ldp x30, x19, [sp, #8]
movz  x1, #0, lsl #0
movz  w0, #0, lsl #0
stp w2, w2, [x1, #0]
sub w0, w0, w2
```

Example2 after

1. In **cyan** color: consecutive load pairing, with decreasing offset (the offset of the second load is *lower* than that of the first one)
2. In **olive** color: forward store pairing, with increasing offset

Validating our peephole with the verifier

Reverse rewritings during translation to AbstractBasicBlock

An instruction: a sequence of assignments where `Old(e)` refers to the initial state of the sequence.

Validating our peephole with the verifier

Reverse rewritings during translation to AbstractBasicBlock

An instruction: a sequence of assignments where `Old(e)` refers to the initial state of the sequence.

How to translate load and store in AbstractBasicBlock?

Validating our peephole with the verifier

Reverse rewritings during translation to AbstractBasicBlock

An instruction: a sequence of assignments where `Old(e)` refers to the initial state of the sequence.

How to translate load and store in AbstractBasicBlock?

- `LDR` and `STR` are directly translated.

  ```
  PLd_rd_a ld rd addr
  ```

  is translated as

  ```
  [#rd ← (Oload ld (chunk_load ld) addr [#base; #pmem])]
  ```

Validating our peephole with the verifier

Reverse rewritings during translation to AbstractBasicBlock

An instruction: a sequence of assignments where `Old(e)` refers to the initial state of the sequence.

How to translate load and store in AbstractBasicBlock?

- LDR and STR are directly translated.
  ```
  PLd_rd_a ld rd addr
  ```
  is translated as
  ```
  [#rd ← (Oload ld (chunk_load ld) addr [#base; #pmem])]
  ```
- LDP and STP are expanded back to pairs (i.e. we perform the **reverse** transformation).
  ```
  Pldp ld r1 r2 chk1 chk2 addr
  ```
  is translated as
  ```
  [#r1 ← (Oload ldi1 (ldp_chunk chk1) addr [#base; #pmem]);
  [#r2 ← (Oload ldi2 (ldp_chunk chk2) addr [Old(#base); #pmem])]
  ```

Instruction scheduling

Reordering instructions to increase Instruction Level Parallelism (ILP).

Cortex-A53: two-wide decode superscalar processor

- Two instructions decoded in parallel
- Dual-issue for arithmetic operations

Instruction scheduling

Reordering instructions to increase Instruction Level Parallelism (ILP).

Cortex-A53: two-wide decode superscalar processor

- Two instructions decoded in parallel
- Dual-issue for arithmetic operations

Latencies: time (in cycles) during which the core unit is busy

- Memory transfer instructions have a latency of 3 cycles
- `ADD` and `SUB` have a latency of 1
- **Challenging task:** retrieve latencies for each instruction.

Instruction scheduling

Reordering instructions to increase Instruction Level Parallelism (ILP).

Cortex-A53: two-wide decode superscalar processor

- Two instructions decoded in parallel
- Dual-issue for arithmetic operations

Latencies: time (in cycles) during which the core unit is busy

- Memory transfer instructions have a latency of 3 cycles
- `ADD` and `SUB` have a latency of 1
- **Challenging task:** retrieve latencies for each instruction.

Usually split in two passes : a "coarse-grain" one (before register allocation, see [Six et al., 2021]) and a "fine-grain" one after.

Example: the finer capabilities of postpass

Reordering an instruction expanded at the Asm level

```c
int main(int x, int y) {
  int z = x << 32;
  y = y - z;
  return x + y;
}
```

Source code

```
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

Example: the finer capabilities of postpass

Reordering an instruction expanded at the Asm level

```
int main(int x, int y) {
  int z = x << 32;
  y = y - z;
  return x + y;
}
```

Source code

```
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.

Example: the finer capabilities of postpass

Reordering an instruction expanded at the Asm level

```
int main(int x, int y) {
  int z = x << 32;
  y = y - z;
  return x + y;
}
```

Source code

```
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

- others=1

bad scheduling

EXEC1	EXEC2

running time

Stalls info

Example: the finer capabilities of postpass

Reordering an instruction expanded at the Asm level

```
int main(int x, int y) {
  int z = x << 32;
  y = y - z;
  return x + y;
}
```

Source code

```
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

- others=1

Stalls info

bad scheduling

EXEC1	EXEC2
I_1	

running time

Example: the finer capabilities of postpass

Reordering an instruction expanded at the Asm level

```c
int main(int x, int y) {
  int z = x << 32;
  y = y - z;
  return x + y;
}
```

Source code

```
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
```

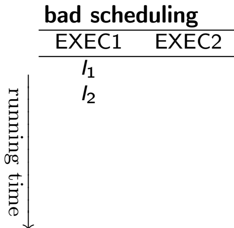
After postpass

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

Latencies

- others=1
- LSL=2

Stalls info



Example: the finer capabilities of postpass

Reordering an instruction expanded at the Asm level

```
int main(int x, int y) {
  int z = x << 32;
  y = y - z;
  return x + y;
}
```

Source code

```
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

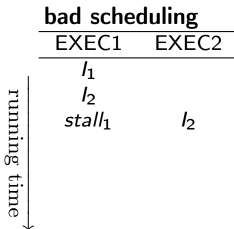
Main difference: the load of the return address is lifted.

Latencies

- others=1
- LSL=2

Stalls info

1. w2 is not ready!



Example: the finer capabilities of postpass

Reordering an instruction expanded at the Asm level

```
int main(int x, int y) {
  int z = x << 32;
  y = y - z;
  return x + y;
}
```

Source code

```
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

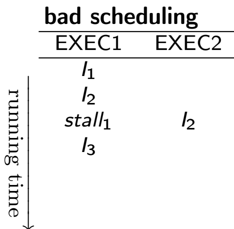
Main difference: the load of the return address is lifted.

Latencies

- others=1
- LSL=2

Stalls info

1. w2 is not ready!



Example: the finer capabilities of postpass

Reordering an instruction expanded at the Asm level

```
int main(int x, int y) {
  int z = x << 32;
  y = y - z;
  return x + y;
}
```

Source code

```
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

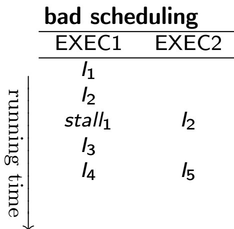
Main difference: the load of the return address is lifted.

Latencies

- others=1
- LSL=2
- LDR=3

Stalls info

1. w2 is not ready!



Example: the finer capabilities of postpass

Reordering an instruction expanded at the Asm level

```
int main(int x, int y) {
  int z = x << 32;
  y = y - z;
  return x + y;
}
```

Source code

```
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

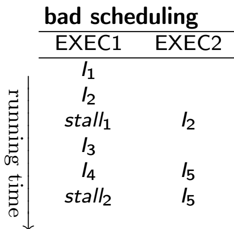
Main difference: the load of the return address is lifted.

Latencies

- others=1
- LSL=2
- LDR=3

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

Example: the finer capabilities of postpass

Reordering an instruction expanded at the Asm level

```
int main(int x, int y) {
  int z = x << 32;
  y = y - z;
  return x + y;
}
```

Source code

```
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

- others=1
- LSL=2
- LDR=3

Stalls info

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

bad scheduling (running time ↓)

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

Example: the finer capabilities of postpass

Reordering an instruction expanded at the Asm level

```c
int main(int x, int y) {
  int z = x << 32;
  y = y - z;
  return x + y;
}
```

Source code

```
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

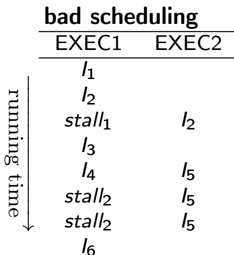
Main difference: the load of the return address is lifted.

Latencies

- others=1
- LSL=2
- LDR=3

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	

Example: the finer capabilities of postpass

Reordering an instruction expanded at the Asm level

```c
int main(int x, int y) {
  int z = x << 32;
  y = y - z;
  return x + y;
}
```

Source code

```
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

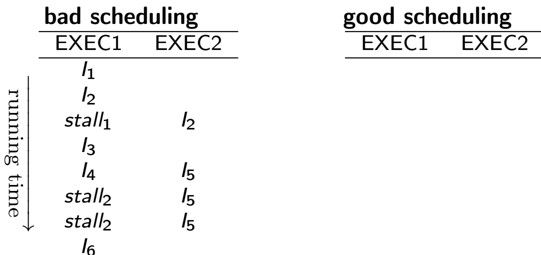
Main difference: the load of the return address is lifted.

Latencies

- others=1
- LSL=2
- LDR=3

Stalls info

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



Example: the finer capabilities of postpass

Reordering an instruction expanded at the Asm level

```c
int main(int x, int y) {
  int z = x << 32;
  y = y - z;
  return x + y;
}
```

Source code

```
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

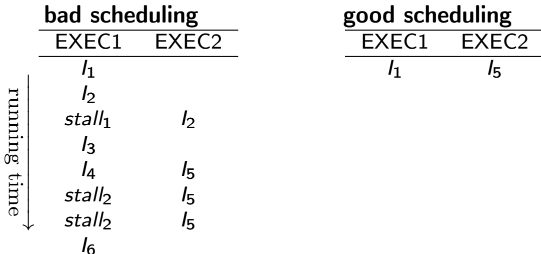
Main difference: the load of the return address is lifted.

Latencies

- others=1
- LSL=2
- LDR=3

Stalls info

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



Example: the finer capabilities of postpass

Reordering an instruction expanded at the Asm level

```
int main(int x, int y) {
  int z = x << 32;
  y = y - z;
  return x + y;
}
```

Source code

```
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

- others=1
- LSL=2
- LDR=3

Stalls info

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

bad scheduling (running time ↓)

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	

good scheduling

EXEC1	EXEC2
I_1	I_5
I_2	I_5

Example: the finer capabilities of postpass

Reordering an instruction expanded at the Asm level

```
int main(int x, int y) {
  int z = x << 32;
  y = y - z;
  return x + y;
}
```

Source code

```
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

- others=1
- LSL=2
- LDR=3

Stalls info

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

bad scheduling (running time ↓)

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	

good scheduling

EXEC1	EXEC2
I_1	I_5
I_2	I_5
I_2	I_5

Example: the finer capabilities of postpass

Reordering an instruction expanded at the Asm level

```
int main(int x, int y) {
  int z = x << 32;
  y = y - z;
  return x + y;
}
```

Source code

```
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

- others=1
- LSL=2
- LDR=3

Stalls info

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

bad scheduling (running time ↓)

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	

good scheduling

EXEC1	EXEC2
I_1	I_5
I_2	I_5
I_2	I_5
I_6	I_3

Example: the finer capabilities of postpass

Reordering an instruction expanded at the Asm level

```
int main(int x, int y) {
  int z = x << 32;
  y = y - z;
  return x + y;
}
```

Source code

```
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

- others=1
- LSL=2
- LDR=3

Stalls info

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

bad scheduling (running time ↓)

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	

good scheduling

EXEC1	EXEC2
I_1	I_5
I_2	I_5
I_2	I_5
I_6	I_3
I_4	

Example: the finer capabilities of postpass

Reordering an instruction expanded at the Asm level

```
int main(int x, int y) {
  int z = x << 32;
  y = y - z;
  return x + y;
}
```

Source code

```
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

- others=1
- LSL=2
- LDR=3

Stalls info

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

bad scheduling (running time ↓)

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	

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!

Proof effort and benefits

Overall implementation: three man·months of development.

- Three main translation proofs in Coq
- The verifier frontend implementation for AArch64 (translation & proof) represents about 2Kloc
- The verifier backend, which is architecture independent, fits in 2.2Kloc

Proof effort and benefits

Overall implementation: three man·months of development.

- Three main translation proofs in Coq
- The verifier frontend implementation for AArch64 (translation & proof) represents about 2Kloc
- The verifier backend, which is architecture independent, fits in 2.2Kloc

Bug found in the CompCert Asm specification[2]

- Unsound formal specification of Asm w.r.t. the "Asm pretty-printer"
- Concerns `Pfmovimmd` and `Pfmovimms` macro-instructions
- Instruction behavior was not fully specified

[2] Since, this bug has been patched in the CompCert mainline repository

Contents

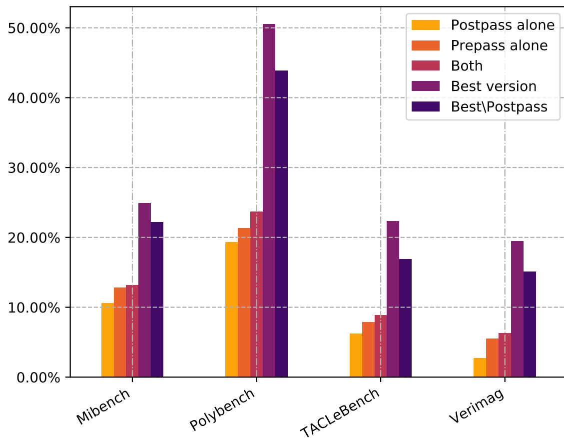
1 Lightweight reasoning on pointer equalities and hash-consing in Coq

2 Realistic applications to formally-verified compilation

3 Experimental results and Conclusion

Running-time gains on the generated code (for Cortex-A53)

Combined with others optimizations



- Our oracle alone raises performance by 9.7% across all our benchmarks
- The "Best" version[3] makes us reach 29.3% in average
- Improvement of "Best" *without* postpass falls back to 24.5%

[3] i.e. LICM+Prepass+Postpass+Tail duplication+Loop unrolling+Loop rotate

Other results

CompCert vs. GCC running times & Code size reduction

GCC w.r.t. "Best" CompCert version (for running times)

- Close to GCC-O1 (1.55% slower in average)
- Still far from GCC-02 (14.5% slower in average)

Benchmark suite	GCC-O1	GCC-O2
Mibench	-6.3%	+9.7%
Polybench	+2.8%	+7.6%
TACLeBench	+6.3%	+24.9%
Verimag	+0.3%	+15.8%

Other results

CompCert vs. GCC running times & Code size reduction

GCC w.r.t. "Best" CompCert version (for running times)

- Close to GCC-O1
  (1.55% slower in average)
- Still far from GCC-02
  (14.5% slower in average)

Benchmark suite	GCC-O1	GCC-O2
Mibench	-6.3%	+9.7%
Polybench	+2.8%	+7.6%
TACLeBench	+6.3%	+24.9%
Verimag	+0.3%	+15.8%

Asm code length reduction with peephole

Benchmark suite	CompCert Gain	CompCert Gain (load/store)
Mibench	3.44%	13.78%
Polybench	2.56%	11.14%
TACLeBench	1.31%	8.93%
Verimag	2.51%	10.74%

Conclusion and future work

This work was a first step for future research on a posteriori validated optimizations:

- Define more general, and more powerful DSL verifiers
- Working on bigger blocks (e.g. extended blocks)
- Strength reduction
- Lifting Asm level expansions

Apart from architecture independent optimizations, we are also interested in the rising open-hardware RISC-V ISA.

Thank you!

Questions?

Please visit our Gitlab repository:
`https://gricad-gitlab.univ-grenoble-alpes.fr/certicompil/compcert-kvx`

References

Filliâtre, J. and Conchon, S. (2006).
Type-safe modular hash-consing.
In *Proceedings of the ACM Workshop on ML, 2006*, pages 12–19. ACM.

Fouilhé, A. and Boulmé, S. (2014).
A certifying frontend for (sub)polyhedral abstract domains.
In *Verified Software: Theories, Tools and Experiments, (VSTTE)*, volume 8471 of *Lecture Notes in Computer Science*, pages 200–215. Springer.

King, J. C. (1976).
Symbolic execution and program testing.
Commun. ACM, 19(7):385–394.

Necula, G. C. (2000).
Translation validation for an optimizing compiler.
In *Programming Language Design and Implementation (PLDI)*, pages 83–94. ACM Press.

Six, C., Boulmé, S., and Monniaux, D. (2020).
Certified and efficient instruction scheduling. Application to interlocked VLIW processors.
PACMPL (OOPSLA 2020).

Six, C., Gourdin, L., Boulmé, S., and Monniaux, D. (2021).
Verified Superblock Scheduling with Related Optimizations.
preprint.

Tristan, J. and Leroy, X. (2008).
Formal verification of translation validators: a case study on instruction scheduling optimizations.
In *POPL*, pages 17–27. ACM Press.