Certified and modular postpass scheduling for VLIW processors in CompCert

Cyril Six (CIFRE PhD student)
Supervised by:
Sylvain Boulmé (Verimag PACSS)
Benoît Dupont de Dinechin (Kalray)
David Monniaux (Verimag PACSS)

cyril.six@univ-grenoble-alpes.fr
sylvain.boulme@univ-grenoble-alpes.fr
benoit.dinechin@kalray.eu
david.monniaux@univ-grenoble-alpes.fr

8th of July, 2019
About us

- PACSS team - Proofs and Code analysis for Safety and Security
  - David Monniaux, CNRS senior researcher
  - Sylvain Boulmé, researcher

- Kalray - Fabless semiconductor company based in Grenoble
  - Benoît Dupont de Dinechin, CTO

- Me
  - ENSIMAG school, then 1 year as engineer in INRIA CORSE (Compiler Optimizations and Runtime Systems)
  - Now: CIFRE PhD student, 1 year and 5 months
Presentation plan

1 Introduction
   - VLIW in-order processors
   - CompCert architecture

2 Contributions
   - Formal blockstep semantics for VLIW
   - Certified intrablock postpass scheduling

3 Results
   - Experimentations
   - Future work
Outline

1. Introduction
   - VLIW in-order processors
   - CompCert architecture

2. Contributions

3. Results
1 processor = 5 compute clusters
1 compute clusters = 16 cores @ (600MHz - 1.2GHz)
Network on Chip allowing point-to-point communication
Main DDR memory of 4 GB
Kalray VLIW k1c core

- ALU: Arithmetic-Logic Unit (x2)
- LSU: Load-Store Unit
- MAU: Multiply-Accumulate Unit
- BCU: Branch Control Unit

- 64x64-bit user registers per core
- Very Large Instruction Word (VLIW): explicit Instruction Level Parallelism
- 5 execution units: ALU0, ALU1, LSU, MAU, BCU
- In-order, pipelined execution
Example of k1c code

```
addw $r2 = $r1, $r0          /* ALU */
;;                         /* bundle delimiter */
mulu $r2 = $r2, 2
addw $r0 = $r2, $r1          /* MAU + ALU */
;;
addw $r0 = $r1, 0
addw $r1 = $r0, 0            /* ALU + ALU */
;;
mulu $r1 = $r1, 2
addw $r3 = $r2, 42
j toto                      /* BCU + ALU + MAU */
```

- Bundles are explicitly delimited by the programmer/compiler
- In-order execution
In-order vs out-of-order

- More predictable, more precise computation of Worst Case Execution Time (WCET)
- Simpler control structure
  - Uses less CPU die space and energy
  - May be more reliable (cf Intel Skylake hyperthreading hardware bug)

=> Good for safety-critical applications
K1c pipeline

```c
int add4(int *t){
}
```

add4:
```assembly
lwz $r1 = 0[$r0]
;;
lwz $r4 = 4[$r0]
;;
/* 2 cycles stall ($r4) */
addw $r1 = $r1, $r4
;;
lwz $r3 = 8[$r0]
;;
/* 2 cycles stall ($r3) */
addw $r0 = $r1, $r3
;;
lwz $r2 = 12[$r0]
;;
/* 2 cycles stall ($r2) */
addw $r0 = $r0, $r2
;;
ret
;;
```

13 cycles
### K1c pipeline

```c
int add4(int *t) {
}
```

**add4:**
```
lwz $r1 = 0[$r0] ;;
lwz $r4 = 4[$r0] ;;
/* 2 cycles stall ($r4) */
addw $r1 = $r1, $r4 ;;
lwz $r3 = 8[$r0] ;;
/* 2 cycles stall ($r3) */
addw $r0 = $r1, $r3 ;;
lwz $r2 = 12[$r0] ;;
/* 2 cycles stall ($r2) */
addw $r0 = $r0, $r2 ;;
ret ;;
```

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Issue</th>
<th>Read Regs</th>
<th>E1</th>
<th>E2</th>
<th>E3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>lwz[r1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

13 cycles

---

Cyril Six (PACSS & Kalray)  
CompCert VLIW scheduling  
8th of July, 2019  
9 / 43
# K1c pipeline

```c
int add4(int *t)
{
}
```

```
add4:
    lwz $r1 = 0[$r0] ;
    lwz $r4 = 4[$r0] ;
    /* 2 cycles stall ($r4) */
    addw $r1 = $r1, $r4 ;
    lwz $r3 = 8[$r0] ;
    /* 2 cycles stall ($r3) */
    addw $r0 = $r1, $r3 ;
    lwz $r2 = 12[$r0] ;
    /* 2 cycles stall ($r2) */
    addw $r0 = $r0, $r2 ;
    ret ;
/*
   Cycle | Issue | Read Regs | E1 | E2 | E3
   1     | lwz\r_0 \n   2     | lwz\r_4 \n           lwz\r_1

13 cycles
```
```c
int add4(int *t)
{
}
```

add4:

```c
lwz $r1 = 0[$r0]
;;
lwz $r4 = 4[$r0]
;;
/* 2 cycles stall ($r4) */
addw $r1 = $r1, $r4
;;
lwz $r3 = 8[$r0]
;;
/* 2 cycles stall ($r3) */
addw $r0 = $r1, $r3
;;
lwz $r2 = 12[$r0]
;;
/* 2 cycles stall ($r2) */
addw $r0 = $r0, $r2
;;
ret
;;
```

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Issue</th>
<th>Read Regs</th>
<th>E1</th>
<th>E2</th>
<th>E3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>lwz(r_0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>lwz(r_4)</td>
<td>lwz(r_0)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>add(r_1, r_4)</td>
<td>lwz(r_0)</td>
<td>lwz(r_1)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

13 cycles
int add4(int *t) {
}

add4:
  lwz $r1 = 0[$r0] ;
  lwz $r4 = 4[$r0] ;
  /* 2 cycles stall ($r4) */
  addw $r1 = $r1, $r4 ;
  lwz $r3 = 8[$r0] ;
  /* 2 cycles stall ($r3) */
  addw $r0 = $r1, $r3 ;
  lwz $r2 = 12[$r0] ;
  /* 2 cycles stall ($r2) */
  addw $r0 = $r0, $r2 ;
  ret ;

13 cycles
```c
int add4(int *t)
{
}
```

```
add4:
    lwz $r1 = 0[$r0]
    lwz $r4 = 4[$r0]
    // 2 cycles stall ($r4)
    addw $r1 = $r1, $r4
    lwz $r3 = 8[$r0]
    // 2 cycles stall ($r3)
    addw $r0 = $r1, $r3
    lwz $r2 = 12[$r0]
    // 2 cycles stall ($r2)
    addw $r0 = $r0, $r2
    ret

13 cycles
```
```c
int add4(int *t) {
}
```

```
add4:
lwz $r1 = 0[$r0]
;;
lwz $r4 = 4[$r0]
;;
/* 2 cycles stall ($r4) */
addw $r1 = $r1, $r4
;;
lwz $r3 = 8[$r0]
;;
/* 2 cycles stall ($r3) */
addw $r0 = $r1, $r3
;;
lwz $r2 = 12[$r0]
;;
/* 2 cycles stall ($r2) */
addw $r0 = $r0, $r2
;;
ret
;;
```

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Issue</th>
<th>Read Regs</th>
<th>E1</th>
<th>E2</th>
<th>E3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>lwz</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>lwz</td>
<td>lwz</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>add</td>
<td>lwz</td>
<td>lwz</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>lwz</td>
<td>add</td>
<td>lwz</td>
<td>lwz</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>lwz</td>
<td>add</td>
<td>STALL</td>
<td></td>
<td>lwz</td>
</tr>
<tr>
<td>6</td>
<td>lwz</td>
<td>add</td>
<td>STALL</td>
<td>STALL</td>
<td>lwz</td>
</tr>
</tbody>
</table>

13 cycles
```c
int add4(int *t) {
}
```

```
add4:
    lwz $r1 = 0[$r0]
    lwz $r4 = 4[$r0]
    /* 2 cycles stall ($r4)

addw $r0 = $r1, $r4
    lwz $r3 = 8[$r0]
    /* 2 cycles stall ($r3)

addw $r0 = $r1, $r3
    lwz $r2 = 12[$r0]
    /* 2 cycles stall ($r2) */
addw $r0 = $r0, $r2
    ret
```

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Issue</th>
<th>Read Regs</th>
<th>E1</th>
<th>E2</th>
<th>E3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>lwz$r0$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>lwz$r4$</td>
<td>lwz$r0$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>/* 3 */</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>add$r1,$r4</td>
<td>lwz$r0$</td>
<td>lwz$r1$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>lwz$r0$</td>
<td>add$r1,$r4</td>
<td>lwz$r0$</td>
<td></td>
<td>lwz$r1$</td>
</tr>
<tr>
<td>5</td>
<td>lwz$r3$</td>
<td>add$r1,$r4</td>
<td>STALL</td>
<td>lwz$r4$</td>
<td>lwz$r1$</td>
</tr>
<tr>
<td>6</td>
<td>lwz$r3$</td>
<td>add$r1,$r4</td>
<td>STALL</td>
<td>STALL</td>
<td>lwz$r4$</td>
</tr>
<tr>
<td></td>
<td>/* 7 */</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>add$r1,$r3</td>
<td>lwz$r0$</td>
<td>add$r1,$r4</td>
<td>STALL</td>
<td>STALL</td>
</tr>
</tbody>
</table>

13 cycles
K1c pipeline

```c
int add4(int *t){
}
```

```
add4:
    lwz $r1 = 0[$r0];
    lwz $r4 = 4[$r0];
    /* 2 cycles stall ($r4) */
    addw $r1 = $r1, $r4;
    lwz $r3 = 8[$r0];
    /* 2 cycles stall ($r3) */
    addw $r0 = $r1, $r3;
    lwz $r2 = 12[$r0];
    /* 2 cycles stall ($r2) */
    addw $r0 = $r0, $r2;
    ret;
```

```
add4:
    lwz $r1 = 0[$r0];
    lwz $r4 = 4[$r0];
    lwz $r3 = 8[$r0];
    lwz $r2 = 12[$r0];
    addw $r1 = $r1, $r4;
    addw $r0 = $r1, $r3;
    addw $r0 = $r0, $r2;
    ret;
```

13 cycles  8 cycles
Compilers and VLIW processors

- VLIW compilers should be able to generate bundles to have a good performance
- Instructions should also be reordered to minimize the latencies
- This is usually done by a scheduling pass, after register allocation

### Code to schedule...

```assembly
add4:
  lwz $r1 = 0[$r0]  # 0
  lwz $r4 = 4[$r0]  # 1
  addw $r1 = $r1, $r4 # 3
  lwz $r3 = 8[$r0]  # 2
  addw $r0 = $r1, $r3 # 5
  lwz $r2 = 12[$r0] # 3
  addw $r0 = $r0, $r2 # 6
  ret   # 6
```

### After scheduling

```assembly
add4:
  lwz $r1 = 0[$r0]
  ;;
  lwz $r4 = 4[$r0]
  ;;
  lwz $r3 = 8[$r0]
  ;;
  lwz $r2 = 12[$r0]
  ;;
  addw $r1 = $r1, $r4
  ;;
  addw $r0 = $r1, $r3
  ;;
  addw $r0 = $r0, $r2
  ret
  ;;
```
Register allocation and scheduling

- Register allocation
  - Allocates physical registers (bounded) to virtual registers (unbounded)
  - Performs “spilling” if not able to

Before register allocation

```
add4:
    lwz R1 = 0[R0]
    ;;
    lwz R2 = 4[R0]
    ;;
    addw R3 = R1, R2
    ;;
    lwz R4 = 8[R0]
    ;;
    addw R5 = R3 R4
    ;;
    lwz R6 = 12[R0]
    ;;
    addw R7 = R6, R5
    ;;
    ret
    ;;
```

After a register allocation

```
add4:
    lwz $r1 = 0[$r0]
    ;;
    lwz $r2 = 4[$r0]
    ;;
    addw $r1 = $r1, $r2
    ;;
    lwz $r2 = 8[$r0]
    ;;
    addw $r1 = $r1, $r2
    ;;
    lwz $r0 = 12[$r0]
    ;;
    addw $r0 = $r0, $r0
    ;;
    ret
    ;;
```
Register allocation and scheduling

- **Register allocation**
  - Allocates physical registers (bounded) to virtual registers (unbounded)
  - Performs “spilling” if not able to

- **Instruction scheduling**
  - After register allocation (postpass): more precise informations, can make bundles, but extra dependencies on registers
  - To deal with the register dependencies: instruction scheduling before register allocation (prepass)

- So far, we did a postpass scheduling optimization
1. Introduction
   - VLIW in-order processors
   - CompCert architecture

2. Contributions

3. Results
CompCert in a few words

- Machine checked, formally verified compiler: proof of specification preservation
- Written in Coq and OCaml
- Targets: PPC, ARM, RISC-V, x86
- Performance: close to GCC -O1
CompCert architecture

What we want: intrablock postpass scheduling at Asm level
Our modifications to the CompCert architecture

- Machblock and Asmblock: one block = one basic block, sequential semantics
- AsmVLIW: one block = one bundle, parallel semantics within a bundle
Outline

1 Introduction

2 Contributions
   • Formal blockstep semantics for VLIW
   • Certified intrablock postpass scheduling

3 Results
State: \((rs, m)\)

- Registers state \(rs\): mapping from registers (PC, RA, r1, r2, ..) to values
- Memory state \(m\): mapping from addresses to values

- \(\text{exec\_instr}\): function \(\rightarrow\) instruction \(\rightarrow\) regset \(\rightarrow\) mem \(\rightarrow\) outcome
- outcome: either Stuck, or Next \(rs'\) \(m'\)
- Instructions reside in memory, and are pointed by PC register
- Exemples of execution:
  - \((\text{exec\_instr } f (\text{Pcall } s) rs m)\) returns \((rs[RA \leftarrow rs[PC]; PC \leftarrow @s], m)\)
  - \((\text{exec\_instr } f (\text{Padd } r0 r1 r2) rs m)\) returns \((rs[r0 \leftarrow rs[r1] + rs[r2]], m)\)
Formal definition of a basic block

**Inductive basic**: Type := (* basic instructions *)

**Inductive control**: Type := (* control–flow instructions *)

**Record bblock** := {
  header: list label; body: list basic; exit: option control;
  correct: wf_bblock body exit (* must contain at least 1 instr. *)
}

- **Exemples**: Pcall is a *control*, Padd is a *basic*.

- **Executing the block** ($R_0 \leftarrow R_1; R_1 \leftarrow R_0; \text{jump } \@\text{toto}$) should lead to
  \[
  \text{rs}[R_0] \leftarrow \text{rs}[R_1]; R_1 \leftarrow \text{rs}[R_0]; PC \leftarrow \@\text{toto}
  \]

- **Details of the actual bundle execution**: 
  - All the reads are done at the RR pipeline stage (before any write)
  - Problem: Two concurrent writes on the same register lead to non-determinism
Parallel in-order semantics (1)

- Instead of \((rs, m)\), we use four components in a bundle execution:
  - \(rsr, mr\): the regset and the memory state, prior to executing the bundle
  - \(rsw, mw\): the running state, where all the writes will occur (in order)

\[
(rsr, mr) \xrightarrow{bstep} (rsw_{1}, mw_{1}) \xrightarrow{bstep} \cdots \xrightarrow{bstep} (rsw_{n}, mw_{n}) \xrightarrow{estep} (rsw_{n+1}, mw_{n+1})
\]

\(bstep\) parexec wio \([b_1; b_2; \cdots; b_n]\) ext \(sz\)

- This first version models an atomic parallel execution, where:
  - All reads are done in parallel, prior to any write
  - The writes are done sequentially in the same order

- Example: \((R_0 := R_1; R_1 := R_0; \text{jump @toto})\)
  - \(rsw_1 = rsr[R_0 \leftarrow rsr[R_1]] = rs[R_0 \leftarrow r_1]\)
  - \(rsw_2 = rsw_1[R_1 \leftarrow rsr[R_0]] = rs[R_0 \leftarrow r_1; R_1 \leftarrow r_0]\)
  - \(rs' = rsw_3 = rsw_2[\text{PC} \leftarrow @toto] = rs[R_0 \leftarrow r_1; R_1 \leftarrow r_0; \text{PC} \leftarrow @toto]\)
Parallel in-order semantics (2)

Fixpoint parexec_wio_body bdy rsr rsw mr mw : outcome :=
    match bdy with
    | nil => Next rsw mw
    | bi::bdy' => NEXT rsw', mw' ← bstep bi rsr rsw mr mw
        IN parexec_wio_body bdy' rsr rsw' mr mw'
    end.

Definition parexec_wio f bdy ext sz rs m :=
    NEXT rsw', mw' ← parexec_wio_body bdy rs rs m m
    IN estep f ext sz rs rsw' mw'.
(parexec_bblock \ f \ b \ rs \ m \ o) \ holds \ if \ there \ exists \ a \ permutation \ of \ writes \ that \ gives \ o \ by \ a \ deterministic \ in-order \ execution

- We can reason on permutations of instructions instead of permutations of writes

```
Definition parexec_bblock f b rs m o : Prop :=
 exists bdy1 bdy2, Permutation (bdy1 ++ bdy2) b (body) /
 o=(NEXT rsw', mw' ← parexec_wio f bdy1 b (exit)
   (Ptrofs.repr (size b)) rs m
   IN parexec_wio_body bdy2 rs rsw' m mw').
```

- We would like to determinize it, to use in CompCert
(det_parexec f b rs m rs’ m’) holds if: (rs’, m’) is the unique outcome of the non-deterministic parallel execution

Definition det_parexec f b rs m rs’ m': Prop :=
  forall o, parexec_bbblock f b rs m o -> o = Next rs’ m’.

Remark: in this semantic, Stuck executions cannot happen
Asmblock sequential semantics

- First approach: copy/paste AsmVLIW, but with the usual “Asm” sequential execution
  - All instruction specifications need to be duplicated..
- Our approach: get it directly from the bstep and estep of the AsmVLIW parallel semantics

```ocaml
Fixpoint exec_body bdy rs m: outcome :=
  match body with
  | nil => Next rs m
  | bi::bdy' => NEXT rs' m' ← bstep bi rs rs m m
             IN exec_body bdy' rs' m'
  end.

Definition exec_bblock f b rs m: outcome :=
  NEXT rs' m' ← exec_body b.(body) rs m
  IN estep f b.(exit) (Ptrofs.repr (size b)) rs' rs' m'.
```
So far..

How to reorder a block from Asmblock into several bundles of AsmVLIW, and prove it correct?
Outline

1 Introduction

2 Contributions
   • Formal blockstep semantics for VLIW
   • Certified intrablock postpass scheduling

3 Results
Forward simulations in CompCert

- Simulation diagrams are used to prove semantic preservation
- For this transformation, we use the *Lock-step* and the *Plus* simulations

**Lock-step simulation**

\[ L_1 \{ t \downarrow S_1 \sim \ldots \sim S_2 \downarrow t \} \]

\[ L_1 \{ L_1 \{ t \downarrow S_1 \sim \ldots \sim S_2 \downarrow t \} \}

- Lock: If \( S_1 \xrightarrow{t} S'_1 \) and \( S_1 \sim S_2 \), then \( \exists S'_2, S_2 \xrightarrow{t} S'_2 \) and \( S'_1 \sim S'_2 \).

**"Plus" simulation**

\[ L_1 \{ t \downarrow S_1 \sim \ldots \sim S_2 \downarrow t + \} \]

\[ L_1 \{ L_1 \{ t \downarrow S_1 \sim \ldots \sim S_2 \downarrow t + \} \}

Cyril Six (PACSS & Kalray)
Previous work of J.B. Tristan on scheduling

- Gallium PhD 2009 by J.B. Tristan, *Formal verification of translation validators*

![Diagram](image)

- Verification proof: $V(c_1, c_2) = true \implies c_1 \sim c_2$
- Advantages: easier to prove and modular
- Implemented it at the Mach level, with symbolic evaluation
- Drawback (at the time): scalability issue
Axiom schedule: bblock → list bblock.

Forward simulation in two parts:
  - Proving reordering, in sequential semantics: *plus simulation*
  - Proving for each bundle that parallel execution = sequential execution: *lockstep simulation*
The AbstractBasicBlock language is parametrized by:
- Pseudo registers, operators, values, an environment \( \text{genv} \)
- \( \text{op}\_\text{eval}: \text{genv} \rightarrow \text{op} \rightarrow \text{list value} \rightarrow \text{option value} \)

A state of AbstractBasicBlock is a mapping from pseudo registers to values.

Expressions are defined as combination of pseudo registers and operators:

\[
\text{Inductive exp :=}
\]
\[
\mid \text{Read (x:R.t)} \mid \text{Op (o:op) (le : list_exp)} \mid \text{Old (e : exp)}
\]
\[
\text{with list_exp :=}
\]
\[
\mid \text{Enil} \mid \text{Econs (e : exp) (le : list_exp)} \mid \text{LOld (le : list_exp)}.
\]

\[
\text{Definition inst := list (R.t * exp). ( list of assignments )}
\]

The assignments can be executed sequentially or in parallel.

Examples of traductions:
- Picall \( r \rightarrow [\#RA \leftarrow \text{Read(\#PC)}; \ PC \leftarrow \text{Read(\#r)}] \)
- Paddw \( r0 \ r1 \ r2 \rightarrow [\#r0 \leftarrow (\text{OpAddd}[\#r1; \ #r2])] \)

Defines a certified function \( \text{is}_\text{parallelizable : list inst \rightarrow bool} \)
Translation from AsmVLIW to AbstractBasicBlock:

\[ \text{trans\_block}: \text{bblock} \rightarrow (\text{list inst}) \]

We prove a bisimulation for sequential, and a bisimulation for parallel semantics

Bisimulation for sequential:

\[ \text{match\_states (State rs m) s } \rightarrow \]
\[ \text{match\_outcome (exec\_bblock ge fn b rs m)} \]
\[ (\text{exec Ge (trans\_block b) s}). \]

Bisimulation for parallel:

\[ \text{match\_states (State rs1 m1) s1'} \rightarrow \]
\[ \text{parexec\_bblock ge fn b rs1 m1 o2 } \rightarrow \]
\[ \text{exists o2', prun Ge (trans\_block b) s1' o2'} \]
\[ /\text{ match\_outcome o2 o2'}. \]
Parallelizability checker through AbstractBasicBlock

- We translate the bundle to a block of AbstractBasicBlock
- We prove the following theorem with the sequential bisimulation + parallel bisimulation + correctness of is_parallelizable + other minor lemmas:

\[
\text{bblock\_para\_check bundle } = \text{true } \rightarrow \\
\text{exec\_bblock ge f bundle rs m = Next rs' m' } \rightarrow \\
\text{det\_parexec ge f bundle rs m rs' m'}. 
\]
Towards proving reordering

- Definition of bblock simulation:

  \[
  \text{Definition } \texttt{bblock\_simu } ge \ f \ b \ b' := \\
  \forall rs \ m, \ \texttt{exec\_bblock } ge \ f \ b \ rs \ m \leftrightarrow \text{Stuck } \rightarrow \\
  \texttt{exec\_bblock } ge \ f \ b' \ rs \ m = \texttt{exec\_bblock } ge \ f \ b' \ rs \ m.
  \]

- Definition of a concatenation function, and a predicate \((\text{is\_concat } b \ lb)\), where \(lb\) is a list bblock.

- Definition of a function \((\text{verified\_schedule } b)\) that:
  - Calls the oracle, retrieving a list of bundles
  - Concatenates together the bundles to form a bblock \(B\)
  - Calls the reordering verifier from AbstractBasicBlock (detailed later)

- We then prove the following property:

  \[
  \text{Theorem } \text{verified\_schedule\_correct : } \forall ge \ f \ B \ lb, \\
  (\text{verified\_schedule } B) = (\text{OK } lb) \rightarrow \\
  \exists tb, \text{is\_concat } tb \ lb \land bblock\_simu ge \ f \ B \ tb.
  \]
Symbolic execution: computing symbolic memories, final value of the pseudo register in function of the initial values.

Example:

1. $B_1 = [r_1 := r_1 + r_2; \ r_3 := load[r_2, m]; \ r_1 := r_1 + r_3]$
2. $B_2 = [r_3 := load[r_2, m]; \ r_1 := r_1 + r_2; \ r_1 := r_1 + r_3]$

These two blocks are equivalent to this assignment:

$$[r_1 \leftarrow (r_1 + r_2) + load[r_2, m] \ || \ r_3 \leftarrow load[r_2, m]]$$
Intrablock Reordering verifier of AbstractBasicBlock

Proof that symbolic executions bisimulate the sequential executions of AbstractBasicBlock

Issue: symbolic execution involves computing subtrees of expressions, which can lead to exponential complexity

Solution: memoization of terms by hash consing
Memoization involves calling an untrusted OCaml oracle to give memoized terms out of terms.

- Dynamic checker in Coq that ensures the memoized term and the term have the same evaluation function.
- Check done with OCaml pointer equality.

Axiom: if pointer equality returns true, then the two values are structurally equals.
The untrusted scheduler in a few words

- We want to assign a scheduling date to each instruction
- This schedule should satisfy two constraints:
  - Latency constraints: we must respect the dependencies of each instruction
  - Resource constraints: a bundle must not consume more resources than available
- (drawing on board)

- We implemented several schedulers:
  - Naive greedy one, just packs instructions together (linear time)
  - Variant of Coffman-Graham list scheduler, with critical path heuristic (linear time)
  - Optimal list scheduler by Integer Linear Programming + branch and bound (not linear)

- Experimentally, we barely get better results with ILP than with list scheduling
Outline

1 Introduction

2 Contributions

3 Results
   - Experimentations
   - Future work
Timings obtained by instrumenting the OCaml code
Impact of scheduling: performance gain between 20% and 100%
Results to take with a pinch of salt: GCC backend still in development
- Always better than -O0: 2 to 17 times better
- For most benchmarks, faster than -O1 by 20%. Sometimes better than -O2 and -O3.
- For most others, between 20% and 30% slower than GCC -O3
Optimizations that GCC do compared to us

- Certain strength reductions (replacing multiplication in loop by addition)
- Code motion across basic blocks (e.g. loop invariant code motion)
- Loop unrolling and other loop optimizations
- Prepass scheduling
Outline

1. Introduction
2. Contributions
3. Results
   - Experimentations
   - Future work
Future work

- Prepass intrablock scheduling in RTL
  - RTLblock: generating blocks that conserve the control flow structure
  - Blocks with single entry point, single exit

- Integrate memory alias analysis in the checker
  - Instead of viewing memory as a single pseudo register, have something more elaborate

- Other more complex optimizations
  - Superblock scheduling (one entry point, several exits)?
  - Loop invariant code motion?
  - Loop unrolling?
  - Memory alias analysis, propagated through the IRs?
Thanks for your attention

Do you have any questions?