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

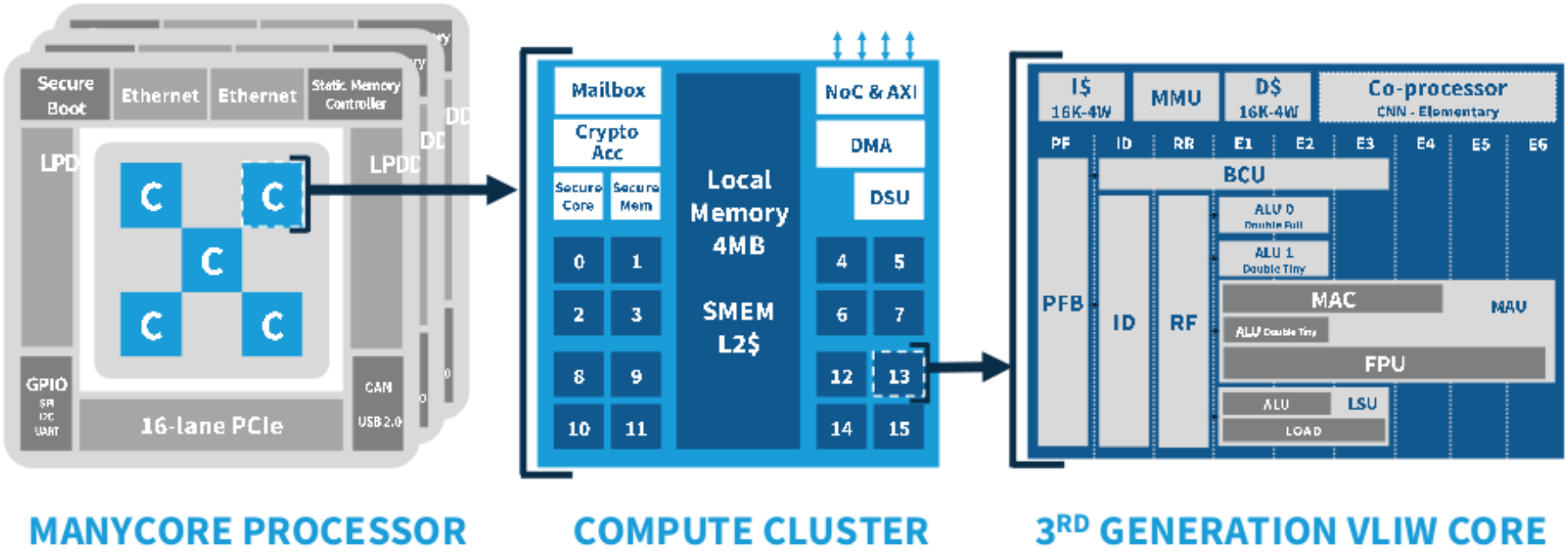
- PACSS team - Proofs and Code analysis for Safety and Security
  - David Monniaux, CNRS senior researcher
  - Sylvain Boulmé, researcher
- Kalray - Fabless semiconductor company based in Grenoble
  - Benoît Dupont de Dinechin, CTO
- Me
  - ENSIMAG school, then 1 year as engineer in INRIA CORSE (Compiler 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

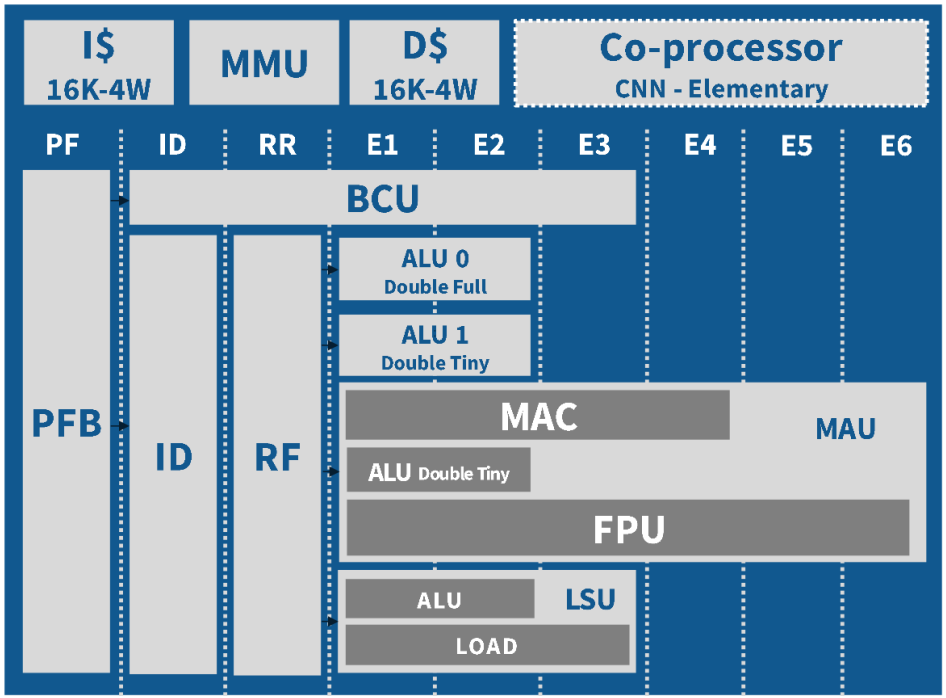
Outline

Kalray processor



- 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 delimitor */
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 predictible, 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

```
int add4(int *t){
  return (t[0] + t[1] + t[2] + t[3]);
}
```

```
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

K1c pipeline

```
int add4(int *t){
  return (t[0] + t[1] + t[2] + t[3]);
}
```

```
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	$lwz\ {}^{r0}_{r1}$				

13 cycles

K1c pipeline

```
int add4(int *t){
  return (t[0] + t[1] + t[2] + t[3]);
}
```

```
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^{r0}_{r1}				
2	lwz^{r0}_{r4}	lwz^{r0}_{r1}			

13 cycles

K1c pipeline

```
int add4(int *t){
  return (t[0] + t[1] + t[2] + t[3]);
}
```

```
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^{r0}_{r1}				
2	lwz^{r0}_{r4}	lwz^{r0}_{r1}			
3	$add^{r1,r4}_{r1}$	lwz^{r0}_{r4}	lwz^{r0}_{r1}		

13 cycles

K1c pipeline

```
int add4(int *t){
  return (t[0] + t[1] + t[2] + t[3]);
}
```

```
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^{r0}_{r1}				
2	lwz^{r0}_{r4}	lwz^{r0}_{r1}			
3	$add^{r1,r4}_{r1}$	lwz^{r0}_{r4}	lwz^{r0}_{r1}		
4	lwz^{r0}_{r3}	$add^{r1,r4}_{r1}$	lwz^{r0}_{r4}	lwz^{r0}_{r1}	

13 cycles

K1c pipeline

```
int add4(int *t){
  return (t[0] + t[1] + t[2] + t[3]);
}
```

```
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^{r0}_{r1}				
2	lwz^{r0}_{r4}	lwz^{r0}_{r1}			
3	$add^{r1,r4}_{r1}$	lwz^{r0}_{r4}	lwz^{r0}_{r1}		
4	lwz^{r0}_{r3}	$add^{r1,r4}_{r1}$	lwz^{r0}_{r4}	lwz^{r0}_{r1}	
5	lwz^{r0}_{r3}	$add^{r1,r4}_{r1}$	*STALL*	lwz^{r0}_{r4}	lwz^{r0}_{r1}

13 cycles

K1c pipeline

```
int add4(int *t){
  return (t[0] + t[1] + t[2] + t[3]);
}
```

```
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^{r0}_{r1}				
2	lwz^{r0}_{r4}	lwz^{r0}_{r1}			
3	$add^{r1,r4}_{r1}$	lwz^{r0}_{r4}	lwz^{r0}_{r1}		
4	lwz^{r0}_{r3}	$add^{r1,r4}_{r1}$	lwz^{r0}_{r4}	lwz^{r0}_{r1}	
5	lwz^{r0}_{r3}	$add^{r1,r4}_{r1}$	*STALL*	lwz^{r0}_{r4}	lwz^{r0}_{r1}
6	lwz^{r0}_{r3}	$add^{r1,r4}_{r1}$	*STALL*	*STALL*	lwz^{r0}_{r4}

13 cycles

K1c pipeline

```
int add4(int *t){
  return (t[0] + t[1] + t[2] + t[3]);
}
```

```
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^{r0}_{r1}				
2	lwz^{r0}_{r4}	lwz^{r0}_{r1}			
*/ 3	$add^{r1,r4}_{r1}$	lwz^{r0}_{r4}	lwz^{r0}_{r1}		
4	lwz^{r0}_{r3}	$add^{r1,r4}_{r1}$	lwz^{r0}_{r4}	lwz^{r0}_{r1}	
5	lwz^{r0}_{r3}	$add^{r1,r4}_{r1}$	*STALL*	lwz^{r0}_{r4}	lwz^{r0}_{r1}
6	lwz^{r0}_{r3}	$add^{r1,r4}_{r1}$	*STALL*	*STALL*	lwz^{r0}_{r4}
*/ 7	$add^{r1,r3}_{r0}$	lwz^{r0}_{r3}	$add^{r1,r4}_{r1}$	*STALL*	*STALL*

13 cycles

K1c pipeline

```
int add4(int *t){
  return (t[0] + t[1] + t[2] + t[3]);
}
```

```
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

```
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
    ;;
```

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..

```
add4:                          # Sched time
    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

```
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 = $r1, $r0
    ;;
    ret
    ;;
```

Register allocation and scheduling

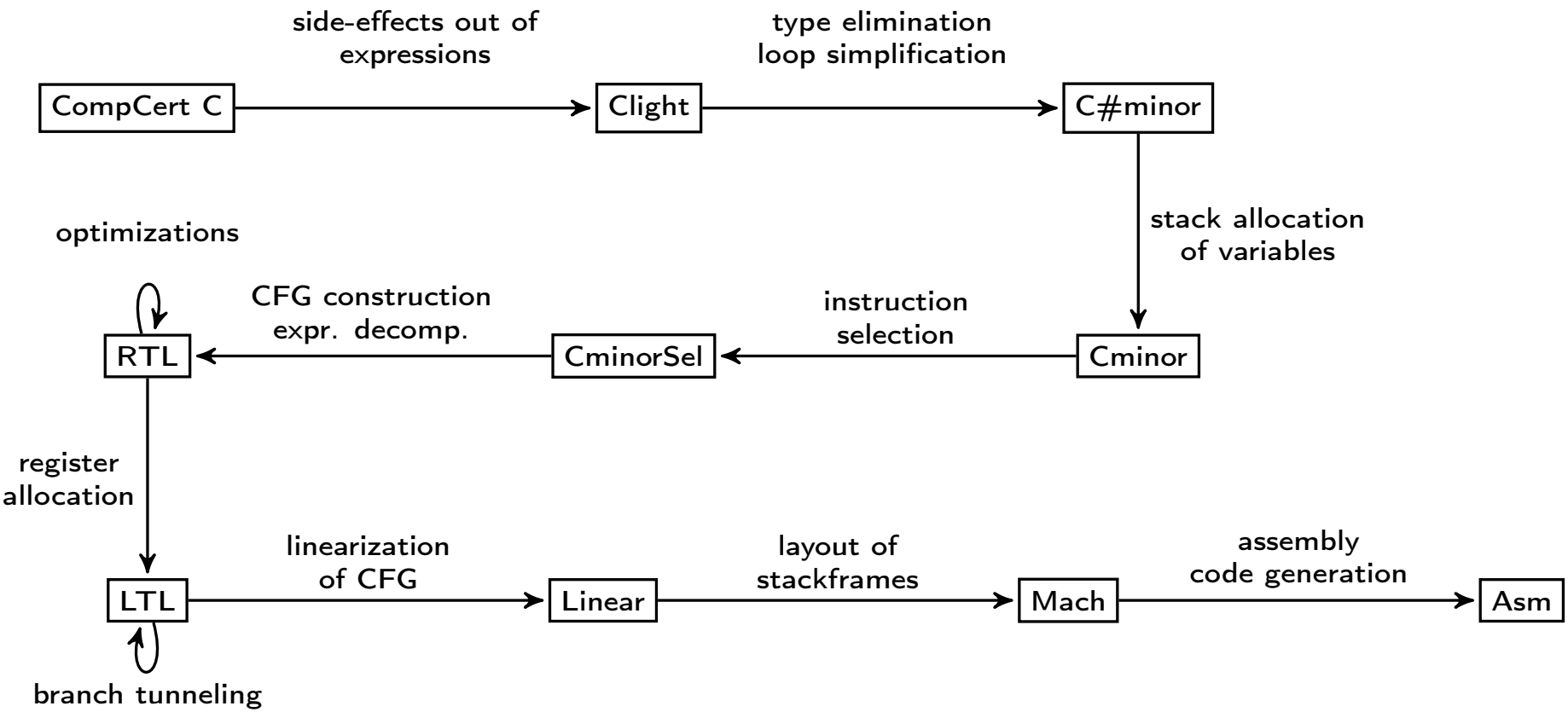
- Register allocation
  - Allocates physical registers (bounded) to virtual registers (unbounded)
  - Performs "spilling" if not able to
- Instruction scheduling
  - After register allocation (postpass): more precise informations, can make bundles, but extra dependencies on registers
  - To deal with the register dependencies: instruction scheduling before register allocation (prepass)
- So far, we did a postpass scheduling optimization

Outline

CompCert in a few words

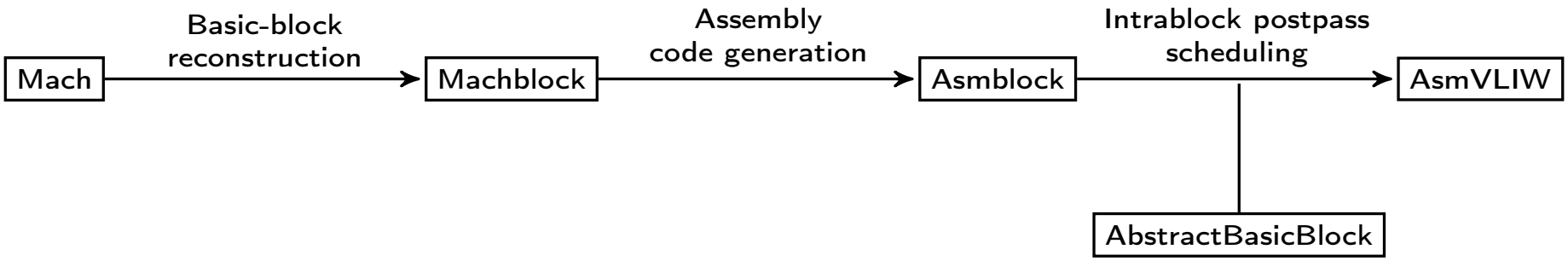
- Machine checked, formally verified compiler: proof of specification preservation
- Written in Coq and OCaml
- Targets: PPC, ARM, RISC-V, x86
- Performance: close to GCC -O1

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

Semantics of Asm

State: (rs, m)

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

- exec_instr: function -> instruction -> regset -> mem -> outcome
- outcome: either Stuck, or Next rs' m'
- Instructions reside in memory, and are pointed by `PC` register
- Exemples of execution:
  - (exec_instr f (Pcall s) rs m) returns $(rs[\texttt{RA} \leftarrow rs[\texttt{PC}];\ \texttt{PC} \leftarrow @s], m)$
  - (exec_instr f (Paddd r0 r1 r2) rs m) returns $(rs[r_0 \leftarrow rs[r_1] + rs[r_2]], m)$

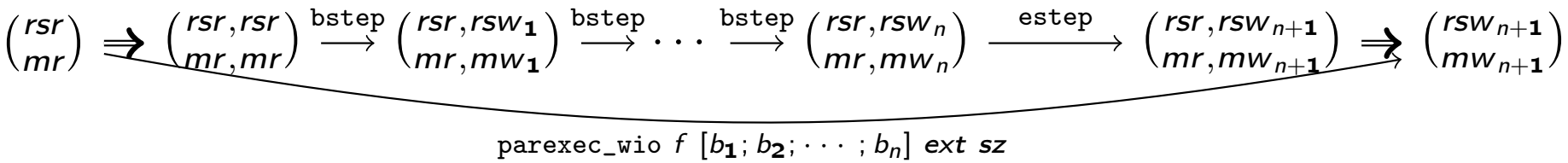
Formal definition of a basic block

```
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*, Paddd is a *basic*.
- Executing the block ($R_0 := R_1; R_1 := R_0;$ `jump @toto`) should lead to $rs[R_0 \leftarrow rs[R_1]; R_1 \leftarrow rs[R_0]; PC \leftarrow @toto]$
- 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)



- 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: $(R_0 := R_1; R_1 := R_0;$ `jump @toto`$)$
  - $rsw_1 = rsr[R_0 \leftarrow rsr[R_1]] = rs[R_0 \leftarrow r_1]$
  - $rsw_2 = rsw_1[R_1 \leftarrow rsr[R_0]] = rs[R_0 \leftarrow r_1; R_1 \leftarrow r_0]$
  - $rs' = rsw_3 = rsw_2[\texttt{PC} \leftarrow \texttt{@}toto] = rs[R_0 \leftarrow r_1; R_1 \leftarrow r_0; \texttt{PC} \leftarrow \texttt{@}toto]$

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'.
```

Non deterministic parallel semantics

- (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

Deterministic out-of-order parallel semantics (2)

- (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_bblock f b rs m o -> o = Next rs' m'.
```

- Remark: in this semantic, Stuck executions cannot happen

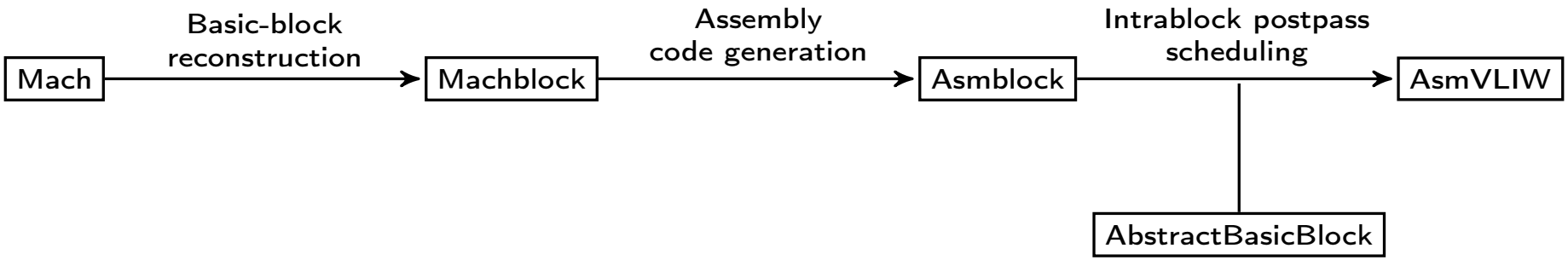
Asmblock sequential semantics

- 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

```
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..



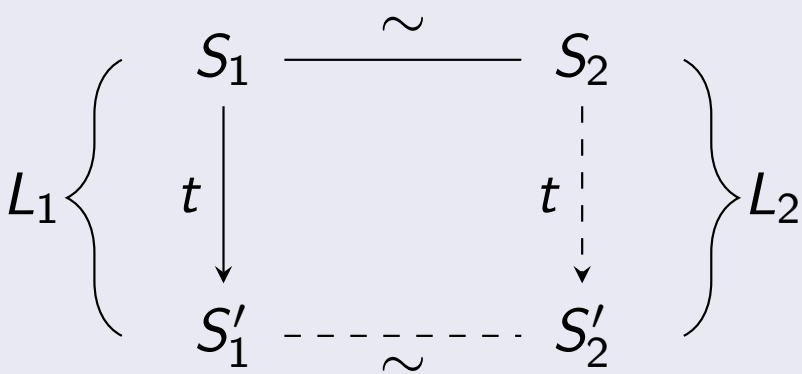
- How to reorder a block from Asmblock into several bundles of AsmVLIW, and prove it correct?

Outline

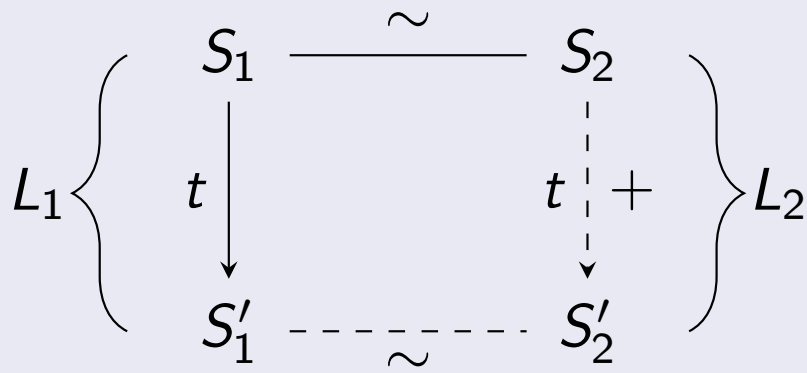
Forward simulations in CompCert

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

Lock-step simulation



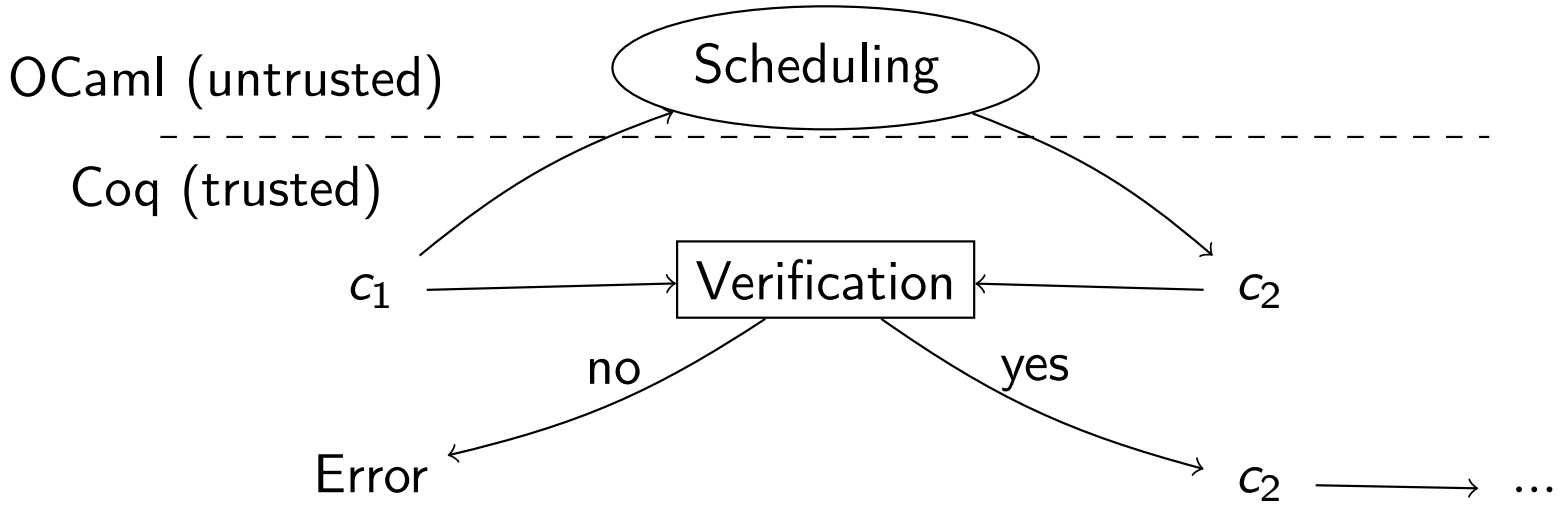
"Plus" simulation



- Lock: If $S_1 \xrightarrow[step_{L_1}]{t} S'_1$ and $S_1 \sim S_2$, then $\exists S'_2, S_2 \xrightarrow[step_{L_2}]{t} S'_2$ and $S'_1 \sim S'_2$.

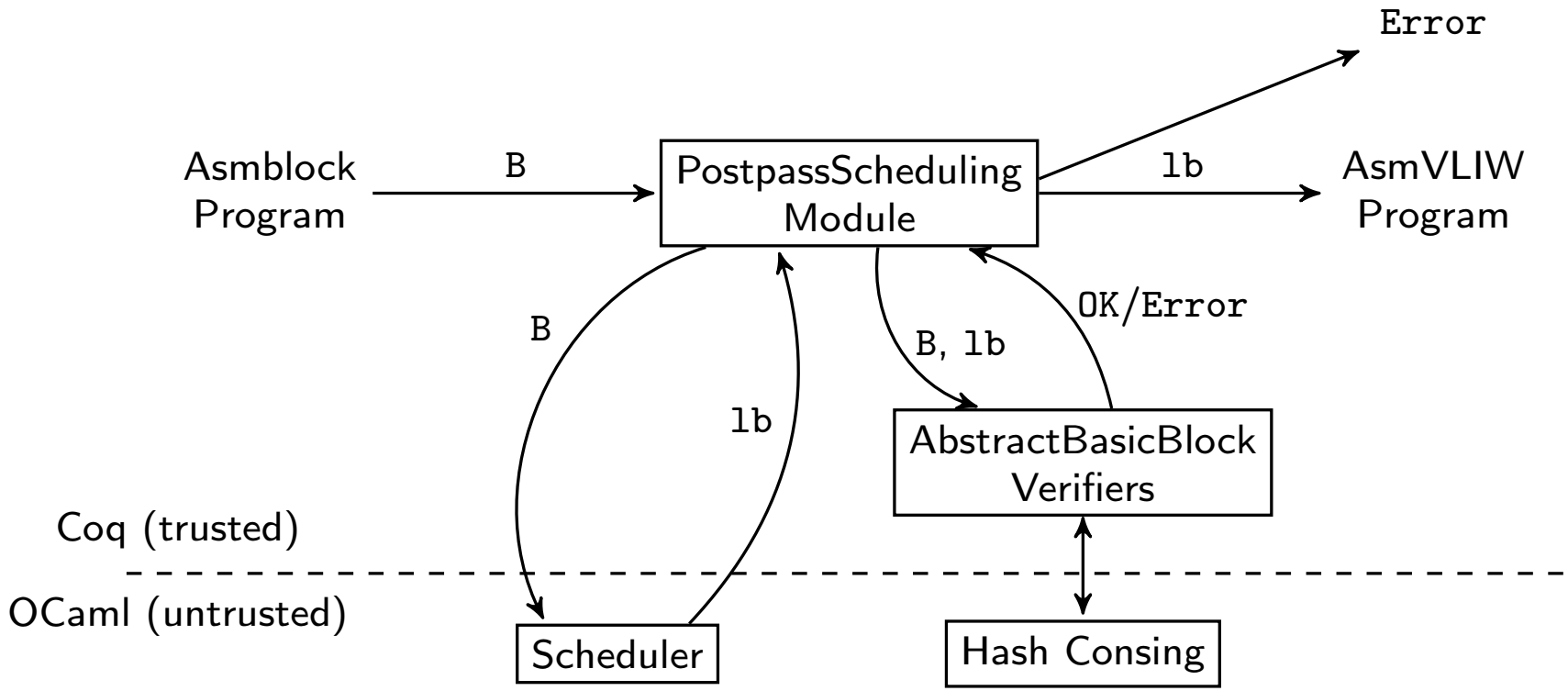
Previous work of J.B. Tristan on scheduling

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



- Verification proof: $V(c_1, c_2) = \mathit{true} \implies c_1 \sim c_2$
- Advantages: easier to prove and modular
- Implemented it at the Mach level, with symbolic evaluation
- Drawback (at the time): scalability issue

Architecture of the pass Asmblock -> AsmVLIW



- 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*
- (Drawing on board)

AbstractBasicBlock (1)

- The AbstractBasicBlock language is parametrized by:
  - Pseudo registers, operators, values, an environment genv
  - op_eval: genv -> op -> list value -> option value
- A state of AbstractBasicBlock is a mapping from pseudo registers to values
- Expressions are defined as combination of pseudo registers and operators

```
Inductive exp :=
  | Read (x:R.t) | Op (o:op) (le: list_exp) | Old (e: exp)
with list_exp :=
  | Enil | Econs (e:exp) (le:list_exp) | LOld (le: list_exp).

Definition inst := list (R.t * exp). (* list of assignments *)
```

- The assignments can be executed sequentially or in parallel
- Examples of traductions:
  - Picall r => $[\#RA \leftarrow Read(\#PC);\ \texttt{PC} \leftarrow Read(\#r)]$
  - Paddw r0 r1 r2 => $[\#r_0 \leftarrow (OpAddd[\#r_1;\ \#r_2])]$
- Defines a certified function is_parallelizable : list inst -> bool

AbstractBasicBlock (2)

- Translation from AsmVLIW to AbstractBasicBlock:
  trans_block: bblock -> (list inst)
- We prove a bisimulation for sequential, and a bisimulation for parallel semantics

Bisimulation for sequential:

```
match_states (State rs m) s ->
match_outcome (exec_bblock ge fn b rs m)
              (exec Ge (trans_block b) s).
```

Bisimulation for parallel:

```
  match_states (State rs1 m1) s1' ->
  parexec_bblock ge fn b rs1 m1 o2 ->
exists o2', prun Ge (trans_block b) s1' o2'
         /\ 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:

```
bblock_para_check bundle = true ->
exec_bblock ge f bundle rs m = Next rs' m' ->
det_parexec ge f bundle rs m rs' m'.
```

Towards proving reordering

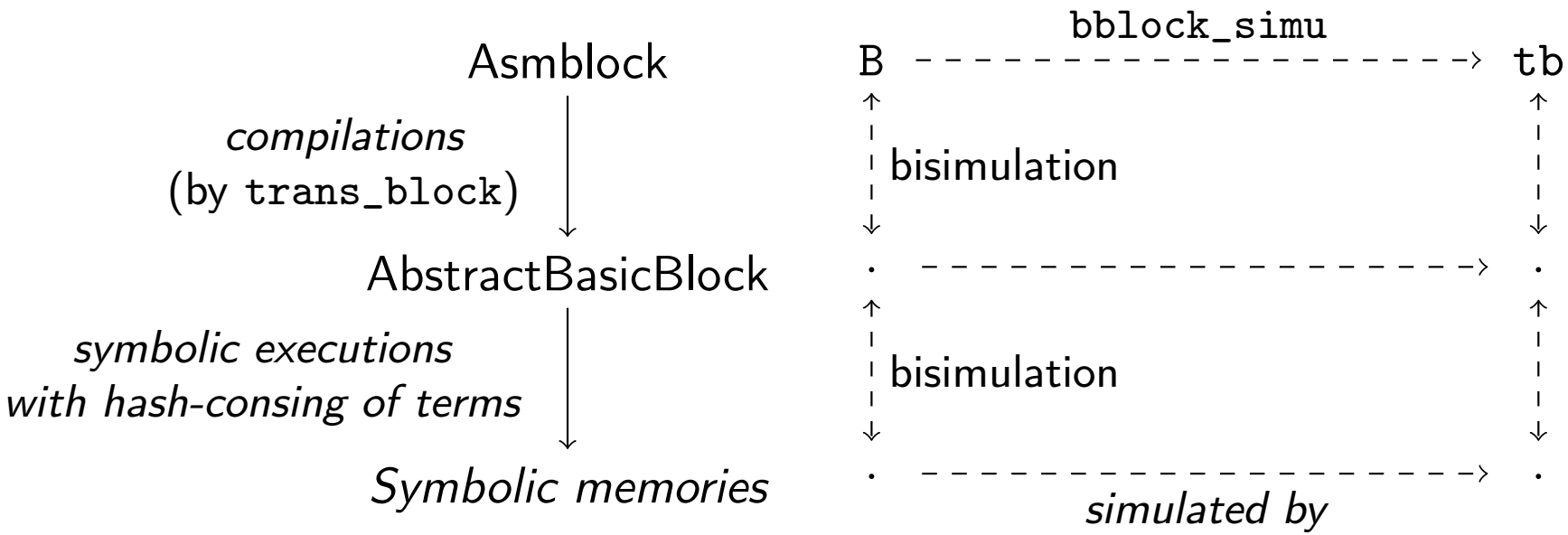
- Definition of bblock simulation:

```
Definition bblock_simu ge f b b' :=
  forall rs m, exec_bblock ge f b rs m <> Stuck ->
  exec_bblock ge f b rs m = exec_bblock ge f b' rs m.
```

- Definition of a concatenation function, and a predicate (is_concat b lb), where lb is a list bblock
- Definition of a function (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:

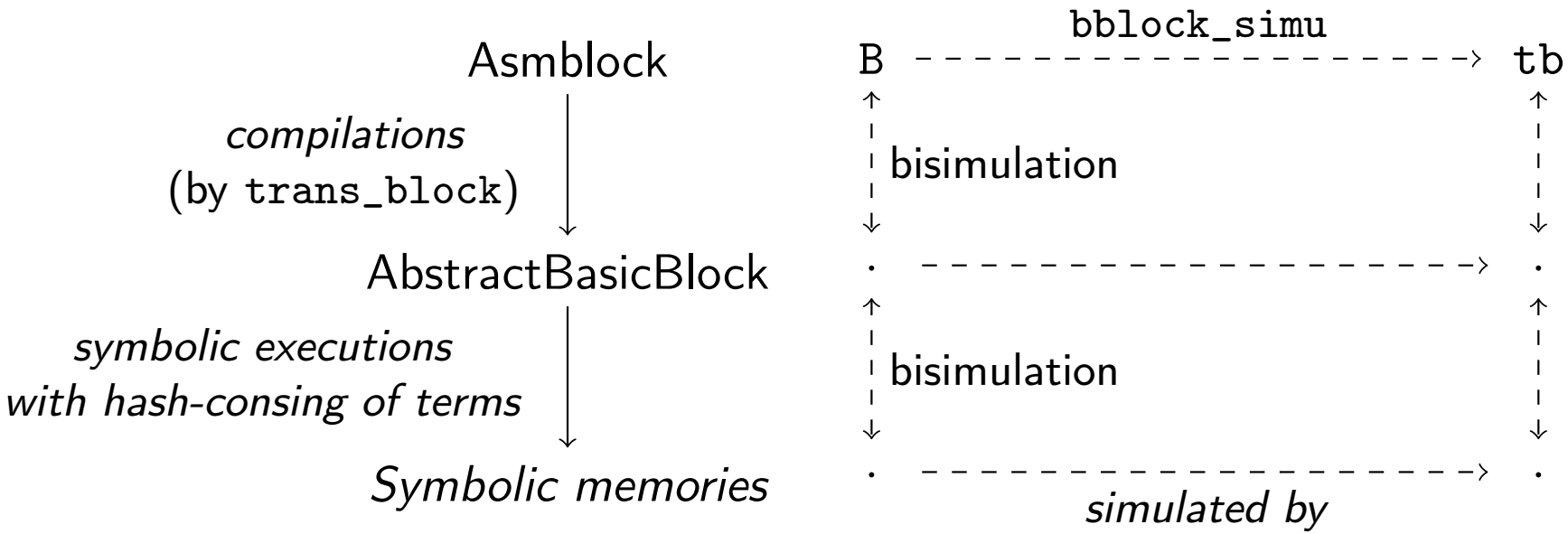
```
Theorem verified_schedule_correct: forall ge f B lb,
  (verified_schedule B) = (OK lb) ->
  exists tb, is_concat tb lb /\ bblock_simu ge f B tb.
```

Intrablock Reordering verifier of AbstractBasicBlock



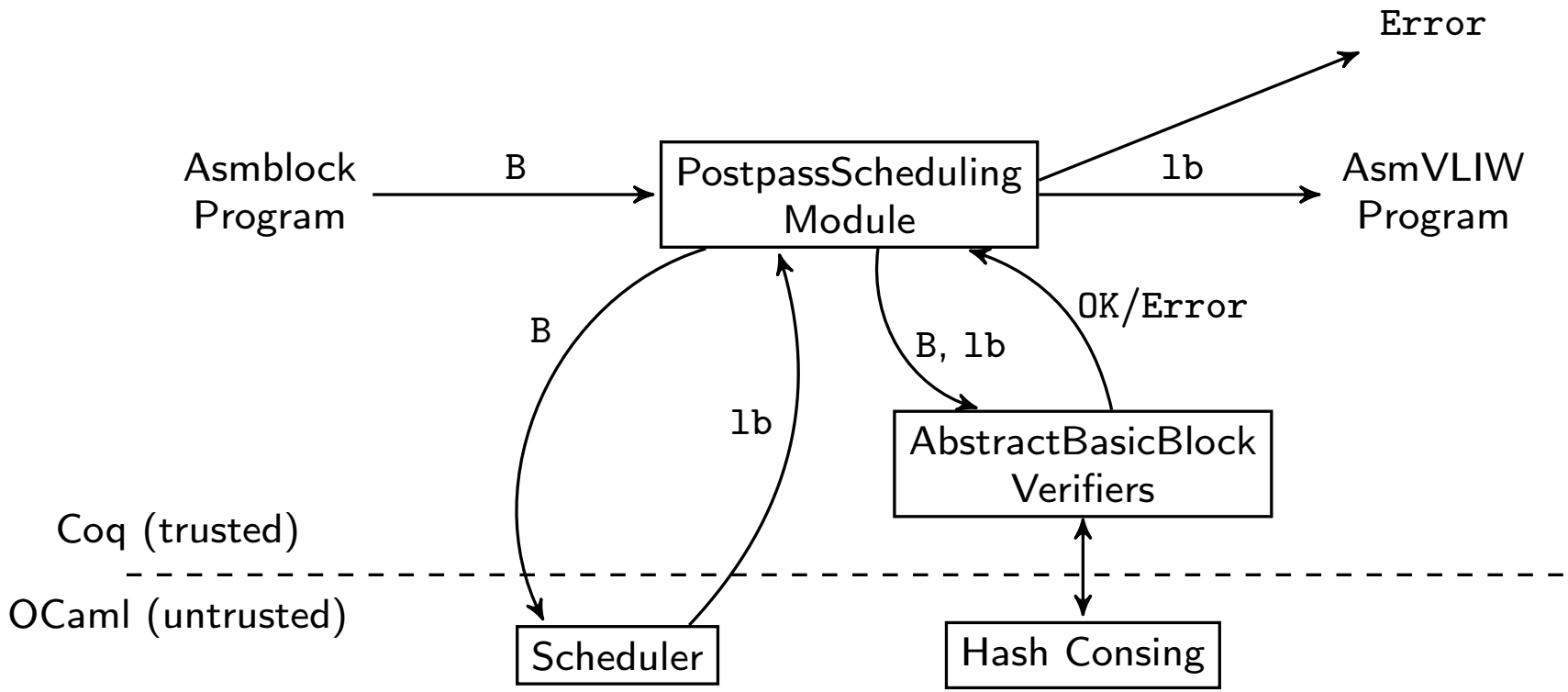
- Symbolic execution: computing symbolic memories, final value of the pseudo register in function of the initial values.
- Example:
  - $B_1 = [r_1 := r_1 + r_2;\ r_3 := load[r_2, m];\ r_1 := r_1 + r_3]$
  - $B_2 = [r_3 := load[r_2, m];\ r_1 := r_1 + r_2;\ r_1 := r_1 + r_3]$
- These two blocks are equivalent to this assignment:
  $[r_1 \leftarrow (r_1 + r_2) + load[r_2, m] \parallel r_3 \leftarrow load[r_2, m]]$

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

Verified hash consing



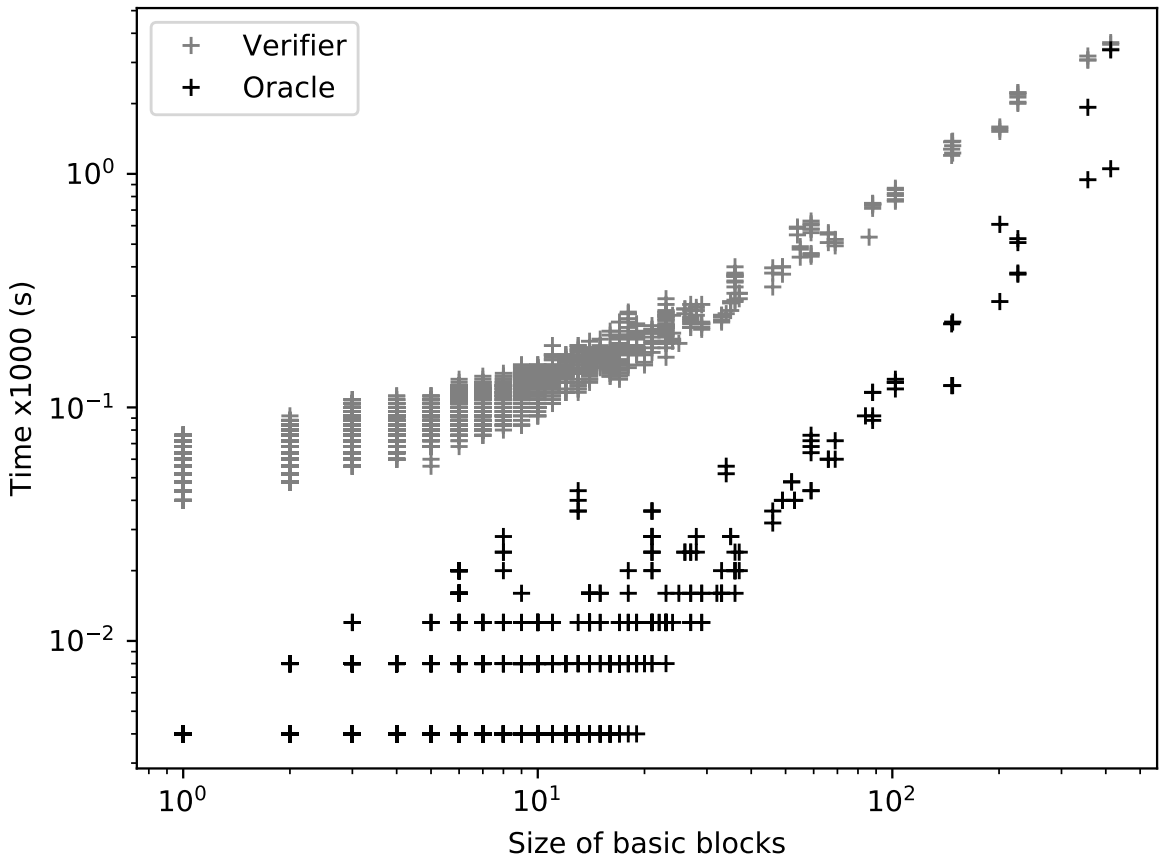
- Memoization involves calling an untrusted OCaml oracle to give memoized terms out of terms
- Dynamic checker in Coq that ensures the memoized term and the term have the same evaluation function
- Check done with OCaml pointer equality
  - Axiom: if pointer equality returns true, then the two values 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

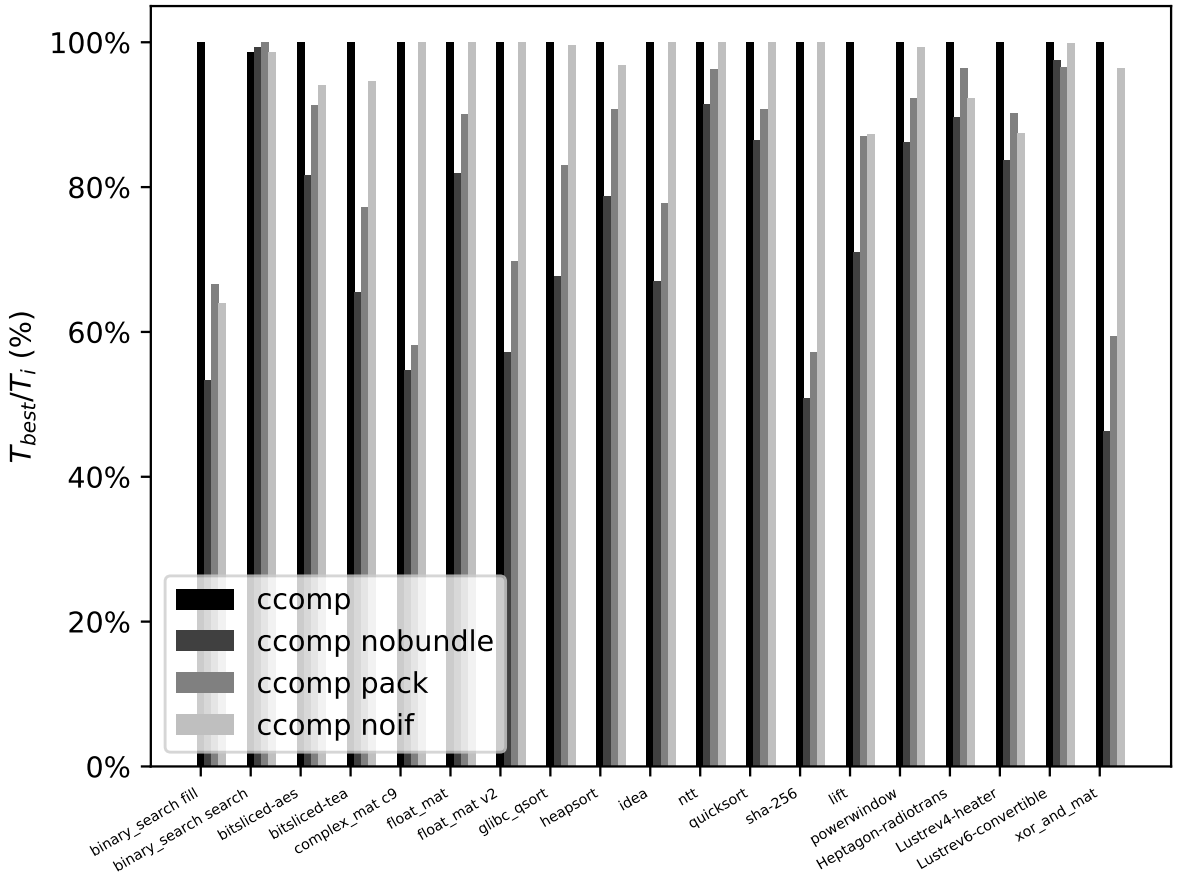
Outline

Experimental time of the oracle and verifier



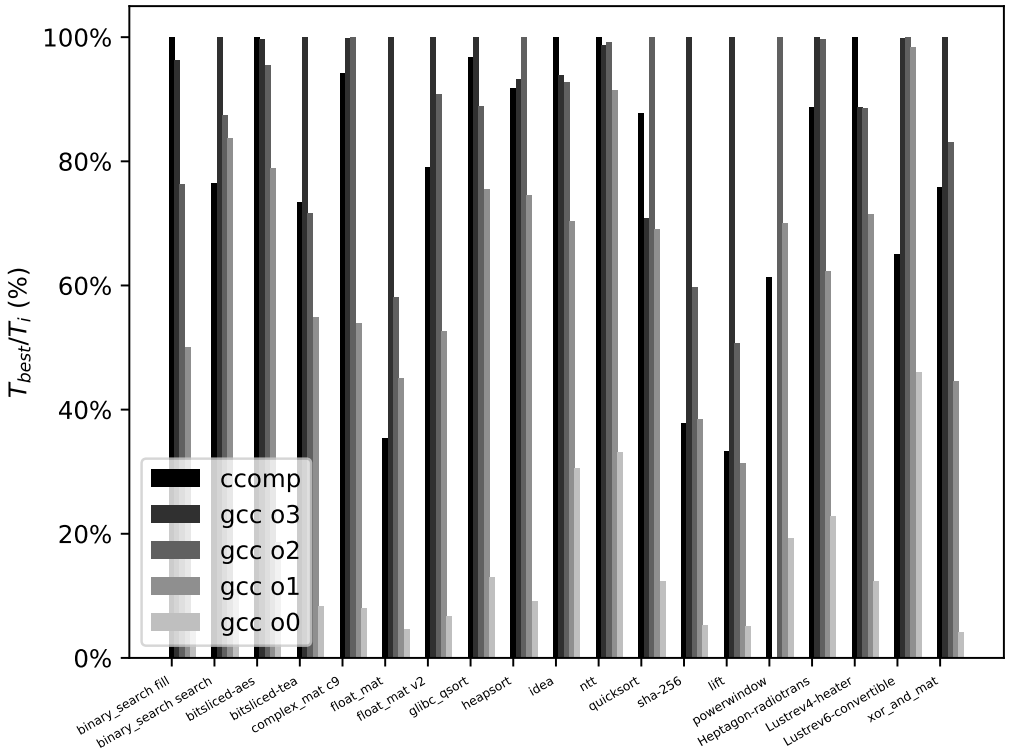
- Timings obtained by instrumenting the OCaml code

Impact of the optimizations on our backend



- Impact of scheduling: performance gain between 20% and 100%

Comparison with GCC



- 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

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?