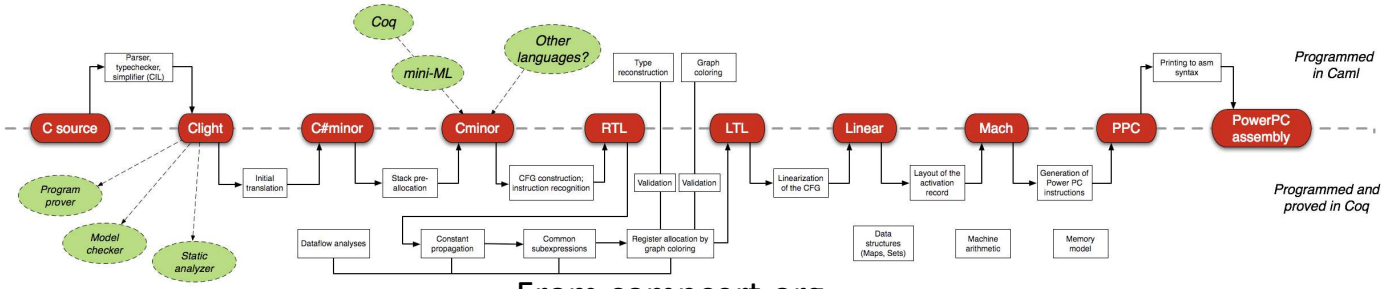
Fully Verified Instruction Scheduling

Ziteng Yang, Jun Shirako, and **Vivek Sarkar**

Compiler Correctness & Verification

- **Formal Verification:** seeking 100% correctness guarantee

- **CompCert**' s approach: directly prove correctness in an interactive theorem prover (Coq)
  - Write the compiler as a Coq function[1]
  - Formalize semantics of C, IR and assembly language in Coq
  - Prove semantics preservation of each translation in Coq



From compcert.org

Other research projects: VeLLVM, CakeML, CertiCoq

[1]In the actual engineering, the Coq function was finally extracted into OCaml function to generate an executable file

Current state of CompCert

- **Only verified several basic optimizations (O1-level optimization)**
  - e.g. Constant propagation, common subexpression elimination, redundancy elimination

- **Only support in-order translation**
  - It cannot reorder instructions a.k.a *instruction scheduling* (O2- and O3-level)

Motivation for Compiler-level Instruction Scheduling

- Improve instruction-level parallelism and reduce pipeline stall for ***in-order processors***. [e.g. Cortex-A53, U74MC]

original order

i_1 reads r_1
i_2 writes r_2, r_3
i_3 writes r_1, r_2
i_4 writes mem
i_5 reads r_3, writes mem

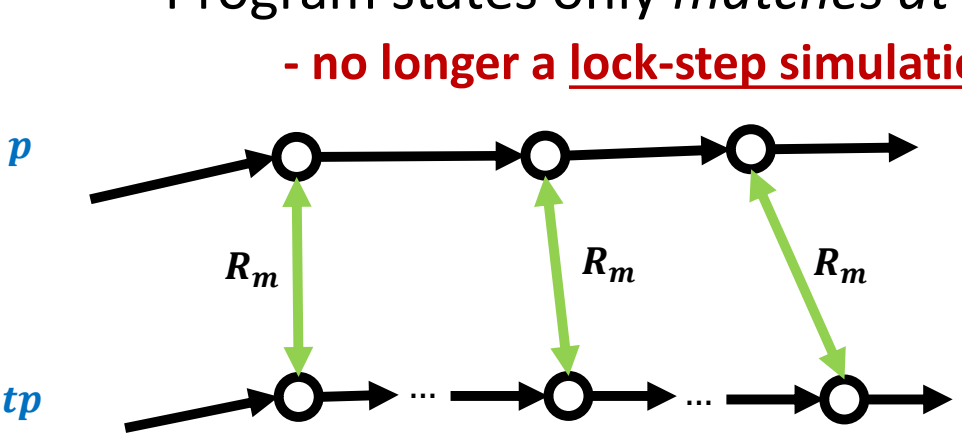
Cycle	ADD	SUB	FLOAT
1	i_1		i_1
2			i_1
3	i_2		i_2
4			i_2
5	i_3		i_3
6	i_4	i_4	
7	i_5	i_5	i_5

Cycle	ADD	SUB	FLOAT
1	i_1		i_1
2	i_4	i_4	i_1
3	i_2		i_2
4			i_2
5	i_3		i_3
6	i_5	i_5	i_5
7			

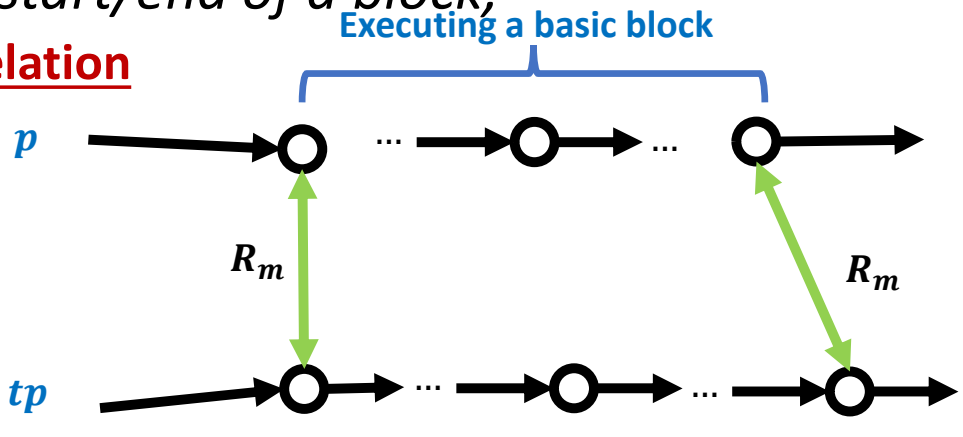
Challenges in Verified Instruction Scheduling (intra-block)

- Semantics details of a reordering instructions
- Program states only *matches at the start/end of a block,*

- no longer a lock-step simulation relation



State transition diagram (regular pass)



State transition diagram (scheduling pass)

  - Handling dependence relations between each instructions
  - Potentially heavy proof workload
- **Previous work:** verified translation validation
  - [POPL'08] Tristan, Jean-Baptiste, and Xavier Leroy. "Formal verification of translation validators: a case study on instruction scheduling optimizations."
  - [OOPSLA'20] Six, Cyril, Sylvain Boulmé, and David Monniaux. "Certified and efficient instruction scheduling: application to interlocked VLIW processors."

Full Verification v.s. Verified Translation Validation

	Full Verification	Verified Translation validation
Algorithm Correctness	Yes	No
Development Difficulty	Harder (Potentially)	Easier
Compile Time Overhead	Lower (develop-time proof)	Higher (compile-time validation)
Flexibility	Lower (May need to change proof when changing algorithm)	Higher (Only validate output with input)
Methods	Proof Assistant + Principle of Algorithm Correctness	Proof Assistant + Symbolic Execution

Full verification v.s. Verified Translation Validation

Previous work on instruction scheduling [Tristan et al. 2008] [Six et al. 2020]

Final theorem of a verified translation validation:

$$\forall\ p\ tp, if\ compile\_pass(p) = tp\ /\backslash\ validate(p, tp) = True,$$
$$then\ semantics\_preserve(p, tp)$$

Final theorem of a fully verified compilation:

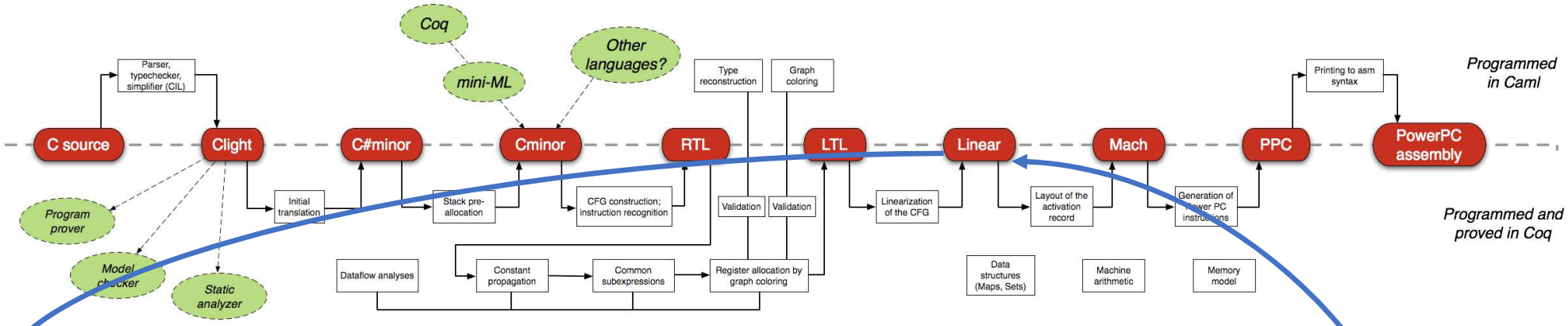
(harder to prove, but stronger result)

$$\forall\ p\ tp, if\ compile\_pass(p) = tp,$$
$$then\ semantics\_preserve(p, tp)$$

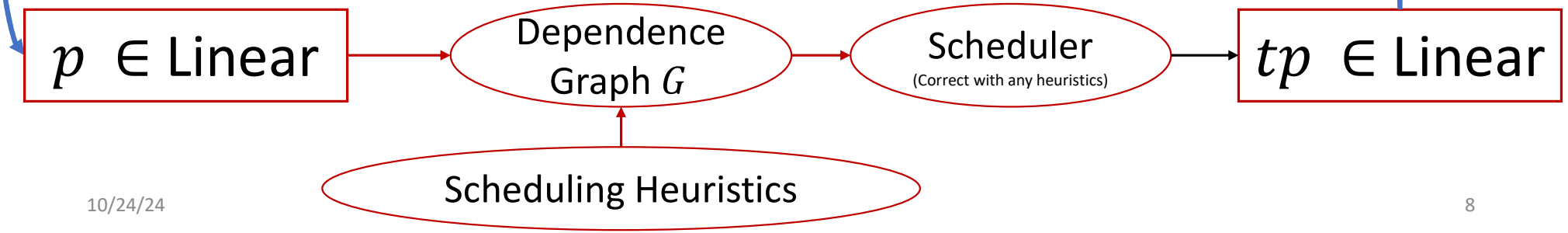
This project: make this part verified, while keep the proof work lightweight

a.k.a. correct-by-construction

This project



Instruction Scheduling



Proof Logical Chain

Part I: swapping lemma : a property of topological order

Part II: syntax-level valid instruction scheduler

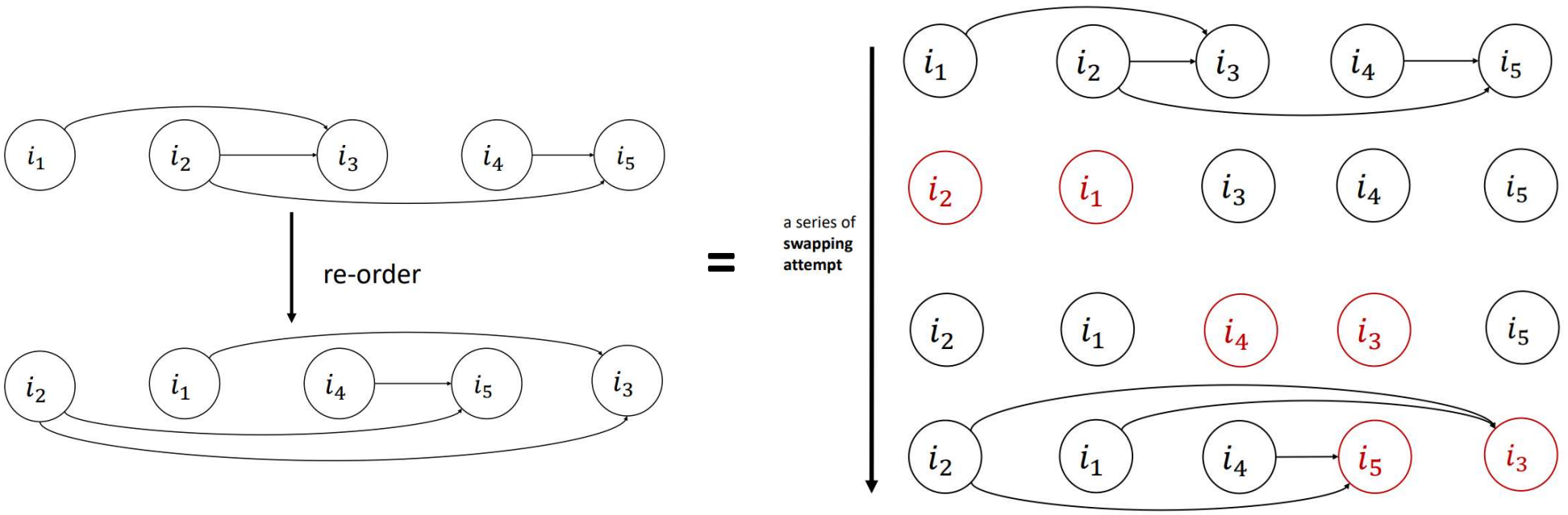
Part III: decomposing a valid scheduler

Part IV: transitivity of semantics preservation

Part V: correctness of swapping (semantics level)

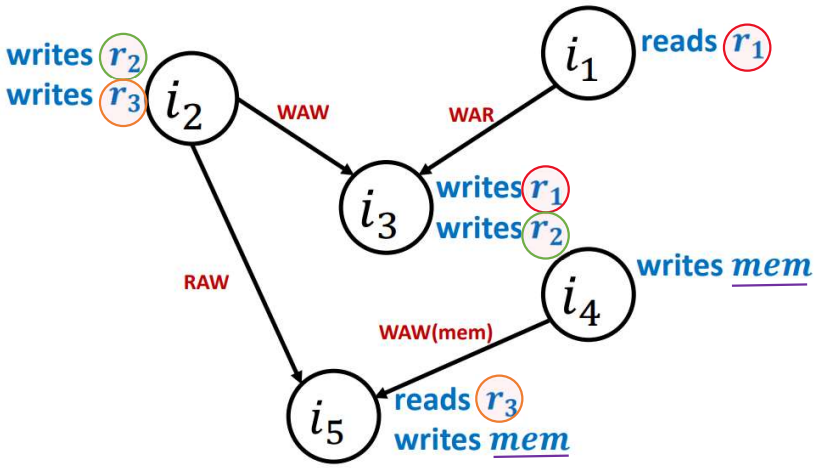
Part I: swapping lemma- a property of topological order

Swapping lemma: A topological reordering of a list of partially ordered elements is equivalent to a finite sequence of **swaps** of **adjacent but not ordered** elements.



Part II: syntax-level valid instruction scheduler (intra-block)

The dependence constraints of the original program: a valid instruction scheduler conduct a topological reordering based on the **dependence relation (defined in syntax level, by matching the register name)**.



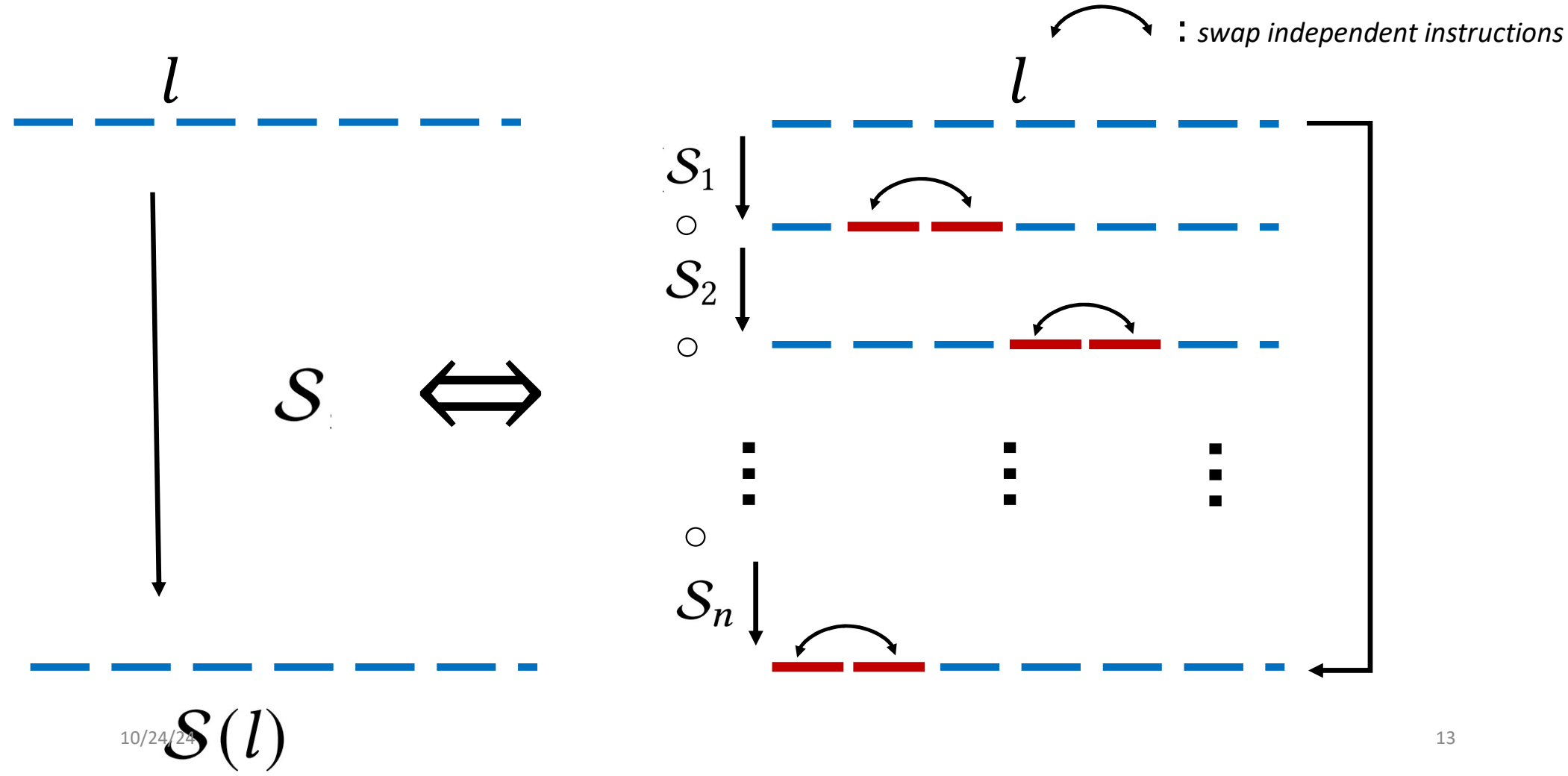
Part III: decompose a valid scheduler

Any syntax-level valid scheduler,
reorders a program's instructions

=

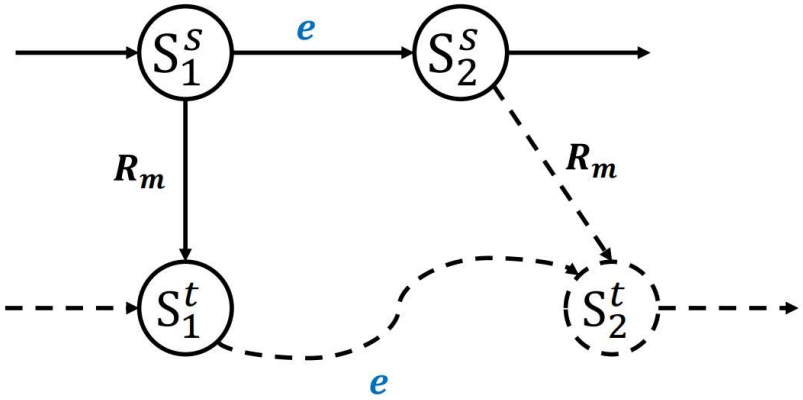
Composition of a finite sequence of compiler passes,
that only swap one pair of independent instructions
(named *single swappers*)

Decomposing a scheduler



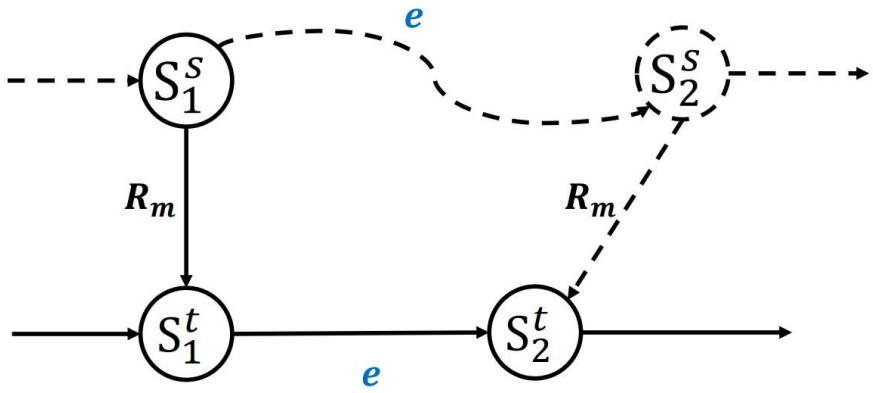
Part IV: transitivity of semantics preservation

- **The final goal of CompCert proof:** *backward simulation* of state transition between C and compiled Asm program's small-step semantics, through **only proving *forward simulation* of each pass** and lemmas that "flips" the simulation direction[1]



Forward simulation (sufficient to prove this only for each single compiler pass)

implies

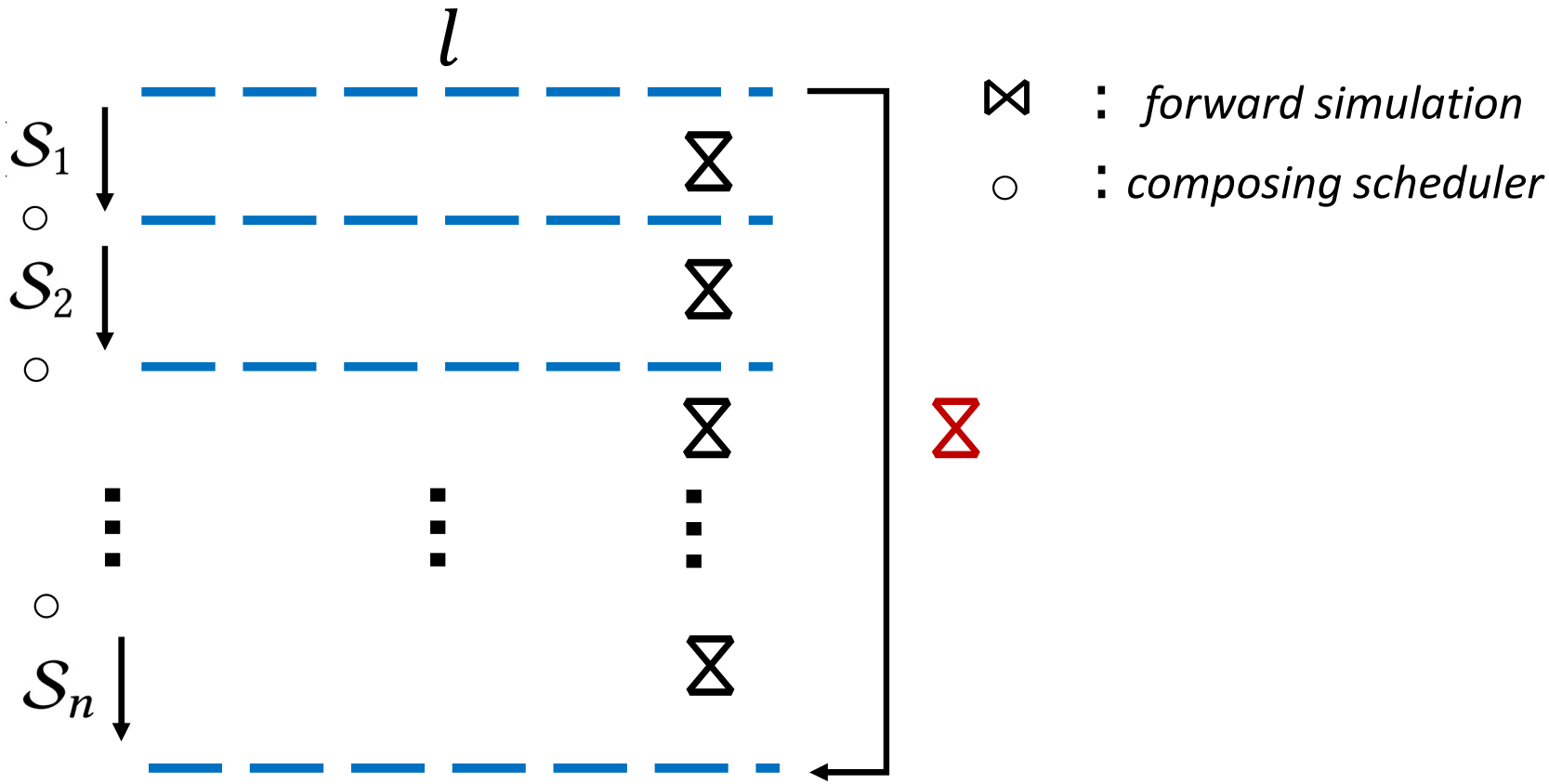


Backward simulation (final goal of whole compiler, derived by forward simulation and determinism of assembly language)

[1]One of the base theory of CompCert

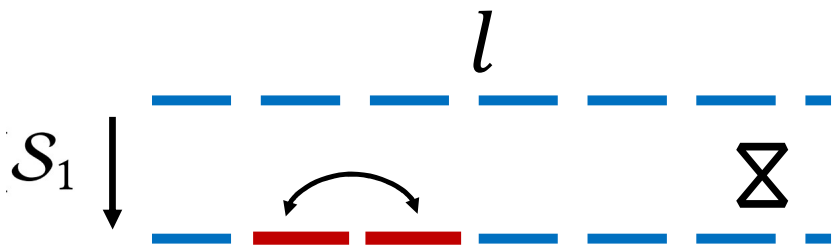
Lemma: *forward simulation is transitive*

- one of the base theory of CompCert



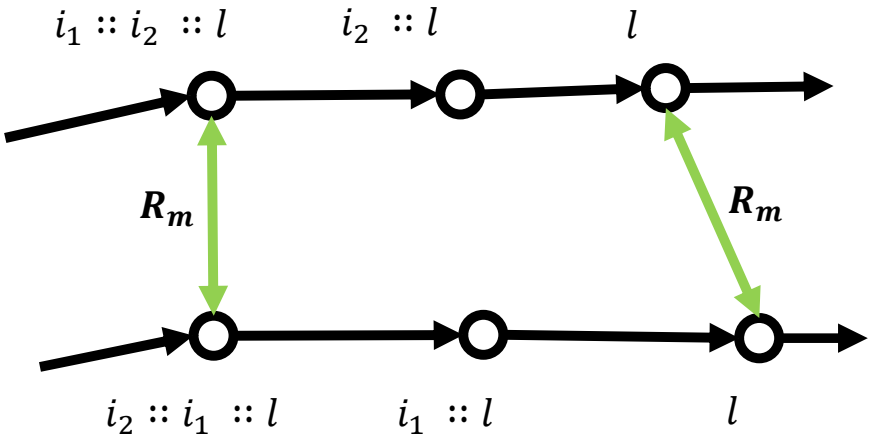
Part V: correctness of swapping

The only lemma that requires reasoning on semantics details: swapping only one pair of adjacent **syntax-level** independent instructions (RAW/WAR/WAW dependence derived by pattern-match) inside only one basic block of a program satisfied the forward simulation, a.k.a. **semantics-level equivalence** of the program

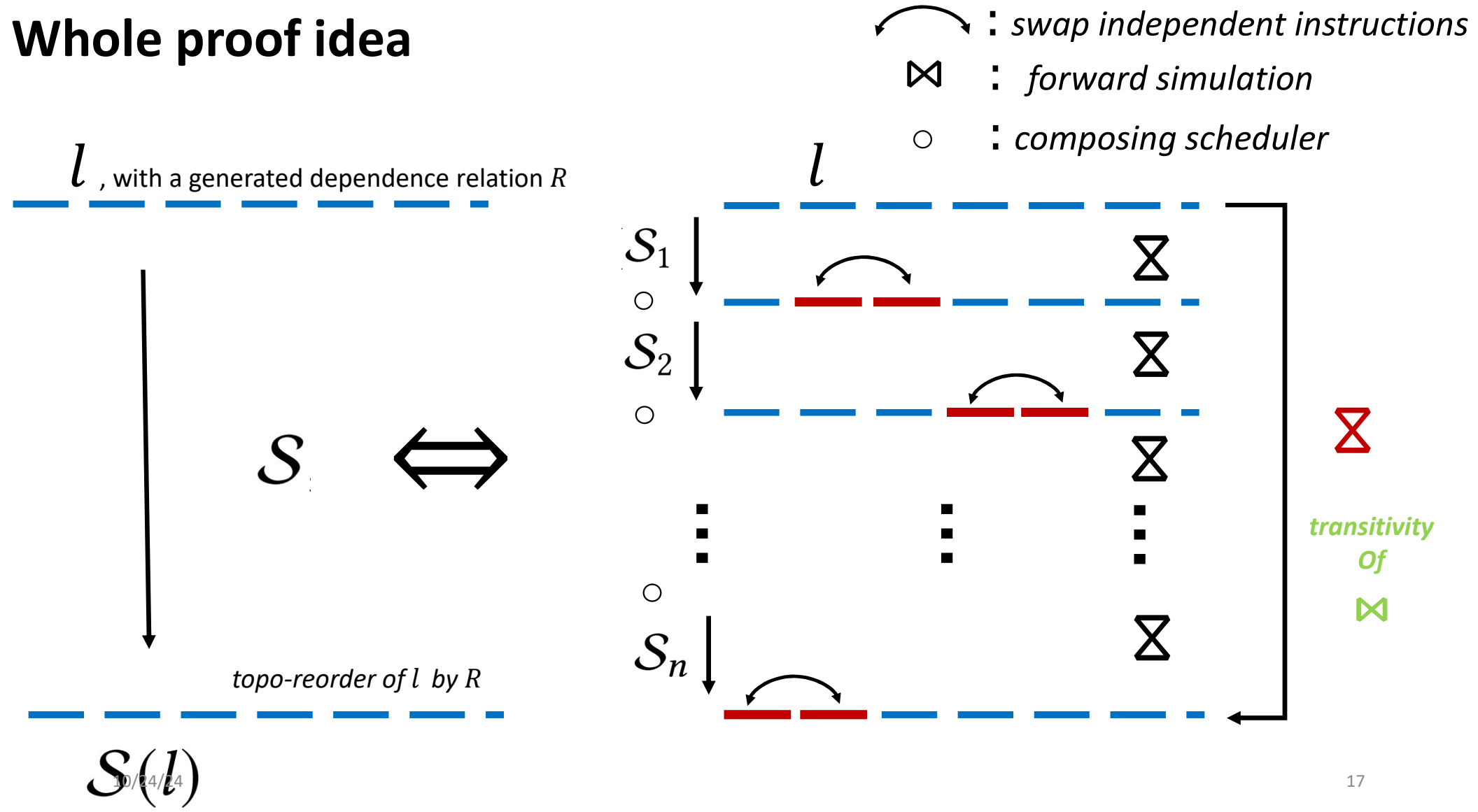


: *swap independent instructions*

$\bowtie$: *forward simulation*



Whole proof idea



What did we get till here?

A general framework to prove any instruction scheduling algorithm

- In other words, the theory above is once-for-all

The framework was formalized in Coq (based on CompCert framework)

Prove a list-scheduling using our framework

- **A concrete instruction scheduling implementation**
  - Generate the dependence graph of original basic block
  - Iteratively choose and pop an available instruction, according to an outside scheduling heuristics to the scheduled list

Algorithm 1 Dependence Graph Generating: $DRel(l)$

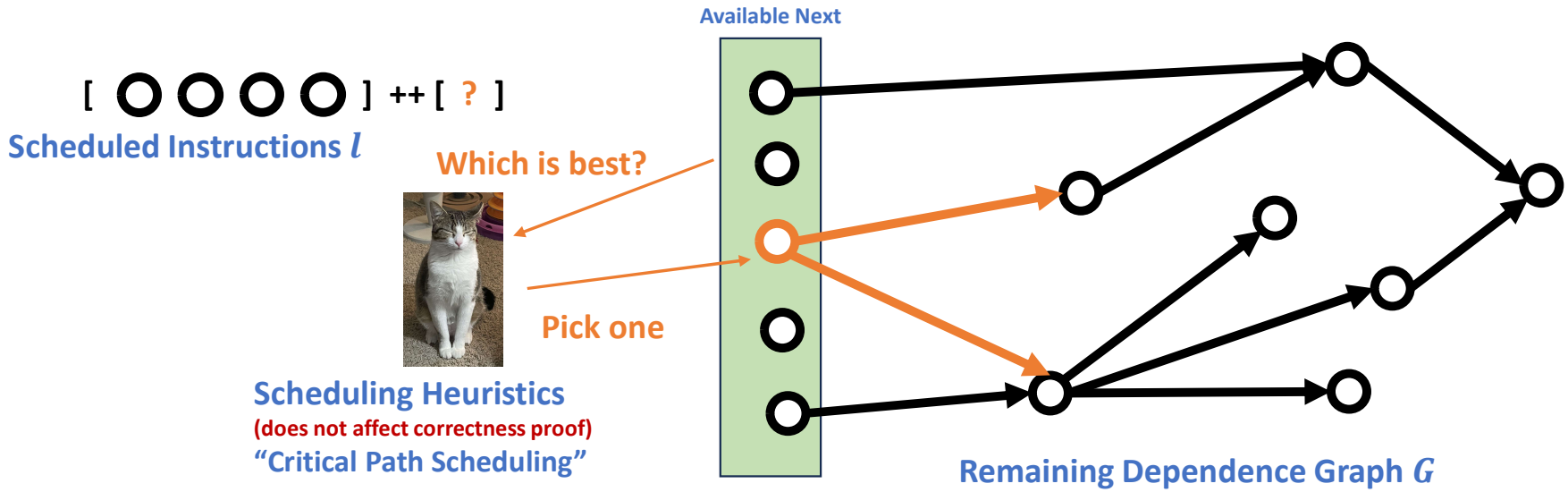
Require: List of instructions $l = [i_1, i_2, ..., i_n]$ ▷ Non-duplicate by giving index to them
Ensure: Graph G that records $\mathcal{G}_l^{\mathcal{D}}$, the generated order of l by $\mathcal{D}$ ▷ Proved in Section.5.3
  if $l = nil$ **then**
    G.$nodes \leftarrow \mathrm{E}_l$
    G.$edges \leftarrow \emptyset$
  else if $l = i' :: l'$ **then**
    G.$edges \leftarrow$ G.$edges \cup \{(i', i) | i \in l' \wedge \mathcal{D} i' i\}$
    G.$edges \leftarrow$ G.$edges \cup DRel(l').edges$
  end if

Algorithm 2 List Scheduling $\mathcal{S}^*(\mathcal{P}, l)$

Require: A heuristic function $\mathcal{P}$: *list instruction* $\rightarrow$ *list N*
Require: List instructions $l = [i_1, i_2, ..., i_n]$ ▷ Non-duplicate by giving index to them
Ensure: l^* is a topo-reorder of l by $\mathcal{G}_l^{\mathcal{D}}$ ▷ Proved in Section.5.3
  G $\leftarrow DRel(l)$
  $Priority \leftarrow \mathcal{P}(l)$ ▷ $\mathcal{P}(l)(k)$ will the priority of i_k
  $l^* \leftarrow []$
  while G not empty **do**
    A $\leftarrow \{i_k \in l \,|\forall i_{k'} \in l.(i_{k'}, i_k) \notin \mathrm{G}\}$
    $i_{k^*} \leftarrow i_{k^*} \in$ A such that $Priority[k^*]$ *is max*
    $l^* \leftarrow l^* + +[i_{k^*}]$
    G $\leftarrow$ remove node i_{k^*} from G
  end while
    return l^*

Prove a list-scheduling using our framework

- **A concrete instruction scheduling implementation**
  - Generate the dependence graph of original basic block
  - Iteratively choose and pop an available instruction, according to an outside *scheduling heuristics* to the scheduled list

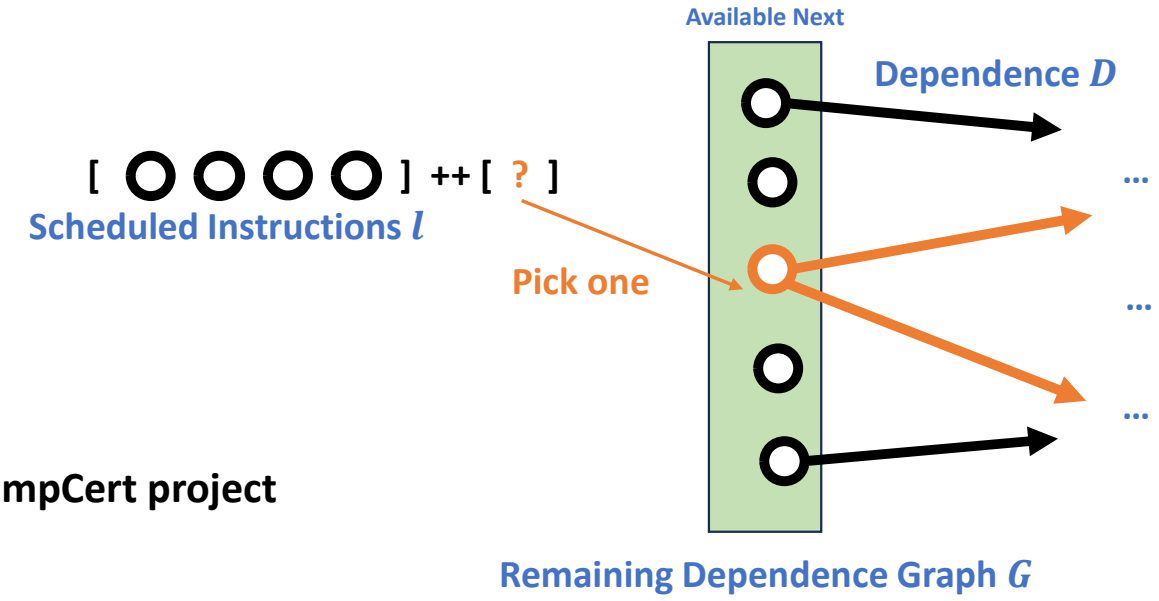


Prove a list-scheduling using our framework

Brief idea of prove the topo-logical reorder:

the scheduler maintains an invariant during scheduling

- **EXCLUSIVE:** $l \cap G.node = \emptyset$
- **SUB:** $l \cup G.node \subseteq original$
- **SORT:** l **is sorted by** D



Everything is formalized/checked in Coq,

and incorporated into CompCert project

An Evaluation on Proof Engineering

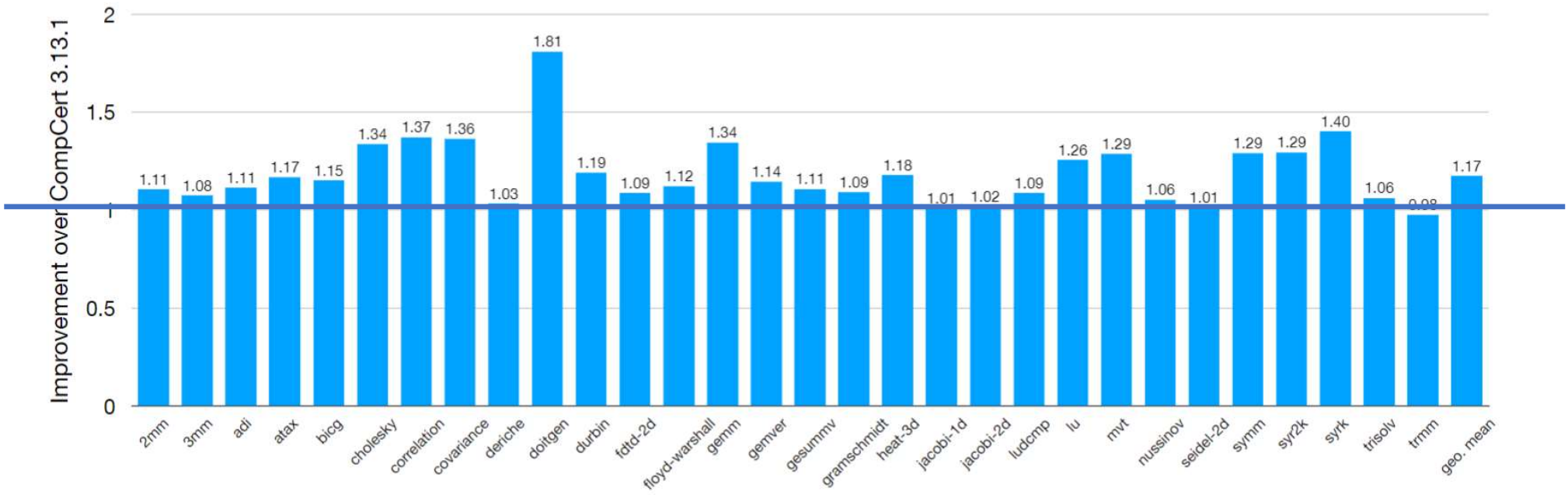
	Language	Functions	Proofs
Base theories on topo-reorder's properties (once-for-all)	Coq	-	0.8k
Base theories on semantics (once-for-all)	Coq	-	2.2k
List-scheduling algorithm (excluding heuristics)	Coq	0.15k	1.0k
Scheduling heuristics	Ocaml	25	-
Scheduling heuristics	C	0.7k	-
Machine dependent proof (Risc-V)	Coq	-	40
Machine dependent proof (x86)	Coq	-	35

LOC of program/functions and proofs in our work

	Fully Verified	Scope	Lines of Proof Code
This work	**Yes**	**List Scheduling**	4k
[Tristan et al. 2008]	**No**	**List and Trace Scheduling**	11k
[Six et al. 2020]	**NO**	**List Scheduling (VLIW)**	18k + 10k (architecture)

LOC of related work

An Evaluation on Our List-scheduler's Performance
(Improvement in execution times on Risc-V hardware platform)



Performance improvements by the certified instruction scheduler for PolyBench C 4.2

Future work

- **Verifying Inter-block Scheduler, with alias analysis of memory access**
  - Best existing method also used verified translation validation only
  - Block size change. Swapping lemma won't work

- **General: towards multi-level parallelism for verified compiler**
  - Data-level parallelism
  - Instruction-level parallelism (this work improved)
  - Task-level parallelism (e.g. loop parallelism)

  a.k.a. bringing CompCert to O2/O3-level optimization

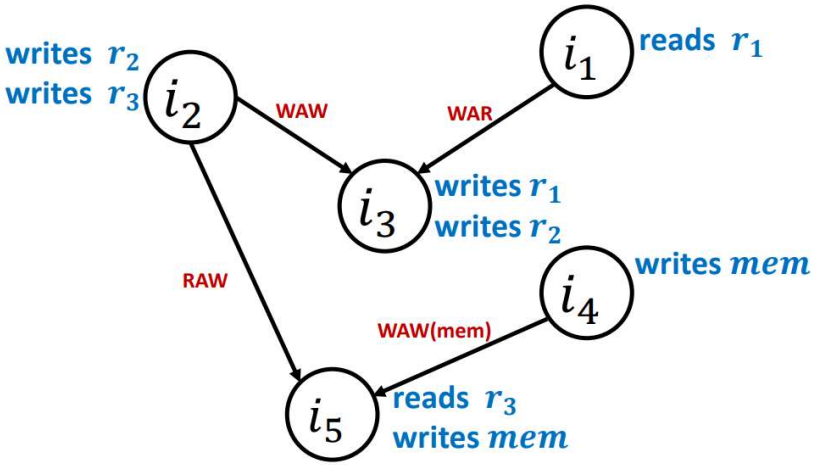
Summary: verified instruction scheduling framework with multi-level flexibility

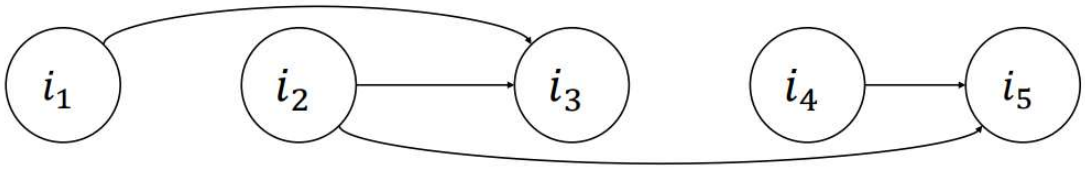
- **Flexible algorithm changes**
  - Change algorithm => only change proofs on syntax dependence preservation
  - Nothing about semantics again
- **Flexible instruction scheduling heuristics**
  - Change scheduling heuristics => no change of correctness proof
- **Flexible Machine Architecture**
  - Machine independent implementation (40 lines of Coq code diffs between x86-64/Risc-V)
  - Only implement different heuristics for different architecture

- **Q & A?**

A BLANK PAGE

Topological order: given a list l and a partial order R on its elements E_l, l is said to be an topo-sorted list by R if $\forall i_1\ i_2\ \in\ N, R\ l[i_1]l[i_2] \rightarrow i_1 < i_2$ ($l[i\,]$ means the i -th element of l)

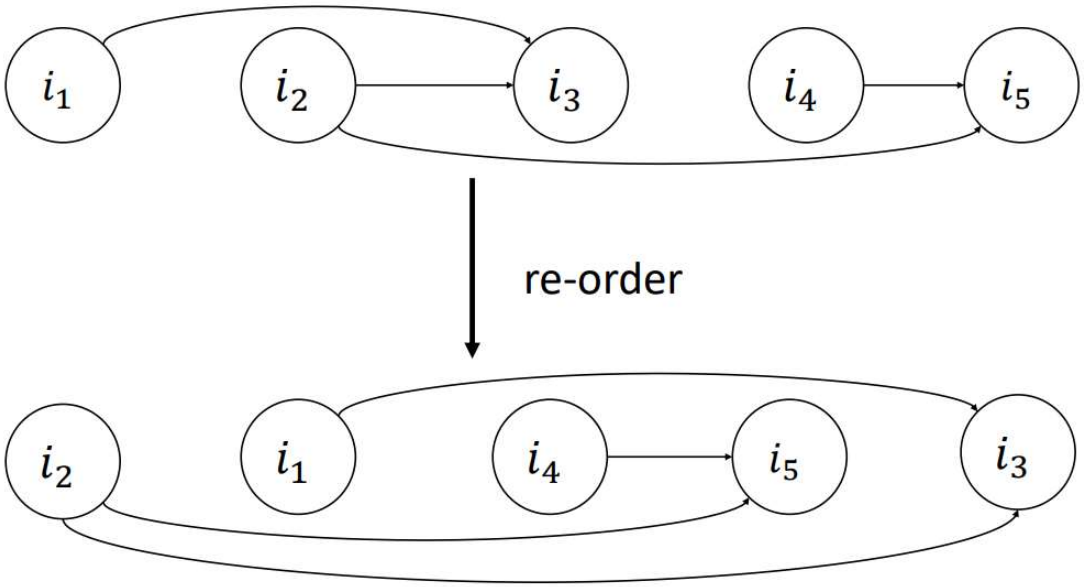




a partial order R from dependence relation

a **topo-sorted** list

Topological reorder: Given a topo-sorted list l of elements A by R, another list l' is said to be a topo-reorder of l iff l' contains exactly the same elements as l and is also topo-sorted by R.



Scheduling heuristics for Risc-V: an engineering trick

Coq-OCaml interface: a scheduling heuristics that only affect performance, not correctness, was not implemented in Coq but directly in an OCaml function

```
Require Import ExtrOcamlIntConv.
Parameter prioritizer : list int -> int -> list (list int) -> int -> (list int).
...
(* definition of encoding of instruction to an integer *)
...
Definition prioritizer' (l: list instruction): list positive :=
  let nodes := block2ids l in
  let edges := nblock2edges (numlistgen l) in
  let prior' := prioritizer nodes (int_of_nat (length nodes))
                            edges (int_of_nat (length edges)) in
```

- Since the CompCert's Coq code will eventually be extracted to OCaml, this does not change the ***trusted computing base***

Scheduling heuristics for Risc-V: an engineering trick

OCaml-C interface: the scheduling heuristics in OCaml actually uses C interface further to reduce the developing time (we have existing tools in C)

```
open Ctypes
(* The prioritizer function in OCaml *)
let prioritizer nodes n edges m: int list =
  (* First, we will need to convert them to C arrays *)
  let nodes_arr = CArray.of_list int nodes in
  let edges_arr =
    let inner = List.map (fun e -> CArray.of_list int e |> CArray.start) edges in
    let outer = CArray.of_list (ptr int) inner in outer
  in
  (* Now, we pass arguments into prioritizer *)
  let result =
    C.Functions.prioritizer (CArray.start nodes_arr) n (CArray.start edges_arr) m
  in
  CArray.from_ptr result n |> CArray.to_list
```

```
int *prioritizer(int *nodes, int n, int **edges, int m);
```

Scheduling heuristics for Risc-V : an engineering trick

Using Coq-OCaml-C interface is just an engineering choice, not a necessity of our implementation

We can still do everything in Coq, but it will increase the learning burden of both *proof engineer* and *compiler-backend engineer* without improving the correctness result.