Fully Verified Instruction Scheduling

ZITENG YANG, Georgia Institute of Technology, USA
JUN SHIRAKO, Georgia Institute of Technology, USA
VIVEK SARKAR, Georgia Institute of Technology, USA

CompCert project, the state-of-the-art compiler that achieves the first end-to-end formally verified C compiler, does not support fully verified instruction scheduling. Instead, existing research that works on such topics only implements translation validation. This means they do not have direct formal proof that the scheduling algorithm is correct, but only a posterior validation to check each compiling case. Using such a method, CompCert accepts a valid C program and compiles correctly only when the untrusted scheduler generates a correct result. However, it does not guarantee the complete correctness of the scheduler. It also causes compile-time validation overhead in the view of runtime performance.

In this work, we present the first achievement in developing a mechanized library for fully verified instruction scheduling while keeping the proof workload acceptably lightweight. The idea to reduce the proof length is to exploit a simple property that the topological reordering of a topological sorted list is equal to a sequence of swapping adjacent unordered elements. Together with the transitivity of semantic simulation relation, the only burden will become proving the semantic preservation of a transition that only swaps two adjacent independent instructions inside one block. After successfully proving this result, proving the correctness of any new instruction scheduling algorithm only requires proof that it preserved the syntax-level dependence among instructions, instead of reasoning about semantics details every time. We implemented a mechanized library of such methods in the Coq proof assistant based on CompCert's library as a framework and used the list scheduling algorithm as a case study to show the correctness can be formally proved using our theory.

We show that with our method that abstracts away the semantics details, it is flexible to implement any scheduler that reorders instructions with little extra proof burden. Our scheduler in the case study also abstracts away the outside scheduling heuristic as a universal parameter so it is flexible to modify without touching any correctness proof.

CCS Concepts: • **Theory of computation → Program verification**; **Operational semantics**; • **Software and its engineering → Formal software verification**; **Compilers**.

Additional Key Words and Phrases: Compiler Verification, Coq Proof Assistant, CompCert, Instruction-level Parallelism

ACM Reference Format:
Ziteng Yang, Jun Shirako, and Vivek Sarkar. 2024. Fully Verified Instruction Scheduling. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 299 (October 2024), 26 pages. https://doi.org/10.1145/3689739

Authors' Contact Information: Ziteng Yang, Georgia Institute of Technology, Atlanta, USA, ziteng.yang@gatech.edu; Jun Shirako, Georgia Institute of Technology, Atlanta, USA, shirako@gatech.edu; Vivek Sarkar, Georgia Institute of Technology, Atlanta, USA, vsarkar@gatech.edu.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2024 Copyright held by the owner/author(s).
ACM 2475-1421/2024/10-ART299
https://doi.org/10.1145/3689739

1 Introduction

1.1 Compiler Correctness and Formally Verified Compiler

There has been significant progress on the topic of compiler correctness in the past decades. A key goal for compiler correctness is to formally verify the code translations performed by a compiler. The CompCert certified compiler [31, 32] is the first successful project to formally verify a realistic compiler translation. In the later decade, more advanced formal theories on more optimizing passes [35], supporting linking [26, 27, 37, 46], concurrency [26, 43], improving memory models [6, 50], etc. have occurred. In another aspect of the actual performance, both the compiler testing experiment of [52] and [54] shows that CompCert's verified passes have passed all the provided test cases while GCC and Clang were found to have hundreds of test failures respectively.

We observed that most of the state-of-the-art research for certified compilation involving parallelism/concurrency are compiling functions of the multi-thread program, a.k.a developing advanced correctness theory for concurrent settings (e.g. [26, 33]). However, there's much less progress on improving the parallelism of a program: compile-time optimization for data/instruction/thread-level parallelism. In other words, CompCert does not support -O2 or -O3 optimizations like GCC or Clang currently. *Instruction scheduling* [4, 12, 17, 19, 22, 23, 51] is one of the most important methods to improve instruction-level parallelism. It can be conducted during two different stages: runtime or compile-time. Instruction scheduling during runtime is known as out-of-order (OOO) execution.

Compiler-level instruction scheduling statically reorders the instruction sequence to improve CPU throughput by removing pipeline hazards and multi-issue pipeline performance by enhancing instruction-level parallelism. Instruction scheduling is a key optimizing pass for in-order processors, which do not support out-of-order pipeline execution in hardware so as to achieve higher energy efficiency, lower hardware cost, and more predictable execution time. These characteristics are often critical for various areas, including embedded systems and mobile devices [15, 49].

1.2 Related Work on Instruction Scheduling and its Verification

Instruction scheduling is a classical compiler optimization pass that aims to minimize the schedule length of given code fragments on the target processor. The instruction scheduling problem can be classified into *local scheduling* to reorder instructions within a basic block and *global scheduling* to reorder instructions within and across basic blocks (also referred as *intra-block scheduling* and *inter-block scheduling*). The optimal local scheduling problem for modern multi-issue pipelined processors is known to be NP-complete [5]. Although the approach based on integer programming can produce the optimal schedule in a reasonable time [51], heuristic approaches typified by list scheduling [12, 23, 28] are often used in production compilers to achieve fast and efficient scheduling. The global scheduling problem also has a long history and there is an extensive body of literature including scheduling algorithms targeting VLIW processors [17, 20, 22, 30, 41] and superscalar machines [4, 11, 13, 19].

To the best of our knowledge, the most recent existing works for supporting instruction scheduling in a verified compiler are that of [48], which achieved translation validation [36, 39] and a case study on list scheduling and trace scheduling [18, 22], and the more recent work on [44] which also used translation validation for a list-scheduling instruction scheduling pass for VLIW architecture and its follow-up papers [24, 45] that use the same approach to further support and improve inter-block instruction scheduling. However, there's a major difference between direct verification (the goal of this paper) and verified translation validation (past work), as explained in the next subsection.

1.3 Full Verification v.s. Verified Translation Validation

Full verification includes proving the correctness of the entire algorithm for a compiler pass, while verified translation validation includes only proving the correctness of the checker of the algorithm. In general, full verification is preferable to translation validation. However, it can be much more challenging due to its heavy verification requirements. A representative example is verifying register allocation compiler pass in CompCert using a graph coloring algorithm. In the work of [7], 4300 lines of Coq proof codes were used to formally verify the register allocation pass in CompCert, with a prior lemma of 10,000 lines proof on the graph coloring algorithm. Compared with this, the verified translation validation was also presented in the same year, with only 900 lines of proofs [42].

We summarize both the benefits and drawbacks of these two technical routes that aim to guarantee software safety:

Full Verification: verifying the correctness of an algorithm for a compiler pass,

- Core Technology: correctness of an algorithm
- **Pro:** Full correctness guarantee: verified scheduler has no bug and any execution instance will be correct.
- **Cons:** Development-time overhead: potentially heavy work and strongly related to algorithm implementation (our work will show there exists acceptable length of proofs for verifying instruction scheduling).
- **Pro:** No runtime overhead: correctness is checked during development with no overhead incurred at runtime.

Verified Translation Validation: correctness of each result from each input of an algorithm

- Core Technology: Symbolic Execution of source/target codes
- **Pro:** Simpler Correctness Proof and independent of algorithm implementations
- **Cons:** Partial correctness guarantee: only guarantee each execution instance will reject the wrong result by scheduler bugs. Also if the validator is incomplete, it may reject a valid program.
- **Cons:** Run-time overhead: correctness requires runtime validation and usually has high complexity

More formally, the correctness proof for the verified translation validator builds on the following lemma:

$$\forall p_s\ p_t : Compile(p_s) = OK\ p_t \rightarrow\ validate(p_s, p_t) = true \rightarrow Simulation(p_s, p_t)$$

while the result of direct verification is proved to have:

$$\forall p_s\ p_t : Compile(p_s) = OK\ p_t \rightarrow Simulation(p_s, p_t)$$

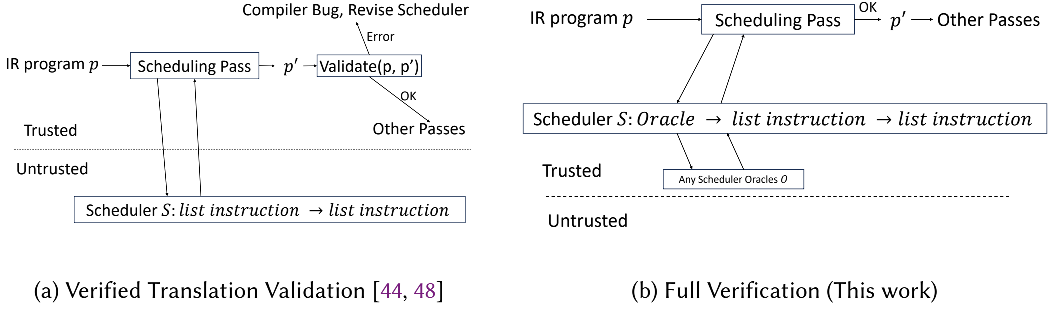
which is a much stronger result.

Fig. 1 shows the difference between verified translation validation and full verification in terms of instruction scheduling passes.

1.4 Contributions

In the conclusion of [48], the authors write that they believe it would be significantly more difficult to directly prove the correctness of list scheduling. However, our experience in this work shows that the proof burden can lead to relatively lightweight workloads for proof, compared with previous verification work on instruction scheduling.

In this work, we introduce the first known machine-independent correctness framework that supports a full correctness proof for intra-block scheduling, i.e. instruction scheduling within a basic



(a) Verified Translation Validation [44, 48]

(b) Full Verification (This work)

Fig. 1. Comparasion between two verification results of instruction scheduling

block. Our result is a once-for-all formal correctness framework with an acceptable length (around 3,000 lines of Coq code) for proving any instruction scheduling algorithm. We then implemented a case study on its usage of proving list scheduling on Risc-V architecture using critical path scheduling as a heuristic, which also shows optimization improvements on benchmarks PolyBench C 4.2. The proof of this case study has around further 1,000 lines of Coq code.

We summarize the highlights of our results as multiple levels of flexibility:

- Flexible algorithm changes: Our framework guarantees that any program transformation can be proved semantically sound by only proving that it satisfies the dependency relation defined in syntax level of the original program, without repeatedly reasoning anything about semantic details again. That means changing a scheduling algorithm only changes a small part of the proofs.
- Flexible instruction scheduling heuristics: as we implemented list scheduling as a case study, the heuristics to give priority when scheduling an instruction is an abstract parameter in our proof. This means it can be any heuristics that vary among different machine architectures, runtime environments, or even profiling results, etc., and the actual choice does not influence the correctness so it does not change proof codes. This part makes our methods equally flexible as previous translation validation methods
- Flexible Machine Architecture: combined with CompCert's original design, our implementation of correctness theorems is almost machine-independent except for only less than 100 lines of lemmas related to machine architecture.

We built our implementation of this theory into CompCert's backend (CompCert 3.12) thus it can directly improve the compiler effect of verified compilation of C programs and any other compiler engineering that uses CompCert's backends.

1.5 Structure of This Paper

Section 2 introduces some fundamental concepts related to instruction scheduling and basic settings of the language model of CompCert's low-level IR that make this paper's theory established from the ground up.

Section 3 briefly summarizes the main logical chain of our solution and the architecture of our implementation. Readers can refer to it for a quick overview of our approach.

Sections 4 and 5 give our complete theory following the logical chain introduced in Section 3. Section 5 shows how to use theories in Section 4 to prove the correctness (semantics preservation) of a concrete instruction scheduling algorithm without additional reasoning on semantics details.

Section 6 explains critical implementation details in Coq proof assistant based on CompCert's original framework and case implementation on list scheduling (excluding heuristics). Readers interested in Coq definition/proof details can refer to this section.

Section 7 introduces our case implementation of scheduling heuristic towards the RISC-V machine, together with Section 8 that presents the evaluation results in both view of proof engineering and compiler optimization performance of the scheduler we implemented for CompCert.

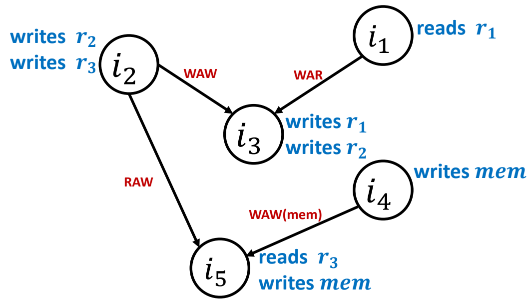
2 Background

In this section, we introduce several existing basic concepts and settings as a preparation for our work.

2.1 Compiler-level Instruction Scheduling

Pipeline stall or pipeline hazards is a common performance issue in pipelined architecture, which decreases the overall CPU throughput due to instruction-level data/control dependencies and hardware resource conflicts. Out-of-order execution is a key micro-architectural technique to avoid pipeline stall by dynamically reordering the instruction sequence, at the cost of hardware and algorithmic complexity in processor pipelines. To achieve higher energy efficiency and lower hardware cost, in-order processors are often adopted in areas including mobile devices and embedded systems [15, 49]. On in-order processors, it is the responsibility of compilers to avoid pipeline stall and enhance instruction-level parallelism for multi-issue pipelines. Instruction scheduling serves as the fundamental compiler pass that statically reorders the instruction sequence to improve instruction-level parallelism and reduce pipeline stall for in-order processors. Cited as an example, GCC implemented this pass in its O2 options.

Examples in Fig. 2b shows the effect of reordering independent instructions: it may reduce the cycles per instruction (CPI) by enabling parallel functional unit resource usage.



(a) Data dependence of some basic block $[i_1, i_2, i_3, i_4, i_5]$.

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			

(b) Effect of Instruction Scheduling. An instruction was filled in some cycle if it occupies that resource in that cycle.

Fig. 2. Example on instruction scheduling under some dependence

2.2 Dependence Relation

The execution of a program at machine's view can be considered as a sequence of instructions that are processed under program control flow. We revisit the basic concept of data dependence, which is the principal legality constraint of instruction scheduling.

Dependence relation is a binary relation between two program instructions that access a register or the same memory location and at least one is write access. A dependence relation, $i_1 \rightarrow i_2$

to denote i_1 must be in front of i_2, can be classified into three groups: (i) read-after-write (RAW) if instruction i_1 writes an identical register or memory location r and instruction i_2 reads r; (ii) write-after-read (WAR) if i_1 reads r and i_2 writes r; (iii) write-after-write (WAW) if i_1 writes r and i_2 writes r. The scheduling of example in Fig. 2b follows the dependence relation in Fig. 2a. There is an extensive body of literature to analyze whether two memory accesses are in isolated locations, which is orthogonal to the scope of this paper. We roughly define memory accesses in default have data dependence on each other in this paper.

Any compiler pass that reorders instructions must obey the rules that dependence relation is not changed. One of the major works of this paper is formally proving that as long as such rules are obeyed, the (mechanized) semantics preservation will be preserved.

2.3 Abstract Language of Low-level IR and Semantics Model of CompCert

We introduce the abstract model of the low-level IR where we conduct scheduling algorithms.

2.3.1 IR Language Model. For simplification, we ignored most of the irrelevant structure of IR and only kept the common three-address code that triggers the main formal proof burdens of this work.

***Definition* 1.** (Abstract IR Model)

- $instruction ::= I_{op}(op, r_{src}, r_{dst}) \mid I_{load}(addr, dst) \mid I_{store}(src, addr) \mid I_{call}(f) \mid I_{label} \mid I_{goto}$
- $function\ f ::= (Id_f,\ B_f = [i_1 :: i_2 :: \ldots :: i_m])$
- $program\ p ::= (Id_p^*,\ \{f_1, f_2, \ldots f_n\})$

2.3.2 Operational Semantics. The semantics of a program in CompCert is defined as a transition system (small step semantics) starting from some initial state and, if not entering an "infinite stuttering", ending at a final state. Readers can refer to Part IV of [2] for details. Here we give a concrete semantics model for low-level IR same way as [48], but omit some of the semantics details. A state $S = (t, Id_f, M, R, C)$ consists of state type $t \in \{Call,\ Regular,\ Return\}$, current function-id Id_f, memory states M, register states R, and remaining code. We also use $States(p)$ to denote the set of reachable program states of a program p. Here are some semantics rules of the IR, given the global environment G:

- **SEM-OP:** $\dfrac{i = I_{op}(op, r_{src}, r_{dst}),\ \ v = eval(op, r_{src}, r_{dst})}{G \vdash (Regular, f, M, R, i :: l) \rightsquigarrow_\epsilon (Regular, f, M, R[r_{dst} \leftarrow v], l)}$
- **SEM-LOAD** $\dfrac{i = I_{load}(addr, dst),\ \ v = load(M, addr)}{G \vdash (Regular, f, M, R, i :: l) \rightsquigarrow_\epsilon (Regular, M, R[r_{dst} \leftarrow v], l)}$
- **SEM-STORE** $\dfrac{i = I_{store}(src, addr),\ \ R[src] = v,\ M' = store(M, v)}{G \vdash (Regular, f, M, R, i :: l) \rightsquigarrow_\epsilon (Regular, f, M', R, l)}$

An initial state of a program is in the form $\mathbf{S_p^I} = (Call, \mathbf{M^I}, \mathbf{R^I}, C(f_{Id^*}))$. A state is said to be a final state if its in the shape $(Return, _, _, nil)$

2.3.3 Program Behavior. We define the behaviour of a program to be the event trace generated by it.

***Definition* 2.** (Program Behavior: halting) We say a program p behaves with a finite event trace e, denoted by $\mathcal{E}(p, e)$, if exists a final state S_f such that $\mathbf{S_p^I} \rightsquigarrow_e^* S_f$

***Definition* 3.** (Program Behavior: stuttering) We say a program p behaves with an infinite event trace e^∞, denoted by $\mathcal{E}(p, e^\infty)$, if exists a final state S_f such that $\mathbf{S_p^I} \rightsquigarrow_e^* S_f$

***Definition* 4.** (Program Behavior: error) We say a program p can result in an error with a finite event trace e, denoted by $\mathcal{E}(p, e)$, if exists a non-final states S_{err} such that $\mathbf{S_p^I} \rightsquigarrow_e^* S_{err}$ and there doesn't exist any state S such that $S_{err} \rightsquigarrow S$.

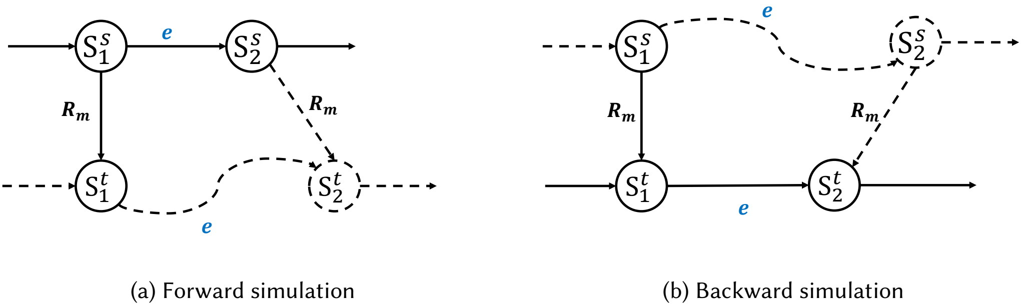
We also use $\mathcal{E}(p)$ to denote the set of all possible program behavior of a program p.

2.4 Compiler Correctness (Semantic Preservation)

The correctness of a compiler is defined as a refinement relationship: the behavior of target program is a "subset" of the behavior of source program.

***Definition* 5.** (Refinement Relation) We say the target program p_t is a refinement of the source program p_s if $\mathcal{E}(p_t) \subseteq \mathcal{E}(p_s)$, denoted by $p_t \sqsubseteq p_s$

The simulation relation that implies the refinement relation in this paper is discussed between two programs in the same intermediate language. There are two types of semantics preservation: forward simulation and backward simulation. The backward simulation of any source program and its compiled target program is the final result we need to reach, since it directly implies the refinement relation.



(a) Forward simulation (b) Backward simulation

Fig. 3. Simulation relations between program's small-step semantics

***Definition* 6.** (Backward Simulation, Fig. 3b) The backward simulation $\mathcal{B}(p_s,\ p_t) :=$ is satisfied if exists a matching relation relation $R_m \in State \times State$ between states such that $\forall\ S_1^t\ S_2^t \in States(p_t).\ \forall S_1^s\ \in States(p_s).\forall e \in Events.R_m(S_1^s, S_1^t) \to S_1^t \rightsquigarrow_e S_2^t \to \exists S_2^s.\ S_1^s \rightsquigarrow_e S_2^s \wedge R_m(S_2^s, S_2^t)$

However, the backward property is usually hard to prove directly. All the previous proof engineering used the trick that proves forward simulation first, which is usually straightforward, then used a theorem that forward simulation implies backward simulation as long as the compiled program is deterministic.

***Definition* 7.** (Forward Simulation. 3a) The forward simulation $\mathcal{F}(p_s,\ p_t)$ is defined like flipping backward simulation as: exists a matching relation $R_m \in State \times State$ between states such that $\forall\ S_1^s\ S_2^s\ \in\ States(p_s).\ \forall S_1^t\ \in\ States(p_t).\forall e\ \in\ Events.R_m(S_1^s, S_1^t)\ \to\ S_1^s\ \rightsquigarrow_e\ S_2^s\ \to\ \exists S_2^t.\ S_1^t\ \rightsquigarrow_e S_2^t \wedge R_m(S_2^s, S_2^t)$

***Definition* 8.** (Determinism) A program p is said to be deterministic if for all program state S such that $\mathrm{S_p^I} \to^* S$, there exists at most one program states S' and event sequence e such that $S \rightsquigarrow_e S'$

The following lemmas was proved in general theory of the original work of CompCert.

Lemma 1. Forward simulation implies backward simulation if target program is deterministic: forall program p_s and p_t, if p_s is deterministic and $\mathcal{F}(p_s, p_t)$, then $\mathcal{B}(p_s, p_t)$

Lemma 2. Forward simulation is transitive.

Lemma 3. Backward Simulation Implies Behavior Refinement: forall program p, if $\mathcal{B}(p_s, p_t)$ then $\mathcal{E}(p_s) \subseteq \mathcal{E}(p_t)$.

3 Overview of Our Approach

Figure. 4 shows the structure of our system, and how it was incorporated into the CompCert project. The boxes and arrows in black denote the compiler passes of the original CompCert project. Those in red denote our extension of instruction scheduling pass and heuristics, accompanied by our correctness proof in Coq.

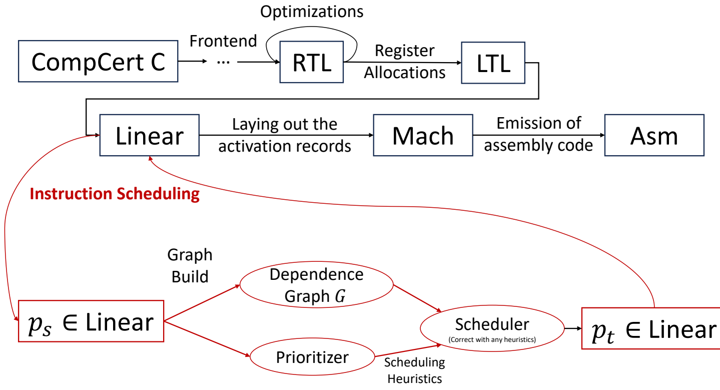


Fig. 4. Architecture of our system

Our correctness proof includes the following components, which are discussed in more detail in the following sections:

(1) **Swapping-lemma:** A topological reordering of a list of partially ordered elements is equivalent to a finite sequence of swaps of adjacent non-ordered elements. See Fig. 5. A restricted version of this result was proved in [29] [1].
(2) **Rule of instruction scheduling:** Any valid scheduler (or other compiler pass) that reorders a program's instructions must obey the dependence constraints of the original program, i.e., conduct a toplogical reordering (topo-reorder) based on the dependence relation.
(3) **Decompose scheduler:** Combining (1) and (2), any valid scheduler (or other compiler pass) that reorders a program's instructions within a basic block is equivalent to a finite sequence of compiler passes that only swap one pair of instructions not ordered by the original program dependence (a.k.a. independent instructions).
(4) **Swapping correctness:** Swapping only one pair of adjacent independent instructions inside only one basic block of a program preserves the semantics of the program.
(5) **Transitivity of semantics preservation:** if two program transformations preserve program semantics, composing them also preserves it. Recursively, composing a finite sequence of semantics-preserving transformation preserves program semantics
(6) **Final result:** Combining (3) (4) (5), given any instruction scheduler, as long as it preserves the instruction dependence relations of the original program, it preserves the semantics of the original program. See Fig. 6

[1]This paper proved that there exists a sequence of all permutations of a set that adjacent two permutations only differs on one pair of adjacent elements was swapped. In our work, the conclusion has a different requirement that only unordered elements can be swapped.

4 Main Theorem

The structure of our main theorem follows the six steps summarized in Section 3, which are described in detail in this section. We use formal mathematical notation for lemmas and proofs in this section that reflect the structure of the proof that we developed using the Coq proof assistant.

4.1 Topological Sort and Topological Re-ordering

Assuming $i \in N^+$, we use $l[i]$ to denote the ith element of a list l of elements taken from set A ($i <=$ size of l by default), and $\mathbf{E}_l$ to denote the set of all elements in l. We also assume that there are no duplicate elements in list l.

***Definition* 9.** (Topo-sorted List) Given l and a partial order R on $\mathbf{E}_l$, a list of elements from $\mathbf{E}_l$ is said to be an topo-sorted list by R if $\forall i_1\ i_2 \in N, Rl[i_1]l[i_2] \longrightarrow i_1 < i_2$.

***Definition* 10.** (Generated Order by Position) Given a no-duplicate list l of elements from some set A, a partial order relation R on $\mathbf{E}_l$, we define a generated order by position (GOP) of l using R, denoted by $\mathcal{G}_l^R$, to be: $\forall i_1\ i_2\ \mathcal{G}_l^R\ l[i_1]\ l[i_2]$ iff $i_1 < i_2 \wedge R\ l[i_1]\ l[i_2]$.

Lemma 4. A list is topo-sorted by its own GOP: for any no-duplicate list l with length n and relation R on A, l is a topo-sorted list $\mathcal{G}_l^R$

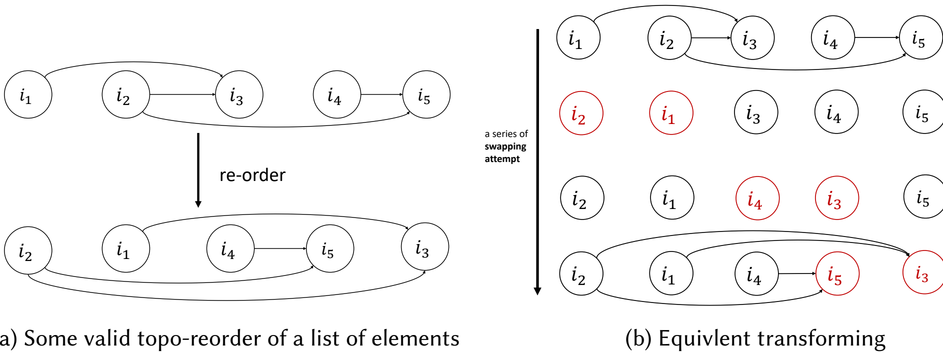
Proof. Immediately by definition of GOP. □

In later sections, we will see that this abstract definition of GOP represents the data dependence definition within a basic block. That is, R will be instantiated by data dependence definition between two instructions (RAW/WAR/WAW), and $\mathcal{G}_l^R$ will be the dependence relation derived from a whole basic block (a.k.a happens-before relation inside a basic block).

4.2 Swapping Lemma

We introduce a simple but the most important mathematical property of topological orders used in this work. It is also one of the core ideas to reduce the verification work's hardship to only one complicated but trivial lemma. We name the property swapping lemma. We prove it in a purely mathematical way independent from the compiler engineering.

This lemma summarizes that, given a topo-sorted list of elements following some order, any topological reordering of this list is equal to a finite sequence of swapping adjacent unordered elements. See Fig. 5 for an illustration: all the three swapped pair (i_1, i_2), (i_3, i_4), (i_3, i_5),



(a) Some valid topo-reorder of a list of elements (b) Equivlent transforming

Fig. 5. Illustration of Swapping Lemma

Now we formally define and prove it.

***Definition* 11.** (Swapping Attempt) Given a set A, relation R on A, and an topo-sorted list $l = [i_0, i_1, ..., i_n]$ by R containing a finite number of elements of A, we say a swapping attempt of l at location $k \in N$, denoted as $l' = SA^R(l, k)$ is a transformation from $l = [i_0, ..., i_k, i_{k+1}, ..., i_n]$ to $l' = [i_0, ..., i_{k+1}, i_k, ..., i_n]$ if $R\ i_k\ i_{k+1}$ does not hold or $l' = l$ if $R\ i_k\ i_{k+1}$ holds. We also extend this definition to a list of natural numbers recursively as a sequence of swapping attempts by $SA(l, [n_1, n_2, ...]) = SA^R(SA^R(l, n_1), [n_2, ...])$.

***Definition* 12.** (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.

Lemma 5. (Swapping Lemma) Given a relation R, a topo-sorted list l, for any l's topological reorder l', exists a finite list of nature number l_n such that $l' = SA^R(l, l_n)$

Proof. This lemma declares that a topological reorder of a list is equivalent to a series of swappings of adjacent elements. We prove this by induction on the length of l. Base case is trivial. Suppose the conclusion holds for any l with $1 \leq length(l) \leq k$, we prove it also holds for any l_* that $length(l_*) = k + 1$. Given an l_*'s topological reorder l'_* we destruct it by two cases:

- If $l_*[0] = l'_*[0]$, then the remaining parts of these two lists are also a pair of topological reordering with length k, and the conclusion holds by induction hypothesis.
- Otherwise, writing $l'_* = [i_0, i_1, ..., i_{k+1}]$, l_* can be separated into $l_1 + +[i_0] + +l_2$. Since l'_* is topo-sorted, for any i_j in l_1 or l_2, $Ri_j i_0$ does not hold. This means we can get $l''_* = [i_0]{+}{+}l_1{+}{+}l_2$ from $l_* = l_1 + +[i_0] + +l_2$ by swapping i_0 with every elements of l_1 one by one and l''_* is still topo-sorted. Since l''_* has the same head and same elements as l'_*, by induction hypothesis, same as the previous case, l'_* can be derived from swapping a sequence of adjacent elements of l''_*. Connecting with the previous swapping sequence from l_* to l''_*, we can construct the final swapping sequence from l_* to l'_*

□

4.3 Rule of Valid Scheduler

We define an instruction scheduler as a transformation on a list of instructions indexed by their original position at the list so that the list to schedule is always non-duplicate.

***Definition* 13.** (Instruction Scheduler) Given a indexed list of instructions $l = [i_1, i_2, ..., i_n]$, a scheduler is a function $\mathcal{S} : list\ instruction \rightarrow list\ instruction$ [2] such that l and $\mathcal{S}(l)$ have exactly the same elements, i.e. $\mathbf{E}_l = \mathbf{E}_{\mathcal{S}(l)}$. We also use the same symbol $\mathcal{S}(p)$ for scheduler of a whole program p

The dependence relation between two instructions depends on their operation to registers and memory:

***Definition* 14.** (Dependence Relation) $\mathcal{D} = \mathcal{D}_{RAW} \bigcup \mathcal{D}_{WAR} \bigcup \mathcal{D}_{WAW} \bigcup \mathcal{D}_{solid}$ where: (i) $\mathcal{D}_{RAW} i_1 i_2$ iff $writes(i_1) = reads(i_2)$, (ii) $\mathcal{D}_{WAR} i_1 i_2$ iff $reads(i_1) = writes(i_2)$, (ii) $\mathcal{D}_{WAW} i_1 i_2$ iff $writes(i_1) = writes(i_2)$, (iv) $\mathcal{D}_{solid} i_1 i_2$ iff one of i_1 and i_2 was a function call or branch jump, or writes to a memory.

A valid instruction scheduler should always follow this dependence relation at syntax level:

***Definition* 15.** (Valid Instruction Scheduler) An instruction scheduler $\mathcal{S}$ is said to be valid if for any list of instructions l, $\mathcal{S}(l)$ is a topo-reorder of l by $\mathcal{G}_l^{\mathcal{D}}$.

[2] The index was omitted from the scheduler's type for simplification.

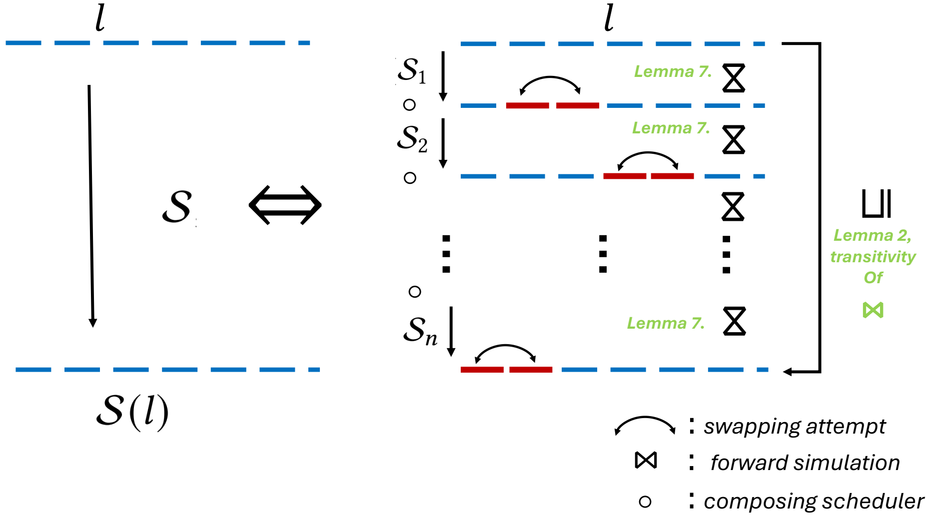


Fig. 6. Structure of our proofs

4.4 Equivalence of Valid Schedulers

***Definition* 16.** (Single Swapper) A single swapper $\mathcal{S}_s^n$ where $n \in N$ is a special kind of instruction scheduler that only tries to swap one pair of adjacent instructions with a parameter to specify the swapping location defined inductively: (i) $\mathcal{S}_s^n$ $([]) = []$ for any $n \in N$, (ii)$\mathcal{S}_s^{n+1}$ $(i :: l) = i :: \mathcal{S}_s^n$ (l), (iii)$\mathcal{S}_s^0$ $(i_i :: i_2 :: l) = i_2 :: i_1 :: l)$ if not $\mathcal{D}i_1 i_2$, (iv) $\mathcal{S}_s^0$ $(i_i :: i_2 :: l) = i_1 :: i_2 :: l$ if $\mathcal{D}i_1 i_2$.

We also use the same symbol $\mathcal{S}_s^n(f, p)$ to denote swapping a single pair in side code blocks of the function f in a program p

***Definition* 17.** (Composing Scheduler) Given a list of scheduler $L_{\mathcal{S}} = [\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_n]$, we directly use $L_{\mathcal{S}}$ as the function by composing the scheduler in it one by one, i.e. $L_{\mathcal{S}}(l) = \mathcal{S}_n(\mathcal{S}_{n-1}(\ldots\mathcal{S}_1(l)))$

Now we can conclude that a valid scheduler is equal to a composing of a sequence of single swappers.

Lemma 6. (decomposing lemma) Given a valid scheduler $\mathcal{S}$, there exists $L_{\mathcal{S}} = [\mathcal{S}_s^{i_1}, \ldots, \mathcal{S}_s^{i_n}]$ such that for any l, $\mathcal{S}(l) = L_{\mathcal{S}}(l)$.

Proof. According to the definition of valid scheduler, since both l and $\mathcal{S}(l)$ are sorted by $\mathcal{G}_l^{\mathcal{D}}$, this property is proved by swapping lemma. □

4.5 Swapping Correctness and the Final Theorem

We have proved in Coq that swapping a single pair of adjacent independent instructions still preserves the forward simulation relation of the program.

Lemma 7. (correctness of single swapper) Given a single swapper $\mathcal{S}_s^n$ of a program, $\mathcal{F}(p, \mathcal{S}_s^n(p))$.

Lemma 7 is the only lemma that requires us to reason about the different cases of semantic details of IR execution. Briefly, we have to reason case by case that executing two independent instructions consecutively results in the same memory and register states no matter which one

was executed first. Till here, we have shown how breaking a scheduler down to single swappers made the verification of a scheduler easier.

Together with Lemma 6 and 2, we can prove that any valid scheduler of a program preserves the forward simulation relation, thus can be incorporated into the original CompCert system.

Lemma 8. For any valid instruction scheduler $\mathcal{S}$ of a program p, $\mathcal{F}(p, \mathcal{S}(p))$.

The proof procedure of Lemma 8 is described in Fig. 6.

5 Correctness of List Scheduling

We show how our framework works when proving a concrete instruction scheduling algorithm in this section, i.e., list scheduling (one of the simplest intra-block scheduling algorithms). As mentioned in Section 1.2, no previous work had ever achieved a formally verified compiler pass for list scheduling, even for such a simple algorithm. For list scheduling, the work from [44, 48] used verified translation validation to check the validity each time a program is compiled.

The scheduling algorithm that we focus on in this proof consists of three parts: dependence graph, scheduling heuristics, and iterative scheduling. With our previous theorem, we only have to prove that our algorithm is a valid scheduler. This proof will no longer involve semantics details since the definition of a correct schedule is only based on the dependence relation, $\mathcal{D}$, defined at the syntax level.

Our algorithm was implemented and proved in Coq and can be directly used by the CompCert project.

5.1 Dependence Graph Construction and Scheduling Heuristics

Scheduling a block of instructions requires us to construct a dependence graph that records the dependence relation inside a basic block. This step is noted in Algorithm. 1.

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**
    $\mathbf{G}.nodes \leftarrow \mathbf{E}_l$
    $\mathbf{G}.edges \leftarrow \emptyset$
  else if $l = i' :: l'$ **then**
    $\mathbf{G}.edges \leftarrow \mathbf{G}.edges \cup \{(i', i) | i \in l' \land \mathcal{D} i' i\}$
    $\mathbf{G}.edges \leftarrow \mathbf{G}.edges \cup DRel(l').edges$
  end if

After building the dependence graph, our scheduling algorithm will need a heuristic (oracle) to make choices among several available instructions to be scheduled during each step in iterative scheduling. The heuristic (also named *prioritizer* in our implementation) we use is an abstract parameter $\mathcal{P}$: *list instruction* $\rightarrow$ *list instruction*. In the actual implementation, it takes the original basic block as input, and returns the priority of each instruction based on analyzing some performance aspects (e.g., specific architectures, clock time of instruction types, etc). In each iteration, our scheduler will pick an instruction with the highest priority.

The details of heuristics do not influence the correctness of the scheduling algorithm, but only the performance of scheduled code. That means no matter how unreasonable the priority function is, the scheduled code should be correct in semantics. Therefore, we do not introduce a concrete heuristic in this section but leave that to our final instantiation during experiments in Section 7.

5.2 Scheduling Algorithm

The list scheduling algorithm we are going to prove correctness for (Algorithm. 2) is an iterative processing algorithm on the dependence graph. It has exactly the same number of iterations as the length of the instruction list (basic block). Each iteration identifies all nodes that do not depend on any other nodes, and picks one according to the priority from the input heuristics.

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 be the priority of i_k
  $l^* \leftarrow []$
  while G not empty **do**
    A $\leftarrow \{i_k \in l \mid \forall i_{k'} \in l.(i_{k'}, i_k) \notin$ 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^*

5.3 Proving Correctness

Now we prove that for any scheduling heuristic $\mathcal{P}$, our list scheduler $\mathcal{S}^*(\mathcal{P})$ is correct. Based on Lemma 8, all we need to do is to prove $\mathcal{S}^*(\mathcal{P})$ is a valid scheduler, a.k.a. $\mathcal{S}^*(\mathcal{P}, l)$ will generate an l' that is a topo-reorder of l by the generated order from l. To achieve this, we first prove that the dependence graph represents $\mathcal{G}_l^{\mathcal{D}}$ correctly. Then we prove that an invariant was preserved during the iterative scheduling process.

5.3.1 Graph Construction. We firstly proved that the graph we constructed from given l using Algorithm. 1 correctly stores the relation of $\mathcal{G}_l^{\mathcal{D}}$.

Lemma 9. Given a list of instruction $l = [i_1, i_2, ...]$, $\forall i_j, i_k \in DRel(l).edges$ we have $\mathcal{G}_l^{\mathcal{D}} i_j i_k$.

5.3.2 Scheduling Invariant: In the loop inside Algorithm 2, we proposed the following invariant:

***Definition* 18.** (Scheduling Invariant) The invariant during the iterative scheduling process consists of the following assertions:

- **LENGTH:** $|l^*| + |\text{G}.node| = |l|$
- **SUBSET:** $\text{G}.nodes \in \text{E}_l \wedge \text{E}_{l^*} \in \text{E}_l$
- **INTERSECTION:** $\text{G}.node \cap \text{E}_l = \emptyset$
- **NGT:** $\forall i_j \in \text{E}_l, i_k \in G, (i_k, i_j) \notin DRel(l).edges$
- **SORT:** l^* is sorted by $\mathcal{G}_l^{\mathcal{D}}$

We prove that this invariant is preserved during scheduling:

Lemma 10. (Invariant Preservation) The invariant in Definition 18 was preserved in the loop inside Algorithm 2.

Proof.

- **LENGTH, SUBSET,** and **INTERSECTION:** One node was removed from **G** to the tail of l^* in each iteration
- **NGT:** let the new graph be G' after removing i_{k^*} from it. For any $i_j \in \mathbf{E}_{l++[i_{k^*}]}$, if $i_j = i_{k^*}$, since $i_{k^*} \in G.node$ was the one with no incoming edges, this invariant was still preserved since $\forall i_k \in G', (i_k, i_{k^*}) \notin DRel(l).edges$. Otherwise if $i_j \in \mathbf{E}_l$, this invariant was also preserved immediately.
- **SORT:** by combining **NGT** and Lemma 9.

□

Now, we establish the property that the scheduled instructions have the same elements as before our schedule:

Lemma 11. For any scheduling heuristic $\mathcal{P}$, given an l and $l' = \mathcal{S}^*(\mathcal{P}, l)$, $\mathbf{E}_l = \mathbf{E}_{l'}$

Proof. In iterative scheduling, one instruction was removed from **G** to the tail of l^* in each iteration until **G** becomes empty. □

5.3.3 *Validity:* Now we can immediately reach the conclusion that our algorithm is a valid scheduler.

Lemma 12. Given a list of instruction l and scheduling heuristic $\mathcal{P}$, $\mathcal{S}^*(\mathcal{P}, l)$ is a topo-reorder of l by $\mathcal{G}_l^{\mathcal{D}}$ i.e. Algorithm 2. provides a valid scheduler.

Proof. Combining Lemma 11. and **SORT** of the preserved invariant after scheduling. □

5.3.4 *Final Correctness:* With the conclusion from Section.4, our algorithm preserves the forward simulation relation of a program.

Theorem 1. Scheduler in Algorithm 2 preserved the forward simulation relation of program semantics regardless of scheduling heuristic: for any program p and heuristic $\mathcal{P}$, $\mathcal{F}(p, \mathcal{S}^*(\mathcal{P}, p))$.

Proof. Immediately, by combining Lemma 8 and Lemma 12. □

Incorporating this theorem into original CompCert theories, our new compiler still preserves the backward simulation between source C program and compiled Asm program.

6 Framework Implementations in Coq Proof Assistant

This section describes some essential details of the formalization of our theory in Coq. The establishment of formal Coq theorems follows the same route as the paper-written mathematical theorems Section 4 and 5. However, the detail of implementation involves much more trivial definitions or parameters. For example, to distinguish repeated same instructions, we have to give an index to each instruction in a list so the the list we are scheduling becomes non-duplicate.

```
Context {A: Type}.

Fixpoint numlistgen' (l: list A) (n: positive): list (positive * A) :=
  match l with
  | [] => []
  | x :: l' => (n, x) :: numlistgen' l' (n + 1)
  end.

Definition numlistgen (l: list A) := numlistgen' l 1.
...
Lemma numlistgen_NoDup: forall l, NoDup (numlistgen l).
```

6.1 Formalization of Topological Properties and Swapping Lemma

6.1.1 topo-sorted and topo-reorder. Given a relation R on some type A, the topo-sorted property and topo-reorder relation on list of A is defined inductively:

```
Context {A: Type}.
Variable R: A -> A -> Prop.

(* not greater than any elements in list *)
Inductive NoDupSame: list A -> list A -> Prop :=
| NoDupSame_intro:
    forall l1 l2, NoDup l1 -> NoDup l2 -> incl l1 l2 -> incl l2 l1 -> NoDupSame l1 l2.

Inductive ngt (a: A): list A -> Prop :=
| ngt_nil: ngt a []
| ngt_cons: forall x l, ngt a l -> ~ R x a -> ngt a (x :: l).

Inductive topo_sorted: list A -> Prop :=
| topo_sorted_nil: topo_sorted []
| topo_sorted_cons: forall x l, ngt x l -> topo_sorted l -> topo_sorted (x :: l).

Inductive topo_reorder : list A -> list A -> Prop :=
| topo_reorder_nil: topo_reorder [] []
| topo_reorder_skip x l l' : ngt x l -> topo_reorder l l' -> topo_reorder (x::l) (x::l')
| topo_reorder_swap x y l : (~ R x y) -> (~ R y x) -> topo_reorder (y::x::l) (x::y::l)
| topo_reorder_trans l l' l'' :
  topo_reorder l l' -> topo_reorder l' l'' -> topo_reorder l l''.
```

Note that the topo-reorder we defined in Coq is an alternative but equal form of the one we defined in Definition 12. This makes our proof of swapping lemma more convenient since the constructor already contains the 'swapping'.

The induction on the length of the list was conducted in the following lemmas and theorem. Note that this proof actually corresponds to the proof of swapping lemma in Lemma 5 due to the alternative definition of topo-reorder.

```
Lemma sorted_same_elements_topo_reorder_ind:
  forall n,
    (forall k l1 l2, k < n -> length l1 = k -> NoDupSame l1 l2 ->
            topo_sorted l1 -> topo_sorted l2 -> topo_reorder l1 l2) ->
    (forall l1 l2, length l1 = n -> NoDupSame l1 l2 ->
            topo_sorted l1 -> topo_sorted l2 -> topo_reorder l1 l2) .

Lemma sorted_same_elements_topo_reorder':
  forall n l1 l2, length l1 = n -> NoDupSame l1 l2 ->
    topo_sorted l1 -> topo_sorted l2 -> topo_reorder l1 l2.

Theorem sorted_same_elements_topo_reorder:
  forall l1 l2,  NoDupSame l1 l2 ->
    topo_sorted l1 -> topo_sorted l2 -> topo_reorder l1 l2.
```

6.1.2 swapping lemma. Now we can show how the final theorem of the swapping lemma was formalized in Coq.

The swapping attempt (Definition 11) was formalized by

```
Context {A: Type}.
Variable (rel: A -> A -> bool).
(* swapping attempt at location n *)
```

```
Fixpoint try_swap (n: nat) (l: list A): list A :=
    match n, l with
    | _, nil => nil | _, i :: nil => l        (* None: do not swap *)
    | O, i1 :: i2 :: l' => if rel i1 i2 then l
                           else (i2 :: i1 :: l')
    | Datatypes.S n', i :: l' => i :: try_swap n' l'
    end.
(* a sequence of swapping attempts *)
Fixpoint try_swap_seq (ln: list nat) (la: list A): list A :=
    match ln with
    | nil => la
    | n :: ln' => try_swap_seq ln' (try_swap n la)
    end.
```

Then the final swapping lemma was formalized by

```
Context {A: Type}.
Variable Rb: A -> A -> bool.

Theorem swapping_property:
  forall l nl', (treorder R l) (numlistgen l) nl' ->
    exists ln, nl' = try_swap_seq Rbnum ln (numlistgen l).
```

6.1.3 Generated order by position. The actual order of a basic block comes from both their original position and their dependence relation to others. In Coq, $\mathcal{D}$ in Definition 10 was formalized as

```
(* Generated order by position from a list,  aux. definition for simpler proofs *)
Inductive GenR' (i: positive) (na1 na2: positive * A): Prop :=
  GenR_intro: List.In na1 (numlistgen' l i) -> List.In na2 (numlistgen' l i) ->
    fst na1 < fst na2 -> R (snd na1) (snd na2)  -> GenR' i na1 na2.
(* Generated order by position from a list *)
Definition GenR := GenR' 1.
```

6.2 Forward Simulation Preservation of Abstract Scheduler

Before showing the final lemma of the correctness of single swapper, we wish to show the only lemma that involves the semantics details of the Linear IR we work on and contributes to most of the proof burden of this work. This is the core lemma of proving that the single swapper preserves the forward simulation of the IR (Lemma. 7).

```
Lemma independent_two_step_match:
  forall stk stk' f f' sp sp' c rs rs' m m' s3 i1 i2 t
    (INDEP: i1 D~> i2 = false)
    (s1:= State stk f sp (i1::i2::c) rs m)
    (STEP13: starN step ge 2 s1 t s3)
    (s1':= State stk' f' sp' (i2::i1::c) rs' m')
    (MAT: match_states s1 s1'),
      exists s3', tPlus s1' t s3' /\ match_states s3 s3'.
```

The detailed definition of the program state of Linear IR and forward simulation relation can be refered at [14]. After that, then the semantics preservation of the single swapper was formalized by

```
Fixpoint transf_program_try_swap_seq1 (seq: list (nat * nat) ) (prog: program):=
  match seq with
  | [] => OK prog
  | (pos, n) :: seq' => do prog' <- transf_program_try_swap_in_one pos n prog;
                        transf_program_try_swap_seq1 seq' prog'
  end.

Lemma transf_program_multi_swap_forward_simulation1:
```

```
  forall seq prog tprog,
    transf_program_try_swap_seq1 seq prog = OK tprog ->
      forward_simulation (Linear.semantics prog) (Linear.semantics tprog).
```

The abstract scheduler is defined to have a parameter of function type from a list of indexed instruction to another list of indexed instructions, with a hypothesis that the result of it generated a topo-reorder. Here the happens_before corresponds to the data dependence relation $\mathcal{D}$ in Definition 14

```
Variable schedule': list (positive * instruction) -> res (list (positive * instruction)).

Let HBR := fun i1 i2 => happens_before i1 i2 = true.
Let HBnum (na1 na2: positive * instruction) := happens_before (snd na1) (snd na2).
Let HBGenR (l: list instruction) := GenR HBR l.

Hypothesis schedule_valid:
  forall l nl', schedule' (numlistgen l) = OK nl' ->
    treorder HBR l (numlistgen l) nl'.

Definition schedule_program (p: program): res program := ... (* based on schedule'  *)

Theorem schedule_program_forward_simulation:
  forall prog tprog: program, schedule_program prog = OK tprog ->
    forward_simulation (Linear.semantics prog) (Linear.semantics tprog).
```

6.3 Case Implementation on List Scheduling and Correctness

The implementation of our scheduler takes a heuristic function as an abstract parameter and builds a lookup table indexed by the location of each instruction based on it.

```
Variable prioritizer: list instruction -> list positive.
Fixpoint prio_map' (cur: positive) (lp: list positive): PMap.t positive :=
    match lp with
    | nil => PMap.init 1
    | p :: lp' => PMap.set cur p (prio_map' (cur + 1) lp')
    end.
Definition prio_map (lp: list positive) := prio_map' 1 lp.
...
(* return the one to schedule and the new dependence graph after removing it *)
Definition schedule_1 (prior: PMap.t positive) (original: DPMap_t)
 (scheduled: list (positive * instruction)) (remain: DPMap_t)
  : res (list (positive * instruction) * DPMap_t) :=
    let available := indep_nodes remain in
    do pi <- firstpick prior available;
    OK (scheduled ++ [pi], remove_node (fst pi) remain).

Fixpoint schedule_n (prior: PMap.t positive) (L: nat) (original: DPMap_t)
  (scheduled: list (positive * instruction)) (remain: DPMap_t)
  : res (list (positive * instruction) * DPMap_t) :=
    match L with
    | O => OK (scheduled, remain)
    | Datatypes.S L' =>
        do (scheduled', remain') <- schedule_1 prior original scheduled remain;
        schedule_n prior L' original scheduled' remain'
    end.

Definition schedule_numblock (nl: list (positive * instruction)) :=
  let m := dep_map_gen nl in (* dependence graph *)
  let prior := prio_map (prioritizer (numlistoff nl)) in
```

```
    do (nl', m) <- schedule_n prior (List.length nl) m [] m;
    OK nl'.

Definition list_schedule' := schedule_program schedule_numblock.
```

The list scheduler with arbitrary prioritizer is proven to preserve the forward simulation of Linear IR through the invariant we used in Lemma 10.

```
Lemma schedule_numblock_correct:
  forall l nl', schedule_numblock (numlistgen l) = OK nl' ->
    treorder HBR l (numlistgen l) nl'.

Theorem abstract_list_schedule_forward_simulation:
  forall prog tprog, list_schedule' prog = OK tprog ->
    forward_simulation (Linear.semantics prog) (Linear.semantics tprog).
Proof.
  intros. eapply schedule_program_forward_simulation; eauto.
  eapply schedule_numblock_correct.
Qed.
```

7 Scheduling Heuristics Implementation for Specified Architecture

In this section, we show how we implement the scheduling heuristic for our target RISC-V Machine. Although it is feasible to completely implement everything in Coq and generate the improved compiler, we chose to use a trick that makes it possible for developers who do not have knowledge of Coq/OCaml systems to collaborate with us. We believe this further improves the flexibility of our result.

The design of CompCert works in the following way. The compiler passes were both implemented and proved in Coq. To make it an executable file that works the same way as GCC or Clang, it uses the Extraction function of Coq to convert the composition of those passes into an Ocaml function of type *C syntax* $\rightarrow$ *Assembly code* (the principle of Extraction can be found at the Coq textbook [3], chapter Extract). Note that such conversion from Coq to OCaml is not guaranteed safe, which is the well-known trusted computing base of CompCert.

As we mentioned, the scheduling heuristic is abstracted away in algorithm implementation as a parameter of type *list instruction* $\rightarrow$ *list N*. Therefore, instead of implementing it in Coq and then extract to OCaml, we can safely implement it directly in OCaml. We have learned that the OCaml language supports interfacing C with itself [34, 47]. We use this support in our OCaml function to invoke a C function that customizes the scheduling priority towards Risc-V. The reason we doing this instead of continuing the implementation in Coq and build the compiler in previous way is that we believe this both reduces the learning process of compiler engineers who are not familiar enough with Coq/OCaml development and proof engineers who are not familiar enough with specific machine architectures.

Nevertheless, using such a method is not necessary but just an option. Future developers that use our result to improve the performance of instruction scheduling can freely choose to modify the C function under the same interface, modify under the OCaml function without using C interface, or modify under the Coq scheduler without using abstract parameters but a concrete heuristic implemented in Coq. Any of the above choices will not change any of the proof codes.

7.1 Critical Path Scheduling

As our instruction scheduling heuristics, we use the Critical Path (CP) method [12, 28], where the instruction with the longest path length has the highest priority. Given a dependence graph whose node represents an instruction and an edge represents inter-instruction dependence, the path length of $node_i$ to the exit node is computed as:

$$path_i = max_{j \in succ_i}(path_j) + cost_i$$

where $succ_i$ is the dependence successors of $node_i$, $cost_i$ is the instruction latency of $node_i$, and exit node is the zero-cost dummy node that precedes all other nodes. Our target machine used for the performance study in Section 8.2 is SiFive U74-MC Core Complex. We collected the instruction latencies from their architecture manual [49] and summarized them as a cost table, i.e., a map of operator type to corresponding latency.

The CP method is a fast and efficient heuristic scheduling algorithm, which can enhance instruction-level parallelism and achieve near-optimal scheduling results in practice. Our certified instruction scheduling uses the above path length as the priority among available instructions (Section 5.2).

7.2 Interaction between Coq/OCaml and C function

7.2.1 Coq-OCaml Interface. In the Coq part, we need a parameter to represent the outside world function from OCaml:

```
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
  List.map pos_of_int prior'.
```

The prioritize function that is manually implemented in OCaml takes the parameters of a list of nodes encoding the instruction's operation type and edges of the same dependence graph from which the scheduling algorithm is processing. It further passed the parameters to invoke the C functions that implement the CP heuristic using the Ctypes [3] library (mentioned in Chapter 22 of [34]) , and get a list of integers that represent the priority of each node.

```
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
```

7.2.2 OCaml-C interface. We then implemented the following C function to compute the path-based priority discussed in Section 7.1, where `nodes` and `edges` respectively capture operator

[3]This library of OCaml has a name conflict with a module in CompCert. We have to change the name of that module when implementing the heuristic in this way

types and dependence edges while the return value represents the computed priorities for given nodes:

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

8 Evaluation

We evaluate our result in two different views: proof workload of our verification and optimization performance of our implementation case of list scheduling with critical path scheduling heuristics. We will show that our proofs have acceptable length and are comparably as lightweight as previous work in lines of codes (LOC) with the same coding style as the open source code of CompCert. We will also show that our implementation case results in an improved compiler optimization on benchmarks.

8.1 View 1: Proof Engineering

We believe our verification work is relatively lightweight in two aspects. Firstly, we used a similar amount of LOC on proofs as in previous work on validating intra-block scheduling. Secondly, 75% of our work are once-for-all result. We believe future work on different scheduling algorithms inside a basic block can directly use our result to finish proof without any reasoning on semantics details again, i.e. we also make future work lightweight.

Table. 1 shows the proof workload of our verification work on instruction scheduling.

Table 1. LOC of program/functions and proofs in our work

	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 code (Risc-V)	Coq	-	40
Machine dependent code (x86)	Coq	-	35

Table. 2 shows proof workload of both our and previous work. Note that the total goal of the three work are not exactly the same: [48] also implemented a validator for trace scheduling and [44] did some specific work to support VLIW instruction parallelism, while our work verified a machine-independent scheduling algorithm with machine dependent heuristics. A remark is that our work has a stronger result, i.e. correctness of an algorithm. Both the work of [44, 48] only guarantees the correctness of the translation validator, a.k.a guarantees the correctness of each compiled case that does not return an error message. With only a verified translation validator, the compilation will be aborted if the unverified scheduler generates a wrong result.

Table 2. LOC of related work

	Fully verified	Scope	LOC of proof codes
This work	Yes	list scheduling	4k
[48]	No	list and trace scheduling	11k
[44]	No	list scheduling (VLIW)	18k+10k(architecture)

In the view of flexibility, the methods of verified translation validation are naturally 100% flexible. That means different scheduling algorithms can share the same verified validator, there's no proof cost to change when changing a scheduler, but only the codes of the scheduler itself. Nevertheless, our result shows that in the goal of a fully verified scheduling algorithm, 75% of the proof codes are fixed, both algorithm-independent and machine-independent. Bringing a list scheduler with a different algorithm only influences the remaining proofs. If we treat our list scheduling as some new work that imports our basic theory as a library, the proof-code LOC ratio is around 1:10, approximating the average ratio at the current age of program verification technology.

8.2 View 2: Effect of Optimization

We used a SiFive U74-MC Core Complex as our experimental platform. The U74-MC Core Complex includes four 64-bit U7 RISC-V cores, each of which has a dual-issue, in-order execution pipeline, with a peak execution rate of two instructions per clock cycle. Each U7 core supports standard Multiply, Single-Precision Floating Point, Double-Precision Floating Point, Atomic, Compressed, and Bit Manipulation RISC-V extensions (RV64GCB) [49]. As our experimental benchmark, we used PolyBench C 4.2 [40], a widely used benchmark suite for compiler evaluations. The PolyBench has 30 numerical benchmarks, extracted from a variety of application domains including linear algebra computations, image processing, physics simulation, dynamic programming, statistics, and stencil computations. As the reference implementation, we used the latest version of CompCert 3.13.1 available from the official website [14].

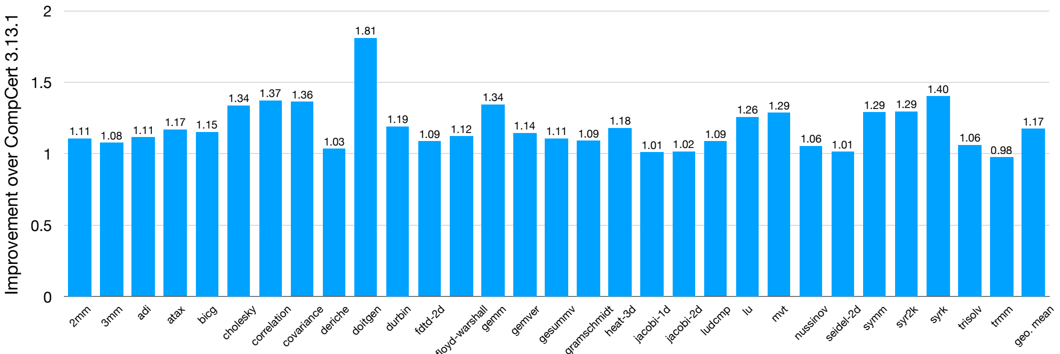


Fig. 7. Performance improvements by the certified instruction scheduler for PolyBench C 4.2

Figure 7 shows the improvement factors of the PolyBench execution performance by our proposed approach, compared to the reference implementation CompCert 3.13.1. To evaluate the impacts of instruction scheduling combined with source-level program optimizations, we applied the PLUTO polyhedral source-to-source compiler [8] to generate the optimized source programs that were used for both the reference version and our version. After the loop transformations including multi-level loop unrolling, the kernel basic blocks (i.e., the innermost loop bodies) have a huge amount of instructions and thereby efficient instruction-level scheduling is the key to enhance the overall execution performance. As shown in Figure 7, our certified instruction scheduler enhanced the performance for most benchmarks, with the geometric mean of 1.17× and up to 1.81× improvements. By comparing the outputs of all 30 benchmarks, we verified that the equivalent outputs were generated: 1) between the reference version and our version; and 2) between enabling and disabling the PLUTO source-to-source compilation.

To see the impact of source-level program optimizations, we also collected the runtime performance numbers when disabling source-to-source polyhedral optimizer. In this case, especially without loop unrolling for parallel loops, the instruction-level parallelism is quite limited for most benchmarks and hence there are few opportunities to reorder instructions. As expected, we could not see notable differences in the scheduling results between the baseline CompCert and our proposed instruction scheduler, with the geometric mean speedup of 1.04.

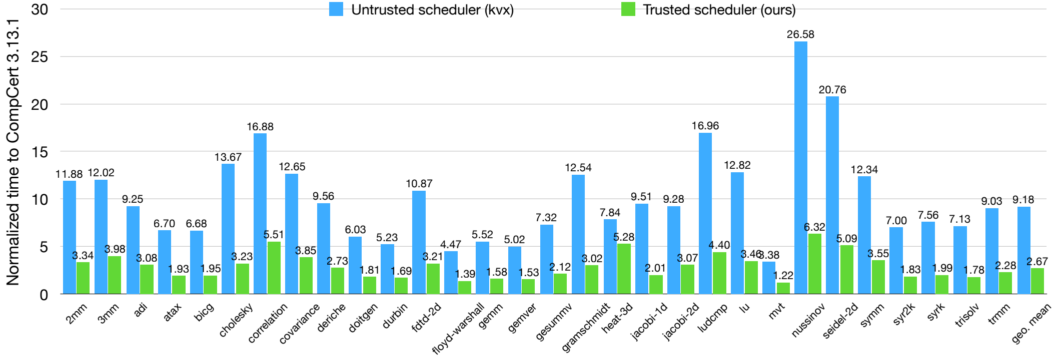


Fig. 8. Normalized compilation times for untrusted KVX scheduler and proposed trusted scheduler, relative to the baseline CompCert 3.13 for PolyBench C 4.2

Figure 8 shows the normalized compilation times to CompCert 3.13.1 for PolyBench kernels using both the proposed certified instruction scheduling pass and the CompCert with scheduler using certified translation validation from [44] (CompCert-KVX). Compared to the original CompCert without the proposed scheduling pass, the normalized compilation times enabling the certified instruction scheduler are between 1.22 (mvt) and 6.32 (nussinov). Our trusted scheduler is also around 3x - 4x faster than CompCert-KVX. We also observed that the absolute compilation times are correlated to the kernel code sizes of benchmarks while there is not strong correlation between the normalized compilation time and kernel code size, i.e. the slowdown of compile time does not depend on the kernel size.

A remark here is CompCert-KVX used different scheduler implementations and heuristics and was implemented directly in OCaml. Nevertheless, in a theoretical view, the compile time overhead of a trusted scheduler in our method has to be strictly less than the untrusted scheduler with the same scheduling pass plus a validating process, whether implemented directly in OCaml or implemented in Coq then extracted to OCaml. Our work just ensures that the validating process can be deleted safely. However, if the untrusted scheduler was implemented in different OCaml codes, it could be possibly faster by using some imperative features, even if it will have extra validating time.

9 Conclusion

9.1 Trusted Computing Base (TCB)

As we mentioned in Section 7, the CompCert project was implemented as a Coq function from C code to assembly code. It will be extracted to OCaml codes to be further compiled into an executable file. The whole verification work only guarantees the correctness of the Coq function. The TCB of the CompCert project trusts the automatic conversion from the Coq functions to the OCaml functions. The TCB of our base theory is the same as the original CompCert. However, in our implementation case using an engineering trick (not mandatory), the implementation of list

scheduling and experiments is slightly different and increased the TCB since the abstract function in Coq was instantiated directly in the OCaml function instead of directly coding it in Coq and it uses OCaml's interface with C language. Doing this adds two more components to the trusted computing base: the reliability of this interface and the convention between Coq natural number and OCaml integers.

As we explained in Section 7 this is not unavoidable but just a better option. We only do it in this way to show how flexible our results can be. It is completely feasible to implement everything of the scheduling heuristics in our case study and experiments in Coq and generate OCaml codes like the original CompCert did, though it requires much more workload for both compiler-backend engineers and proof engineers. Our method makes it possible for compiler engineers to improve the CompCert project without knowing the detailed technology of Coq development.

9.2 Inspiration on Reducing Verification Burden

As a common sense in program verification, the price of a bug-free program is heavy work on proof engineering. In semi-automatic verification methods like Coq proof assistant, the usual idea to reduce the proof workload is using some proof automation tricks. One example are Ltac tactic language embedded in Coq [16, 38].

In the verified translation validation work of [44, 48], the whole scheduling algorithm to be validated can be treated as an uninterpreted parameter since the verified validator will check the correctness result of each translation itself. Similarly, an inspiration we can get from our work is to abstract away components of an algorithm that do not influence correctness but only influence the performance, by abstracting it into a parameter in proof to reduce unnecessary proof code. One example can be the scheduling heuristics during scheduling choice. Another example is the machine architecture: since we implemented the algorithm on intermediate IR that abstracts away machine architecture and only customizes the optimization methods towards architecture in the heuristics, machine details were barely involved in our verification work (only around 40 lines of proof codes which are reusable for different architecture). A similar idea is described in the PhD thesis of [9]

9.3 Future Work

9.3.1 Improved Scheduling Algorithms or Heuristics. The first possible future work is to apply our framework to some advanced efficient (intra-block) instruction scheduling techniques like using profiling [10] and integer programming [51]. Using our framework, those improvements will have little influence on the proof code to further improve optimization performance.

9.3.2 Verified Inter-block Scheduling. Our framework currently only supports optimizations inside a basic block (intra-block scheduling). We believe that similar idea used in our methods can be applied to the correctness of inter-block scheduling. To achieve this, we need to use the concept of program dependence relation that consists of both data dependence and control dependence relations, i.e. constructing the program dependence graph [21, 25].

9.3.3 Verification of Parallelizing Compiler. We wish to stress that this work is not only the first work on fully verified instruction scheduling, but also one of the first step on verification of compiler optimization for multi-level parallelism, a.k.a the first step to bring CompCert to fully verified -O2 and -O3 optimization. Instruction scheduling improves instruction-level parallelism. Besides this, we should also consider the possibility of optimizing a program at data-level and thread-level during compiler time. For example, transforming independent parts of a single-thread program into a multi-thread program like loop parallelization [1].

We believe our result in this work gave a hint on how to improve the parallelism technique of a certified compiler. Current CompCert barely supports optimizations towards parallelly executing a program in either instruction- or task- level.

10 Data Availability

The data and implementation referenced in this paper have been persistently archived [53].

Acknowledgments

We thank Elton Pinto from the University of Pennsylvania for helping us with knowledge of the C interface in OCaml language in our implementation of the list scheduling algorithm. We thank Prithayan Barua from SiFive and Jeffrey Young from Georgia Tech for helping with knowledge and resources related to our experiments on the Risc-V machine. We thank Xiwei Wu from Shanghai Jiao Tong University for inspiration on several mathematical problems related to our formal proofs in Coq.

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under Award Number DE- FOA-0002460: X-Stack: Programming Environments for Scientific Computing.

References

[1] Alexander Aiken and Alexandru Nicolau. 1988. Optimal loop parallelization. *ACM SIGPLAN Notices* 23, 7 (1988), 308–317.

[2] Andrew W Appel. 2014. *Program logics for certified compilers.* Cambridge University Press.

[3] Andrew W. Appel. 2023. *Verified Functional Algorithms.* Software Foundations, Vol. 3. Electronic textbook. http://softwarefoundations.cis.upenn.edu

[4] David Bernstein and Michael Rodeh. 1991. Global instruction scheduling for superscalar machines. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation.* 241–255.

[5] D. Bernstein, M. Rodeh, and I. Gertner. 1989. On the complexity of scheduling problems for parallel/pipelined machines. *IEEE Trans. Comput.* 38, 9 (1989), 1308–1313. https://doi.org/10.1109/12.29469

[6] Frédéric Besson, Sandrine Blazy, and Pierre Wilke. 2019. CompCertS: a memory-aware verified C compiler using a pointer as integer semantics. *Journal of Automated Reasoning* 63 (2019), 369–392.

[7] Sandrine Blazy, Benoît Robillard, and Andrew W. Appel. 2010. Formal Verification of Coalescing Graph-Coloring Register Allocation. In *Programming Languages and Systems*, Andrew D. Gordon (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 145–164.

[8] Uday Bondhugula, Aravind Acharya, and Albert Cohen. 2016. The Pluto+ Algorithm: A Practical Approach for Parallelization and Locality Optimization of Affine Loop Nests. *ACM Trans. Program. Lang. Syst.* 38, 3, Article 12 (April 2016), 32 pages. https://doi.org/10.1145/2896389

[9] Sylvain Boulmé. 2021. *Formally Verified Defensive Programming (efficient Coq-verified computations from untrusted ML oracles).* Habilitation à diriger des recherches. Université Grenoble-Alpes. https://hal.science/tel-03356701 See also http://www-verimag.imag.fr/ boulme/hdr.html.

[10] William Y. Chen, Scott A. Mahlke, Nancy J. Warter, Sadun Anik, and Wen-Mei W. Hwu. 1994. Profile-assisted instruction scheduling. *International Journal of Parallel Programming* (1994). https://doi.org/10.1007/BF02577873

[11] Hong-Chich Chou and Chung-Ping Chung. 1995. An optimal instruction scheduler for superscalar processor. *IEEE Transactions on Parallel and Distributed Systems* 6, 3 (1995), 303–313. https://doi.org/10.1109/71.372778

[12] Edward G. Coffman and John Bruno. 1976. Computer and job-shop scheduling theory. https://api.semanticscholar.org/CorpusID:60396080

[13] R. Collins and G.B. Steven. 1996. Instruction scheduling for a superscalar architecture. In *Proceedings of EUROMICRO 96. 22nd Euromicro Conference. Beyond 2000: Hardware and Software Design Strategies.* 643–650. https://doi.org/10.1109/EURMIC.1996.546492

[14] CompCert web 2023. COMPCERT: COMPILERS YOU CAN FORMALLY TRUST. https://compcert.org.

[15] Cortex-A53 2012. Cortex-A53. https://developer.arm.com/Processors/Cortex-A53.

[16] David Delahaye. 2000. A tactic language for the system Coq. In *Logic for Programming and Automated Reasoning: 7th International Conference, LPAR 2000 Reunion Island, France, November 6–10, 2000 Proceedings 7.* Springer, 85–95.

[17] J R Ellis. 1985. Bulldog: a compiler for VLIW architectures. (1 1985). https://www.osti.gov/biblio/5724953

[18] John R Ellis. 1986. *Bulldog: a compiler for VLSI architectures.* Mit Press.
[19] Paolo Faraboschi, Joseph A Fisher, and Cliff Young. 2001. Instruction scheduling for instruction level parallel processors. *Proc. IEEE* 89, 11 (2001), 1638–1659.
[20] Paul Feautrier. 1991. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming* (1991). https://doi.org/10.1007/BF01407931
[21] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. 1987. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9, 3 (1987), 319–349.
[22] Fisher. 1981. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Trans. Comput.* C-30, 7 (1981), 478–490. https://doi.org/10.1109/TC.1981.1675827
[23] Philip B Gibbons and Steven S Muchnick. 1986. Efficient instruction scheduling for a pipelined architecture. In *Proceedings of the 1986 SIGPLAN symposium on Compiler construction.* 11–16.
[24] Léo Gourdin, Benjamin Bonneau, Sylvain Boulmé, David Monniaux, and Alexandre Bérard. 2023. Formally Verifying Optimizations with Block Simulations. *Proceedings of the ACM on Programming Languages* 7, OOPSLA2 (2023), 59–88.
[25] Mary Jean Harrold, Brian Malloy, and Gregg Rothermel. 1993. Efficient construction of program dependence graphs. *ACM SIGSOFT Software Engineering Notes* 18, 3 (1993), 160–170.
[26] Hanru Jiang, Hongjin Liang, Siyang Xiao, Junpeng Zha, and Xinyu Feng. 2019. Towards Certified Separate Compilation for Concurrent Programs. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019).* Association for Computing Machinery, New York, NY, USA, 111–125. https://doi.org/10.1145/3314221.3314595
[27] Jeehoon Kang, Yoonseung Kim, Chung-Kil Hur, Derek Dreyer, and Viktor Vafeiadis. 2016. Lightweight verification of separate compilation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* 178–190.
[28] Hironori Kasahara and Seinosuke Narita. 1984. Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing. *IEEE Trans. Comput.* C-33, 11 (1984), 1023–1029. https://doi.org/10.1109/TC.1984.1676376
[29] V. L. Kompel'makher and V. A. Liskovets. 1975. Sequential generation of arrangements by means of a basis of transpositions. *Cybernetics* (1975). https://doi.org/10.1007/BF01069459
[30] M. Lam. 1988. Software pipelining: an effective scheduling technique for VLIW machines. *SIGPLAN Not.* 23, 7 (jun 1988), 318–328. https://doi.org/10.1145/960116.54022
[31] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115.
[32] Xavier Leroy. 2009. A formally verified compiler back-end. *Journal of Automated Reasoning* 43 (2009), 363–446.
[33] Hongjin Liang and Xinyu Feng. 2016. A program logic for concurrent objects under fair scheduling. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* 385–399.
[34] Anil Madhavapeddy and Yaron Minsky. 2022. *Real World OCaml: Functional Programming for the Masses.* Cambridge University Press.
[35] Eric Mullen, Daryl Zuniga, Zachary Tatlock, and Dan Grossman. 2016. Verified peephole optimizations for CompCert. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation.* 448–461.
[36] George C Necula. 2000. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation.* 83–94.
[37] Daniel Patterson and Amal Ahmed. 2019. The next 700 compiler correctness theorems (functional pearl). *Proceedings of the ACM on Programming Languages* 3, ICFP (2019), 1–29.
[38] Pierre-Marie Pédrot. 2019. Ltac2: tactical warfare. In *The Fifth International Workshop on Coq for Programming Languages, CoqPL*, Vol. 2019.
[39] Amir Pnueli, Michael Siegel, and Eli Singerman. 1998. Translation validation. In *Tools and Algorithms for the Construction and Analysis of Systems: 4th International Conference, TACAS'98 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS'98 Lisbon, Portugal, March 28–April 4, 1998 Proceedings 4.* Springer, 151–166.
[40] PolyBench. 2011. The Polyhedral Benchmark Suite. http://www.cse.ohio-state.edu/~pouchet/software/polybench/.
[41] B. Ramakrishna Rau, Christopher D. Glaeser, and Raymond L. Picard. 1982. Efficient code generation for horizontal architectures: Compiler techniques and architectural support. In *Proceedings of the 9th Annual Symposium on Computer Architecture* (Austin, Texas, USA) *(ISCA '82).* IEEE Computer Society Press, Washington, DC, USA, 131–139.
[42] Silvain Rideau and Xavier Leroy. 2010. Validating Register Allocation and Spilling. *CC* 6011 (2010), 224–243.
[43] Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. 2013. CompCertTSO: A verified compiler for relaxed-memory concurrency. *Journal of the ACM (JACM)* 60, 3 (2013), 1–50.
[44] Cyril Six, Sylvain Boulmé, and David Monniaux. 2020. Certified and Efficient Instruction Scheduling: Application to Interlocked VLIW Processors. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 129 (nov 2020), 29 pages. https://doi.org/10.1145/3428197
[45] Cyril Six, Léo Gourdin, Sylvain Boulmé, David Monniaux, Justus Fasse, and Nicolas Nardino. 2022. Formally verified superblock scheduling. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs*

and Proofs (Philadelphia, PA, USA) *(CPP 2022)*. Association for Computing Machinery, New York, NY, USA, 40–54. https://doi.org/10.1145/3497775.3503679

[46] Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W Appel. 2015. Compositional compcert. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 275–287.

[47] The OCaml Manual, chapter 22 2023. https://v2.ocaml.org/manual/intfc.html#c%3Aintf-c

[48] Jean-Baptiste Tristan and Xavier Leroy. 2008. Formal Verification of Translation Validators: A Case Study on Instruction Scheduling Optimizations. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '08)*. Association for Computing Machinery, New York, NY, USA, 17–27. https://doi.org/10.1145/1328438.1328444

[49] U74MC 2021. SiFive U74-MC Core Complex Manual 21G1.01.00. https://starfivetech.com/uploads/u74mc_core_complex_manual_21G1.pdf.

[50] Yuting Wang, Ling Zhang, Zhong Shao, and Jérémie Koenig. 2022. Verified Compilation of C Programs with a Nominal Memory Model. *Proc. ACM Program. Lang.* 6, POPL, Article 25 (jan 2022), 31 pages. https://doi.org/10.1145/3498686

[51] Kent Wilken, Jack Liu, and Mark Heffernan. 2000. Optimal instruction scheduling using integer programming. *Acm sigplan notices* 35, 5 (2000), 121–133.

[52] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. 283–294.

[53] Ziteng Yang, Jun Shirako, and Vivek Sarkar. 2024. Artifact for paper "Fully Verified Instruction Scheduling". https://doi.org/10.5281/zenodo.13625830.

[54] Qirun Zhang, Chengnian Sun, and Zhendong Su. 2017. Skeletal Program Enumeration for Rigorous Compiler Testing. *SIGPLAN Not.* 52, 6 (jun 2017), 347–361. https://doi.org/10.1145/3140587.3062379

Received 2024-04-06; accepted 2024-08-18