Formally Verified Superblock Scheduling

Cyril Six, Léo Gourdin, Sylvain Boulmé, David Monniaux, Justus Fasse, Nicolas Nardino

Verified Superblock Scheduling with Related Optimizations

CYRIL SIX, Kalray S.A., France and Univ. Grenoble Alpes, CNRS, Grenoble INP, Verimag, France
LÉO GOURDIN, Univ. Grenoble Alpes, CNRS, Grenoble INP, Verimag, France
SYLVAIN BOULMÉ, Univ. Grenoble Alpes, CNRS, Grenoble INP, Verimag, France
DAVID MONNIAUX, Univ. Grenoble Alpes, CNRS, Grenoble INP, Verimag, France

Necula [2000] and Tristan et al. [2011] established that *symbolic execution* combined with *rewriting* is effective in *validating* the code produced by state-of-the-art compilers applying various optimizations. In the meantime, Tristan and Leroy [2008] used *formally-verified* symbolic execution to *certify* the schedules produced by untrusted oracles—optimizing pipeline usage—within the CompCert compiler. Alas, their formally-verified checker had exponential complexity and was thus never integrated into mainline CompCert. Recently, Six et al. [2020] solved this performance issue with formally-verified hash-consing within the symbolic execution. Our paper extends these approaches to superblocks (where instructions move across branches) and enables translation validation of instruction rewritings (changing instructions for "better" ones) modulo register liveness (e.g. with introduction of "fresh" registers): a significant step forward for *certifying* compilers from symbolic executions.

1 INTRODUCTION

In-order processor cores execute assembly instructions in the order in which they are in the program. If one instruction computes a register and the next instruction uses this register, then the core may *stall* until the value computed becomes available, which may take a number of clock cycles for expensive instructions. An optimizing compiler will thus reorder instructions to minimize stalling.[1]

CompCert[2] is a compiler for the C programming language[3] with a machine-checked proof of correctness: if compilation succeeds, then the semantics of the assembly code match that of the source code in the sense that an execution of the C program without undefined behaviors translates into an assembly execution with the same sequence of observable events (calls to external functions, accesses to volatile variables…). Vanilla CompCert however does not reschedule instructions, and thus produces suboptimal assembly code for in-order cores.

Tristan and Leroy [2008]; Tristan [2009] added instruction scheduling to CompCert: an untrusted oracle would reorder instructions, and some formally verified checker would witness that the semantics were preserved through *symbolic execution*. But, their checker was unusable because of its exponential runtime complexity [Tristan 2009, §6.7.1][Tristan and Leroy 2008, §7].

Six et al. [2020] added instruction scheduling and some peephole optimizations to CompCert at the assembly level, in *postpass* (after register allocation and final transformations, see Figures 1 and 2). One of their goals was to form instruction "bundles", to be executed in parallel, for a VLIW (very large instruction word) processor, called KVX. However, due to that late position in the compilation process, they were limited to scheduling inside basic blocks (instruction sequences with one single entry point and one single exit point). Consequently, their system was sometimes prevented from finding good schedules by dependencies induced by register reuse. Moreover, the

[1] The alternative is using an *out of order* core, which dynamically reorders instructions. Such cores are more complex and less predictable (e.g., for bounding worst-case execution time), two undesirable characteristics for some safety-critical systems.
[2] https://compcert.org/ vanilla releases at https://github.com/absint/CompCert
[3] There also exists a front-end for a subset of the Lustre synchronous data-flow programming language [Bourke et al. 2017].

Authors' addresses: Cyril Six, Kalray S.A., 180 avenue de l'Europe, Montbonnot, 38330, France, Cyril.Six@kalray.eu, Univ. Grenoble Alpes, CNRS, Grenoble INP, Verimag, 38401, France, Cyril.Six@univ-grenoble-alpes.fr; Léo Gourdin, Univ. Grenoble Alpes, CNRS, Grenoble INP, Verimag, 38401, France, Leo.Gourdin@univ-grenoble-alpes.fr; Sylvain Boulmé, Univ. Grenoble Alpes, CNRS, Grenoble INP, Verimag, 38401, France, Sylvain.Boulme@univ-grenoble-alpes.fr; David Monniaux, Univ. Grenoble Alpes, CNRS, Grenoble INP, Verimag, 38401, France, David.Monniaux@univ-grenoble-alpes.fr.

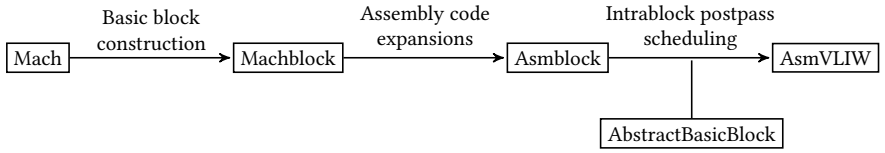

Fig. 1. The KVX backend of Six et al. [2020]

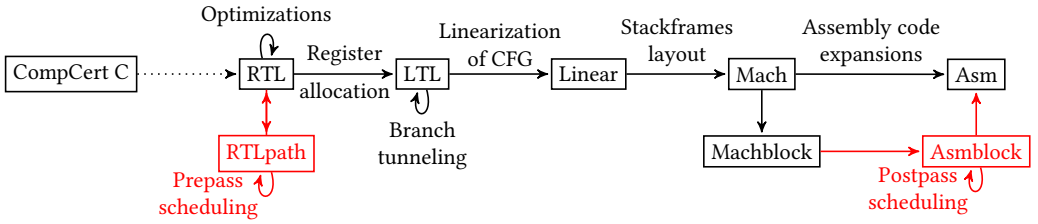

Fig. 2. Extending CompCert Backends with our Scheduling Passes

porting effort of their approach is rather high for every architecture. Section 2 briefly recalls their approach and how **we have ported it to AArch64 architecture, as a secondary contribution**. In particular, it enables a "cheap" verification of some peephole optimizations (e.g. replacing loads and stores to consecutive addresses by double loads and stores) on the final assembly code.

Our main contribution is a verified instruction scheduler and rewriter working on a portable intermediate representation (RTL, see Figure 2). It operates over *superblocks*: a generalization of basic blocks, such that each node of a given block has still at most one successor in this block, but may also branch to another (super)block [Hwu et al. 1993; Lee et al. 1993]. The semantics of the scheduled superblock must preserve the observable outputs on live registers of the non-trapping executions of the original superblock. Undefined behaviors (e.g., traps such as division by zero or incorrect memory accesses) may be preserved or replaced by defined behaviors. For example, **the scheduler may move instructions across some internal conditional branches** as long as this is not observable by other superblocks; it may also introduce fresh registers (e.g. local renamings), or **replace some instructions by equivalent combinations**. See Section 3.

The scheduled superblock must *simulate* the original superblock. We check this simulation property with a formally-proved *simulation test* described in Section 5. Similar to the postpass verifier of Six et al. [2020], this test is based on symbolic execution with formally-verified *hash-consing*. Like them, the only addition we make to CompCert's trusted computing base is that pointer equality implies structural equality, which is non-controversial.[4] Our symbolic execution also normalizes symbolic values in order to validate some rewritings of instructions during the scheduling. Indeed, our untrusted schedulers rewrite or expand some instructions (depending on the target architecture) in a lightweight way compared to the use of a postpass oracle, that is meant for increasing scheduling opportunities.

Prior to scheduling, a preliminary phase selects a superblock structure for each function [Hwu et al. 1993; Lee et al. 1993]. Here, many choices are possible, which have a deep impact on the overall performance, since intra-superblock scheduling amounts to optimizing some execution path at the expense of other execution paths. First, there are often several partitions of a given function block into superblocks, provided that we do not try to create maximal superblocks. Moreover,

[4]Also, at some point, we treat a possibly nondeterministic function as deterministic, to avoid having to make very expensive changes to the rest of CompCert. This is in line with vanilla CompCert using untrusted, possibly nondeterministic, oracles, as though they were deterministic.

during the selection of superblocks, we may duplicate some instructions (tail-duplication, several variations of loop unrolling) in order to create new opportunities of superblock partitioning with larger superblocks at the end. Currently, our superblocks are selected to be the "most likely path" inside the code, and instruction duplications are controlled through compiler options. See Section 4.

2 ADAPTING THE CERTIFIED KVX POSTPASS-SCHEDULER FOR AARCH64

Section 2.1 recalls the formally-verified postpass scheduler of Six et al. [2020] for the Kalray KVX processor. Section 2.2 briefly reports on our port of their system to the AArch64 architecture.

2.1 KVX Postpass-Scheduler of Six et al. [2020]

The certified compiler of Six et al. [2020] extends the usual assembly expansions of vanilla CompCert—the "Mach-to-Asm" pass of Figure 2—with instruction scheduling within basic blocks: the "Mach-to-AsmVLIW" passes of Figure 1. Because the target processor (Kalray KVX) is a VLIW (with explicit parallelism of instructions at the assembly-level), they need to introduce a new kind of assembly language in CompCert—called AsmVLIW—providing an abstract syntax and a parallel semantics for "bundles" of instructions. They have also introduced 3 new intermediate representations. First, Asmblock, an IR specific for the KVX, which shares the syntax and the semantics of atomic instructions with AsmVLIW, but provides a blockstep sequential semantics for "basic blocks" of atomic instructions. Their other IR, called Machblock and AbstractBasicBlock, are generic w.r.t. the target processor (like Mach). Machblock extends Mach with a basic block semantics, and the "Mach-to-Machblock" pass provides a generic pass for recovering the basic block structure from Mach programs. AbstractBasicBlock is the input language for their generic certified static analyzes over basic blocks. Note that because they target a VLIW processor, their scheduling also performs "bundling".

Scheduling is performed block by block from the Asmblock program. As depicted in Fig. 3, it generates a list `lb` of AsmVLIW bundles from each basic block `B`. More precisely, a basic block `B` from Asmblock enters the PostpassScheduling module. This module sends `B` to an external untrusted scheduler, which returns a list of bundles `lb`, candidates to be added to the AsmVLIW program. The PostpassScheduling module then checks that `lb` simulates `B` and through dedicated (certified) verifiers. Then, PostpassScheduling either adds `lb` to the AsmVLIW program, or stops the compilation if the verifiers returned an error.

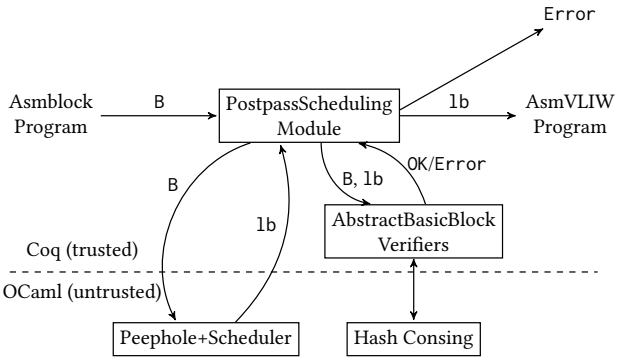

Fig. 3. Certified Scheduling from Untrusted Oracles

The dynamic verification that `lb` simulates `B` is done by composing two independent verifications, after generating an intermediate basic block `tB`, obtained by "sequentializing" the atomic instructions of `lb` into a single basic block: (1) `tB` *simulates* `B` for the sequential semantics of basic block; (2) for each bundle `b` of `lb`, the sequential semantics and the parallel semantics of `b` are *observationally equivalent*.

Each test is implemented on the AbstractBasicBlock IR, after a compilation pass "Asmblock-to-AbstractBasicBlock" that preserves both the sequential and the parallel semantics. In the following,

we focus on the simulation test—test 1) above—because, AArch64 is not a VLIW processor and there is no parallel semantics to consider.

2.1.1 Simulation Test of Six et al. [2020]. The simulation test on the AbstractBasicBlock IR uses symbolic execution, which simply consists in compiling each program into a big symbolic term—actually called a *symbolic state*—in order to deduce the simulation from syntactical equalities on symbolic states. In the case of a basic block—with a single execution path—such a symbolic state simply corresponds to a kind of *preconditioned parallel assignment*, as illustrated on Example 2.1.

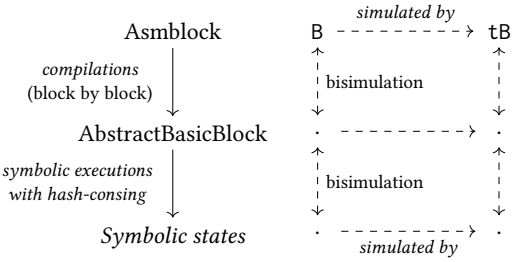

Fig. 4. Diagram of Simulation Test Correctness

Example 2.1 (Simulation on symbolic states). Consider two basic blocks B_1 and B_2:

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

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

Both B_1 and B_2 lead to the same parallel assignment: $r_1 := (r_1 + r_2) + (r_1 + r_2) \parallel r_3 := r_1 + r_2$. But, normal execution of B_1 is preconditioned by "$\texttt{load}[m, r_1 + r_2]$ has not trapped", whereas the precondition of B_2 is trivially true. Hence B_2 simulates B_1, but the converse is false.

Six et al. [2020] encode such a precondition as a list of potentially trapping terms, hence relaxing the implication of preconditions as a list inclusion.

As coined by King [1976], symbolic execution mimics the concrete semantics of programs, while replacing each concrete state by a symbolic state (which thus represents the set of all reachable concrete states). Finally, the overall proof of the simulation of `B` by `tB` corresponds to compose the two commutative diagrams on the right-hand side of Figure 4.

2.1.2 Certified Hash-Consing. As suggested by duplication of term "$r_1 + r_2$" in Example 2.1, symbolic execution involves many replicas of terms. Thus, comparing symbolic states with structural equalities of terms, as performed in [Tristan and Leroy 2008; Tristan 2009; Tristan and Leroy 2010], takes exponential time (and prevents their simulation tests to be usable).

Instead, Six et al. [2020] establish term equality by pointer equality, which assumes that two identical terms lie at the same location in memory. This is achieved by *hash-consing*: when constructing a term, their system looks it up in an *unstrusted* hash table that contains all terms generated by the symbolic execution, and returns the existing instance of that term if there is one.

This "smart constructor" for terms is wrapped into a formally proved function as follows: the term $c(a_1, \ldots, a_n)$ returned by the untrusted hash table, where c is the constructor at the root and $a_1, \ldots, a_n$ are its operands, is formally accepted as equal to the term that we want to create $c'(a'_1, \ldots, a'_n)$ only if $c = c'$ and all $a_i == a'_i$ by pointer equality.

This does not allow proving that if two terms are equal, then their pointers are equal, though this is what happens. But they only need the other direction: if pointers are equal then so are the terms. However, because modeling pointer equality as a pure function is unsound, they model pointer equality as a nondeterministic operator, inside a nondeterministic monad and they admit the axiom that pointer equality implies term equality (see [Six et al. 2020, Sect. 4.4] for details).

2.1.3 Peephole Optimization on the KVX. In the preprocessing of their untrusted scheduling oracle, Six et al. [2020] perform a small peephole optimization: they replace pairs of simple load/store

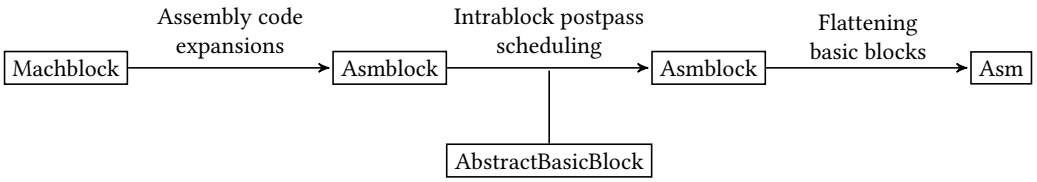

Fig. 5. Our AArch64 backend

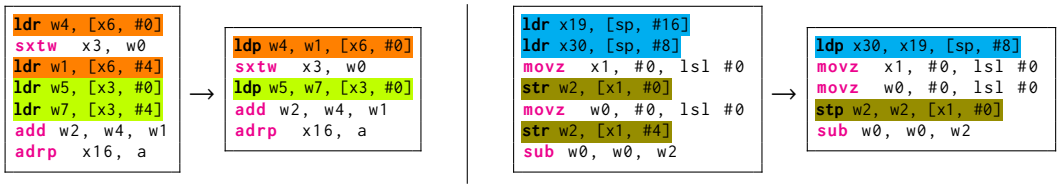

Fig. 6. Four Examples of Load/Store Compaction on AArch64

instructions to consecutive addresses by single double load/store instructions. In the formally-verified simulation test, this optimization is also expressed as a rewriting, of the Asmblock-to-AbstractBasicBlock pass (see Fig. 4). A noticeable feature of their approach comes from the fact that the rewriting rules are expressed in the verifier in the *opposite way* to the oracle. Indeed, in the verifier, each double load/store is rewritten into a pair of simple loads/stores. This way induces a *much simpler proof*, especially for our AArch64 backend, where the replaced simple load/store instructions are not necessarily consecutive within the original basic-block (hence our AArch64 peephole preprocessing performs itself instruction reordering), as illustrated in Figure 6.

2.2 Our Port of the Postpass-Scheduler on AArch64

Vanilla CompCert provides a backend for the AArch64 processor (a non-VLIW processor) and thus provides us the "Mach-to-Asm" pass of Figure 2. We have replaced this pass by the "Mach-to-Asm" passes resulting from the composition of the "Mach-to-Asmblock" of Six et al. [2020] with the passes of "Asmblock-to-Asm" of Figure 5. We have kept AArch64 Asm mostly unchanged, and have introduced a new IR Asmblock for AArch64, providing a basic block syntax and semantics over Asm instructions. The original "Mach-to-Asm" AArch64 pass has been extended into a "Machblock-to-Asmblock" pass by also adapting the 6 Kloc corresponding proof on the KVX (representing a development of 5 Kloc of Coq in one man·month for AArch64). The final pass of "Asmblock-to-Asm" is quite bureaucratic but its proof is still a bit more tedious than informally expected[5]: 2 Kloc of Coq. We also implemented the "Asmblock-to-AbstractBasicBlock" pass necessary to reuse the simulation test of AbstractBasicBlock to check the correctness of our scheduling oracle.

In contrast to the peephole optimizer of [Six et al. 2020], ours (also applied before the untrusted scheduler), is able to merge non-consecutive load or store within the original basic block, as long as they respect the semantic dependencies and offset constraints on double load/store specific to AArch64 ISA. Our algorithm traverses the basic block in both directions, while remembering every encountered compatible load and store as potentials candidates (and forgetting them if another instruction breaks a needed dependency in-between). The first pass (forward) tries to replace the last encountered load or store by the double instruction, and the first one by a `Nop` (no operation)

[5]The translation, unfolding the basic block structure, is completely straightforward. But, its correctness proof, as a "Plus" simulation, is a bit tedious. For instance, this "Plus" simulation results from the composition of either a "Plus" and then a "Star" simulation, or a "Star" and then a "Plus", depending or not on the absence or not of a control-instruction at the end of the input basic block. This case analysis requires the introduction of several "similar" intermediate lemma.

instruction. The second pass (backward) tries the opposite. Figure 6 illustrates four situations found by our peephole. On the left column: (1) backward load pairing, with increasing offset (the offset of the second load is *greater* than that of the first one); (2) consecutive load pairing, with increasing offset. On the right column: (1) consecutive load pairing, with decreasing offset (the offset of the second load is *lower* than that of the first one); (2) forward store pairing, with increasing offset.

Currently, the main benefit of our peephole for AArch64 is a reduction of code size: it reduces the number of generated memory transfer instructions by about 10%, which represents approximately 3% of the total code length (on average across all our benchmarks). Unfortunately, on our Cortex-A53 target, it does not speedup much the generated code, as if, on this dual-issue processor, double memory accesses were actually performed by two accesses. But, this optimization opens the door for future similar replacements: e.g. selection of `ands` or the `bics` instructions[6].

Like Six et al. [2020], our formally-verified simulation test validates these rewritings by performing the reverse rewriting (i.e. from double load/store to pairs of simple load/store) in the Asmblock-to-AbstractBasicBlock pass (see Fig. 4). The overall implementation of our formally-verified postpass scheduler on AArch64 represents a bit more than three man·months of development.

3 OUR SUPERBLOCK SCHEDULING MODULO LIVENESS AND REWRITINGS

With respect to the "intra basic-block" scheduling of Six et al. [2020], our prepass scheduling brings two main improvements: (1) the scheduler operates on superblocks (one entry point, possibly several exit points), and instructions may be moved across intermediate conditional exits; (2) instructions can be rewritten during the scheduling including assignments to fresh registers.

In turn, our formally-verified simulation test must support features not provided by Six et al. [2020]: (1) conditional exits (e.g. several execution paths within symbolic execution); (2) simulation modulo liveness of registers (on each possible exit); (3) normalization of symbolic values during symbolic execution. Sections 3.1 and 3.2 illustrate this on two different processors.

3.1 KVX Speculative Loads in Superblock Scheduling

In addition to normal load instructions that trap (usually aborting the program) if an incorrect address is accessed, the Kalray KVX provides special load instructions known as "speculative", "dismissible" or "non-trapping", which instead return a default value.[7] Speculative loads can be freely moved before a conditional branch, whereas normal loads cannot unless one can prove that it cannot trap; they are thus very interesting for superblock scheduling.

Formally, it is correct to compile C programs entirely using speculative loads, since access to incorrect addresses is undefined behavior, and returning a default value is an acceptable way of implementing undefined behavior. Yet, this would hinder debugging and detection of abnormal behavior. We thus opted to generate speculative loads only as needed by the schedule.

Consider for instance the superblock starting at label `.L100` of Figure 7: it is the body of a loop exiting on label `.L101` that computes in `$r0` the sum of the `$r1` integer array for index variable `$r4` (bounded by `$r2`). On the left, the superblock has been scheduled and bundled with the postpass scheduler of Six et al. [2020]. On the right, our prepass scheduler has moved `sxwd` (originally on line 6) and `lws.xs` (originally on line 9) above the conditional exit originally on line 4. The effect of these moves is to gain one bundle and to remove one pipeline stall one the update of `$r0`.[8] The gain is of $2 \cdot n - 1$ cycles where n is the number of loop iteration (there is 1 cycle loss if there is no

[6] Specific versions of the corresponding arithmetic instructions update the condition flags while writing the result.

[7] If paging, through a memory management unit, is not used, speculative loads are easily implemented in hardware: just return the default value instead of generating an exception. If paging is used, their implementation needs some cooperation from the virtual memory subsystem.

[8] Only one stall remains in the improved scheduling, because loads have latency of 3 cycles on the KVX.

```
1  L100:
2   compw.ge $r32 = $r4, $r2
3  ;;
4   cb.wnez $r32? .L101
5  ;;
6   sxwd $r5 = $r4
7   addw $r4 = $r4, 1
8  ;;
9   lws.xs $r3 = $r5[$r1]
10 ;; // 2 STALLS
11  addw $r0 = $r0, $r3
12  goto .L100
```

```
.L100:                                  1
 sxwd $r5 = $r4                         2
 compw.ge $r32 = $r4, $r2               3
;;                                      4
 lws.s.xs $r3 = $r5[$r1]                5
;;                                      6
 cb.wnez $r32? .L101                    7
;; // 1 STALL                           8
 addw $r0 = $r0, $r3                    9
 addw $r4 = $r4, 1                      10
 goto .L100                             11
```

Fig. 7. On the left: a sequence of KVX bundles as emitted by Six et al. [2020] (makespan of 8 cycles). On the right, the result with our prepass scheduler in between (makespan of 6 cycles).

iteration, because the two moved instructions have been executed while being useless; `sxwd` has been executed in parallel of `compw.ge`, thus its useless execution does not lose a cycle). For this simple loop, this is almost a gain of 25%.

Note that, for Six et al. [2020], these moves were impossible because the original superblock is made of two basic blocks, the first one ended on the conditional exit of line 4. In order to prove that the second superblock simulates the first one, we need to check that the assignment of `$r5` and `$r3` involved in these moved instructions have no effect on the code after the loop exit (at label `.L101`). Fortunately, they are not in the live registers of `.L101` (they are "local" to the loop body). Moreover, we need to check that these moved instructions do not introduce any undefined behavior when label `.L101` should be taken. Actually, this is the case, because the scheduler has rewritten the trapping load `lws.xs` into a non-trapping (speculative) load `lws.s.xs`.

3.2 Expanding Instructions with Immediate on RISC-V in Superblock Scheduling

Figure 8 presents a fragment of C code and the resulting RiscV superblock, both for vanilla CompCert and for our CompCert version. Registers `x10`, `x11` and `x12` respectively correspond to variables `x`, `y` and `t` of the input program. Vanilla CompCert does not attempt to minimize pipeline stalls: on line 1, the `lw` instruction dereferencing `x12` in `x7` may induce pipeline stalls in line 3, where `x7` is added to `x10` in `x6`. Moreover, vanilla CompCert expands the comparison with immediate only in the "Mach-to-Asm" pass (Figure 2): the immediate (here 7) is stored in the scratch register `x31` (in RiscV, `x0` is a read-only register equal to 0). Additionally, vanilla CompCert does not attempt to remember that from line 4, register `x31` has value 7: thus it reloads 7 in `x31` a second time on line 6.

On RiscV, our prepass scheduler performs an expansion of comparisons with immediate (branching or not), and of some others instructions (arithmetic operations on immediate, casts, loads of constant, and length conversions) in the untrusted preprocessor of our scheduling oracle. Intermediate values coming from expansion are stored into fresh pseudo-registers (before the register allocation pass), and the untrusted preprocessor uses a dynamic value numbering system to avoid redundant instructions (it is a Common Subexpression Elimination limited to the superblock - operating on every instruction in the path). Fortunately, in Figure 8, the fresh pseudo-register assigned to immediate 7, has been allocated (after the prepass scheduling) into `x12` (register reuse). In the general case, because the scope of our preprocessing is limited to a superblock, this memoization of immediate should only have a limited impact on register pressure.

Performing these expansions within the prepass scheduling increases scheduling opportunities. Here, the assignment of `x12` to 7 is interleaved by the scheduler between the load of `0(x12)` into `x7`, and the addition of `x7` to `x10`: this potentially saves one cycle.

```
if (x + *t < 7)
  if (y < 7)
    return 421;
```

```
1  lw      x7, 0(x12)
2  // POTENTIAL STALL for x7
3  addw    x6, x10, x7
4  addiw   x31, x0, 7
5  bge     x6, x31, .L100
6  addiw   x31, x0, 7
7  bge     x11, x31, .L100
```

```
lw      x7, 0(x12)
addiw   x12, x0, 7
addw    x6, x10, x7
bge     x6, x12, .L100
bge     x11, x12, .L100
```

Fig. 8. On the left: a fragment of a C function; in the middle: the RiscV assembly produced by vanilla CompCert (3.8); on the right: the RiscV assembly produced with our prepass scheduler (one `addiw` & one potential `lw` stall have been gained, resulting in a potential gain of two cycles).

Our formally-verified simulation test must check that the expansions performed by the untrusted scheduling preprocessor simulate the original RTL superblock. This is achieved by applying rewriting rules mimicking those of the preprocessor within the symbolic execution and formally proving than these rewritings preserve the semantics of symbolic values. It would be difficult to directly prove the memoization mechanism within the preprocessor, whereas it is proved "for free" by our simulation test with symbolic execution. Even without memoization, verifying these rewriting rules on symbolic values is much easier than verifying them directly on the RTL code. Indeed, symbolic values are directly expression trees, whereas the RTL code is a CFG of register assignments. In particular, the rewriting rules on symbolic values do not involve registers (and substitution of registers). The verification that the untrusted rewritings of the untrusted preprocessor correctly deal with "fresh" registers is only a particular case of the simulation test modulo liveness: the expansion on the right-hand side of Figure 8 is correct because `x12` is not live at labels `.L100` and `.L101`. For example, let us consider the following rewriting on RTL conditional branch instructions:

$$L_1\colon\ \texttt{if (Cges 7)}[r_1]\ \texttt{goto}\ L_2\ \texttt{else goto}\ L_3\ \longrightarrow\ \begin{cases} L_1\colon\ r_2 := \texttt{(OEaddiwr0 7)};\ \texttt{goto}\ L_4 \\ L_4\colon\ \texttt{if CEbgew}[r_2, r_1]\ \texttt{goto}\ L_2\ \texttt{else goto}\ L_3 \end{cases}$$

where r_2 is a fresh pseudo-register and L_4 is a fresh node. This rewriting is simply expressed in the symbolic representation by the rule "$\texttt{(Cges 7)}[v]\ \longrightarrow\ \texttt{CEbgew}[\texttt{(OEaddiwr0 7)}, v]$" where v is the symbolic value of r_1.

Note that embedding the rewriting rules within the symbolic execution makes the proof of these rewrite rules easier than having them expressed in a preprocessing of the simulation test (e.g. in the "Asmblock-to-AbstractBasicBlock" pass of Fig. 4 of [Six et al. 2020]). Within the symbolic execution, the proof ignores liveness issues. In a preprocessing of the simulation test, rewriting rules would be required to satisfy a bisimulation modulo register liveness.

4 SELECTION OF "RELEVANT" SUPERBLOCKS FOR INTRA-BLOCK OPTIMIZATIONS

Superblock scheduling [Hwu et al. 1993; Lee et al. 1993] optimizes some given execution path of the program, by moving instructions before conditional branches (or after them, at the discretion of the scheduler). If optimized paths are frequently taken, the function execution will be faster. However, this may be to the detriment of other paths: moving the `lws` instruction above the loop-exit in Figure 7 actually slows down the path exiting from the loop (but eliminates a stall at each loop iteration).

Superblocks must thus be carefully chosen. This section describes how we transform each function into a partition of superblocks. The first step to form superblocks is to identify *CFG paths*—what is called *trace*[9] by Fisher [1981]—likely to be executed. This is static *branch prediction*.

[9]CompCert already defines a *trace* as a sequence of observational events. Instead, we use "CFG path".

Then, based on this prediction, we may duplicate some paths of the CFG in order to increase the size of superblocks, and thus the scope of our superblock scheduling. We implement several kinds of path duplications: (1) *tail duplication* consists in duplicating a path after a join point (i.e. a new superblock entry), in order to move that join point further and hence, extends the superblocks ending on this join point (See Figure 9); (2) *loop body unrolling* consists in unrolling a whole (innermost) loop in order to make its unrolled loop body a big superblock, and thus enable the scheduling across the original loop iterations; (3) *loop rotation* consists in turning "while-do" loops into "do-while" loops, by duplicating the exit-condition and its context: this enables scheduling the context of the exit-condition within the loop body and removes a "goto" in the loop.

We implement each of these RTL transformations by a dedicated *untrusted oracle* and use a single formally-verified test to dynamically check that they preserve the semantics of the original RTL program (see Section 4.1). Finally, the selection of our superblocks for scheduling combines branch predictions (presented in Sections 4.2 and 4.3) and these path duplications (see Section 4.4).

4.1 Formally-Verified Checker of Path Duplications

Tail duplication and the various loop unrollings share an important commonality: they duplicate paths in the CFG, while preserving syntactically the CFG structure modulo renaming of nodes. This is what we call *path duplications*. Hence, we perform each of these transformations via a dedicated untrusted oracle and introduce a single formally-verified test able to dynamically check their results (the checker is also able to verify in a single run any combination of these oracles). Moreover, this checker (with its correctness proof) is both simple and small (only 650 lines of Coq).

Hence, each of these oracles of Coq type "`RTL.function → code * node * (PTree.t node)`" is expected to return: (1) the resulting CFG (of type `code`); (2) the new entrypoint in this new CFG; (3) a mapping from nodes of the *new CFG* to nodes of the *original* CFG, that we call the *duplicate mapping*, and noted ϕ on Figure 9.

Figure 9 illustrates the path duplication: the original code is on the left-hand side; the transformed code is on the right one; and the duplicate mapping ϕ, in red, indicates which node originated from where. Nodes 1, 2, 3 and 4 are unchanged: for all $i \in \{1, 2, 3, 4\}$, $(i \mapsto i) \in \phi$. A new node 4, named 4′ in the figure, is introduced: it is a duplicate of node 4, denoted $(4' \mapsto 4) \in \phi$. Node 3 is then modified so that its successor becomes 4′.

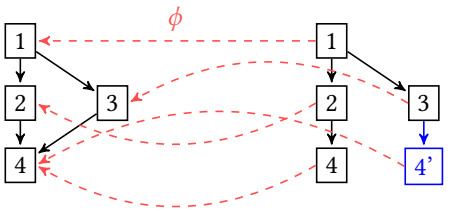

Fig. 9. Example of Path Duplication

Thanks to the duplicate mapping ϕ, it is very simple to check that any execution step of the function f_1 associated to the original CFG is simulated by the new function f_2, for a lockstep simulation, as pictured on Figure 10. Informally, $S_1 \sim S_2$ means that the current program counters n_1 of state S_1 in f_1 and n_2 of state S_2 in f_2 satisfy $(n_2 \mapsto n_1) \in \phi$ (and that return addresses of the respective stacks also match for duplicate mappings of caller functions).

First, our checker verifies that the entrypoint e_1 of f_1 maps the entrypoint e_1 of f_1 through ϕ, ie $(e_2 \mapsto e_1) \in \phi$. Then, for any pair $(n_2 \mapsto n_1)$ in ϕ, we check that, if n_1 is node in f_1 CFG, then n_2 is also a node in f_2 CFG such that: (1) the instruction at node n_1 in f_1 CFG syntactically matches the instruction at node n_2 in f_2 CFG (e.g. if the instruction at n_1 performs assignment "$r' := r_1 + r_2$", then the instruction at n_2 performs exactly the same assignment); (2) n_1 (in f_1 CFG) and n_2 (in f_2 CFG) have the same number of successors; (3) for any i-th successor n_1' of n_1 (in f_1 CFG), the i-th successor n_2' of n_2 (in f_2 CFG)

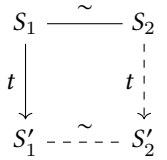

Fig. 10. Lockstep

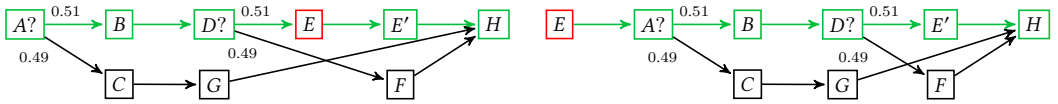

Fig. 11. Program with two branches. Left figure: original program. Right figure: the instruction E is moved before A (under the non-trivial assumption that it is correct to do so). An identified path (that could be turned into a superblock) is shown in green. Each conditional branch edge is annotated with its probability.

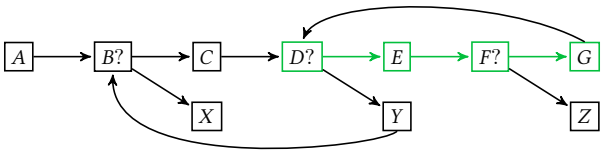

Fig. 12. Whole Body of an Innermost Loop within a Superblock

must satisfy $(n'_2 \mapsto n'_1) \in \phi$. Hence, any step from n_1 can be simulated by a step from n_2 while preserving the simulation relation $\sim$ of Fig. 10.

Note that, this verifier may also accept oracles that prune unreachable part of the CFG (from e_1), as long as these pruned nodes do not appear in ϕ. We have extended the checker described above in order to accept oracles that negate condition on conditional branch: in this case, the two successors must also be exchanged. Indeed, on some pipelined architecture like the KVX, it is more efficient to have the most frequent successor in the "*ifnot*" successor than in the "*ifso*": taking the conditional branch induces a pipeline stall.

4.2 Principles of our Static Branch Prediction

Depending on the branch prediction, our optimizations may end up in a performance gain or a loss.

4.2.1 Storing Prediction Information. Lee et al. [1993] advise storing profiling information as one floating-point number per branch edge, indicating the probability for the execution to take that edge: superblocks are thus selected by following edges with the largest probability. However, selecting paths according to "local" probabilities may lead to bad predictions, as illustrated on the CFG in Figure 11. Here, each letter represents one instruction, with A and D being conditional branches. If each branch has say a 51% probability to take the privileged branch, and we move instruction E to the top of instruction A (because the scheduler ruled that this would give a faster execution *under the assumption that we follow the path* $A - B - D - E - E' - H$), then we would only have a 26% chance to get a faster execution. Indeed, on that particular example, two others paths are possible: $A - C - G - H$ (49%) or $A - B - D - F - H$ (25%). That move would only be beneficial for the first path, and detrimental for the other two, because of the wasted time executing E (wrongly) speculatively.

Moreover, in our compiler architecture, superblock transformations are scattered across several purpose-fit composable passes. Annotating each conditional branch with a probability number would require to interpret and maintain such a number consistently across all of these passes. This seems tricky for transformations like tail-duplication or loop-unrolling. This led us to adopt one `Option bool` prediction p_c per conditional branch c. (1) p_c = `None`: no prediction attached to c; (2) p_c = `Some true`: the branch instruction c is predicted to be very likely to follow the `ifso` branch (condition evaluates to `true`); (3) p_c = `Some false`: the branch instruction c is predicted to be very likely to follow the `ifnot` branch (condition evaluates to `false`). On the CFG in Figure 11, we would have labeled each branch to be "`None`" and stop the path, rather than identifying a *too big* path whose scheduling results in worse performance.

4.2.2 Detecting Innermost-Loops. Figure 12 represents a program consisting of two nested loops. The innermost loop starting at test "D?" is the one of interest: this is usually where most of the computation time lies. Ideally, our static branch prediction should favor the path staying within the innermost loops, instead of the one going out: the "F?" test should be predicted as "`Some false`", in order to enable loop unrolling, which builds a big superblock by duplication of this loop body.

However, there are other examples where both successors of a given test remain within the loop. In such a case, without additional profiling information, we end the superblock at that test. As detailed in Section 4.2.1, a wrongly predicted branch can be harmful to overall performance. In contrast, an unpredicted branch that could have been predicted will just result in a missed opportunity for optimization: it will not decrease performance compared to the original code.

4.3 Acquiring Prediction Information

Prediction information can be acquired by profiling (Section 4.3.1) or by static analysis (Section 4.3.2). Profiling consists in instrumenting the code to record execution statistics into a file. Then, that file is used on next compilations to guide branch prediction. In the absence of profiling information, we exploit certain patterns in the program in order to guess the most likely direction of many branches. In particular, innermost loops as well as their exit branches are detected by static prediction.

4.3.1 Prediction by Profiling. We developed a profiling system in our version of CompCert. As common, our profiling system is used in two steps. First, the program is compiled with special instrumentation: (1) counters are inserted into each object file as local symbols; (2) right after each branch, a special CompCert builtin `EF_profiling` is inserted, which increments the appropriate counter; (3) at program exit, all the counters are written to a file, through special linker sections added to each compiled object file, as for C++ destructors. The program is then run on representative input, and branch counts are accumulated in a log file.

Second, once enough profiling information has been recorded, we use it to add prediction information to branches. The compiler loads the logging file and the profiler-based heuristic consists in assigning the prediction to "`None`" if the relative difference between the two branches counts is below a given small threshold, or "`Some b`" if branch "`b`" is more executed than the other.

4.3.2 Prediction by Heuristics. When no profiling is available, our heuristics—mostly inspired by Ball and Larus [1993]—perform an educated guess of the privileged direction. They are run in sequence, until one of them decides a prediction, otherwise preserving the default "`None`" prediction.

(1) In a conditional branch, a comparison such as $(x < 0)?$ is likely to be an error-code check, so we predict that the check succeeds (that is, the condition is not taken). Similarly, float equality checks are predicted to be `false`.
(2) If a given branch leads to a `return`, then that branch is unlikely to be taken.
(3) If a branch leads away from a loop (the destination is not in the loop body) while the other stays in the loop, we predict the looping branch. This heuristic is very important for later identifying the superblock following that innermost loop.
(4) Finally, if one branch leads to a `call` instruction and the other does not, privilege the latter.

Experimentally, these heuristics seem to detect most branches of interest, though it seem to remain a few slightly disappointing corner cases, in particular for heuristic (1) on conditions.

4.4 Path Selection from Predictions

Once branch prediction has been done, several phases of path selection are done: (1) for tail-duplication, a selection of the paths that will be transformed into superblocks; (2) for loop unrolling and loop rotation, we select the superblocks that encompass innermost loops; (3) for superblock scheduling, the pass "RTL-to-RTLpath" of Figure 13 involves a superblock selection, which partitions

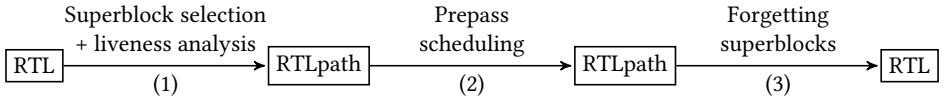

Fig. 13. Our Formally-Verified RTL Superblock Scheduler

each function into superblocks; (4) finally, the linearization pass, "LTL-to-Linear" pass on Figure 2, also partitions each function into superblocks (but on a different representation).

Among these four phases, the path selection by tail-duplication is the most complex, and the one that ends up defining which parts of the code will be optimized. We mostly use the algorithm from [Chang and Hwu 1988], which consists in selecting a node that has the largest execution count, then growing a path forward (by going through the "best successors" in regard to execution count), and then grow it backward. Then, the path stops, and the next unvisited node is selected to grow a new path. Key differences between our implementation and their algorithm:

(1) They have access to precise execution counts for each node. We do not. We follow instead the branch prediction information, stopping the path when that information is None.
(2) Their algorithm was not designed for superblocks: it allows node sharing between paths. In contrast, we enforce paths without intersection, by ending paths before such node sharing.

5 DESIGN OF OUR FORMALLY-VERIFIED SUPERBLOCK SCHEDULER

Figure 13 sketches the design of our formally-verified RTL superblock scheduler. RTLpath is a new IR which annotates RTL programs with information about superblocks: entry-points, exit-points and register liveness. It also represents the execution of a whole superblock in a single step.

Hence, our formally-verified superblock scheduler requires that its input RTL program has been previously rewritten, as described in Section 4, in order to exhibit "relevant" superblocks in each function. It is the "RTL-to-RTL" pass composing the 3 successive simulations depicted in Figure 13.

(1) An untrusted oracle provides the RTLpath annotations for the original RTL program. A formally-verified test checks the correctness of these annotations. It ensures that the RTLpath execution of the original program simulates its RTL execution.
(2) An untrusted scheduler provides a new RTLpath program, with a reverse mapping relating its superblock entry-points to those of the original program. Our formally-verified simulation test ensures that the execution of the scheduled program simulates that of the original.
(3) The final pass simply forgets the RTLpath annotations of the scheduled program. By construction, the RTL execution of the scheduled program simulates its RTLpath execution.

5.1 Definition of the RTLpath IR

RTLpath extends RTL CFG (Control-Flow-Graph) of instructions, with a super CFG-structure of paths, where such a path represents a superblock. But, for the formal proofs, the path-structure does not need to partition the CFG into superblocks: two distinct paths are not required to be disjoint.

Our notion of CFG path is more like the usual notion of *trace* in "trace-scheduling". It derives from `default_succ` defined in Figure 14. We say that a node n in the RTL CFG is a default successor of instruction i, iff `(default_succ i)`$=n$. For $p \geq 0$, a node n_2 in the RTL CFG is the p-th default successor of a node n_1, if there is a sequence of $(p+1)$ nodes in the CFG from n_1 to n_2 such that each node in the sequence is the default successor in the CFG of the previous one in the sequence. Hence, we define a path of size p and of entry-point n as the sequence of nodes leading to the p-th default successor of n (called the final node of the path). Note that instructions with a default successor in the CFG are either basic instructions, or conditional branches. The whole execution of a path of

```
Definition default_succ (i: instruction): option node :=
  match i with
  | Inop s | Iop _ _ _ s | Iload _ _ _ _ _ s | Istore _ _ _ _ s ⇒ Some s (* basic inst *)
  | Icond _ _ ifso ifnot _ ⇒ Some ifnot (* conditional branch *)
  | _ ⇒ None (* others *)
  end

Record path_info := { psize: nat; input_regs: Regset.t; (* ... *) }
Definition path_map: Type := PTree.t path_info
Definition path_entry (pm: path_map) (n: node): Prop := pm!n <> None
Inductive wellformed_path (c:code) (pm: path_map): nat → node → Prop :=...
Definition wellformed_path_map (c:code) (pm: path_map): Prop :=
  ∀ n path, pm!n = Some path → wellformed_path c pm path.(psize) n
Record function : Type :=
 { fn_RTL:> RTL.function;  fn_path: path_map;
   fn_entry_point_wf: path_entry fn_path fn_RTL.(fn_entrypoint);
   fn_path_wf: wellformed_path_map fn_RTL.(fn_code) fn_path
 }

...
(* path step semantics *)
Inductive path_step ge pge (path:nat) stack f sp rs m pc: trace → state → Prop :=...
```

Fig. 14. Definition of RTLpath functions

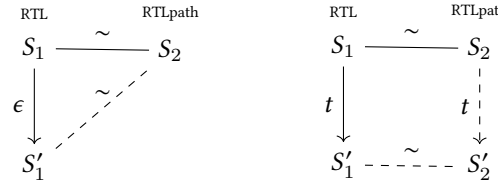

Fig. 15. RTL-to-RTLpath

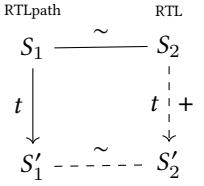

Fig. 16. RTLpath-to-RTL

size p emits at most one observational event: the one emitted by its final node (this is important for forward simulation proofs). Also, note that one given CFG path represents several *execution paths*: the execution may early exit from the CFG path through an intermediate conditional branch.

Records of type `path_info` in Figure 14 store information relative to each path of the CFG. Mainly, the field `psize` giving the size of the path, and optional liveness information which is only relevant for the original RTLpath program, but not the scheduled one.[10] Figure 14 also defines RTLpath functions. A RTLpath function extends a RTL function (field `fn_RTL`), with a map associating nodes to `path_info` (field `fn_path`). By convention, a node with an associated `path_info` in `fn_path` is called a *path entry*. A RTLpath function must provide a proof that the entry-point of the RTL function is a path entry, and also a proof that `fn_path` is *well-formed*, i.e.: (1) for each path entry n associated to a `psize` p in `fn_path`, there exists a p-th default successor in `fn_RTL.(fn_code)` (the RTL CFG); (2) each *path exit*—i.e. any successor of a node in the path that is not itself in the path—is a path entry in `fn_path`.

A step of RTLpath execution (defined through the `path_step` definition sketched in Fig. 14) runs all RTL instructions from a path entry to a path exit. Technically, the notion of RTLpath states must extends the notion RTL states in order to remember that caller functions (stored in the shadow stack of the state) are actually RTLpath functions.

[10]Because, our simulation test only needs liveness information about the original RTLpath, but not about the scheduled one.

Algorithm 1 Checking well-formedness & liveness information of CFG paths

```
procedure PATH_LIVENESS_CHECKER(f)
    for all (pc,path) :∈ f.fn_path do
        live := path.input_regs
        for p := 0 to path.psize do                    ▷ pc is the p-th "default succ." in the path
            inst := f.fn_RTL.fn_code[pc]               ▷ Fails if pc is not in the CFG
            r := read_regs(inst)  ||  w := written_regs(inst)
            assert(r ⊆ live)
            live := live ∪ w  ||  succ := successors(inst)
            if p < path.psize then
                Some(pc) := default_succ(inst)         ▷ Fails if no default successor
                succ := succ\{pc}
            for all path_exit :∈ succ do
                assert(f.fn_path[path_exit].input_regs ⊆ live)   ▷ Fails if path_exit is not a path entry
```

5.2 Bisimulation of RTLpath and RTL Executions

With these definitions, the forward simulation of RTLpath by RTL is reduced to a simple "Plus-simulation" as pictured in Figure 16: a RTLpath step runs at least one RTL instruction, but at most $p + 1$ where p is the size of the executed path entry.

The reverse simulation, of RTL by RTLpath, is less trivial. In a RTLpath execution, successive states necessarily correspond to entry paths. In particular, each return address stored in the stack is itself an entry path of the return function: this is because a CALL instruction has no default successor on Figure 14. Relation $\sim$ matching RTL states with RTLpath states encodes these invariants. More generally, $S_1 \sim S_2$ relates an RTL state S_1 with an RTLpath state S_2 that is the entry point of a CFG path containing S_1. As pictured in Figure 15, one RTL step is simulated by one RTLpath step on path exits. Else, the RTLpath execution stutters. To prevent silent infinite loops from being simulated by any program, CompCert forward simulations require proving that the number of successive stuttering steps is finite: here, this number is bounded by the size of the current CFG path.

5.3 Equivalence of RTLpath Executions Modulo (Register) Liveness

The well-formedness of the RTLpath program produced from the original RTL program, by the oracle pictured in pass (1) of Figure 13, is dynamically verified by the certified checker described in Algorithm 1, applied to each RTLpath function candidate `f`. Explicit assertions in this algorithm also check that the liveness information is correct: the `input_regs` field on each path entry actually corresponds to the set of registers that are read in any execution starting from this path entry. Our oracle partly reuses the existing liveness analysis of CompCert on RTL CFG.

In our Coq code, the success of this procedure on a RTLpath function is abstracted as predicate "`liveness_ok_function`". The main lemma proved on this predicate is given by the `path_step_eqlive` lemma in Figure 17. Indeed this lemma allows us to define a "lockstep" simulation (see Fig. 10) of RTLpath execution modulo liveness: here, $S_1 \sim S_2$ means that RTLpath states S_1 and S_2 only match on *live registers*, including the "live registers" that have been stored in the stack during function calls. In practice, the liveness information of caller functions are recovered from the shadow stack of RTLpath states.

Roughly speaking, `path_step_eqlive` expresses that for all path step starting from an initial state with a (shadow) stack `stk1`, a register state `rs1` and a memory state `m`, such that this path step emits a trace `t` and a final state `s1`, for all register state `rs2` equivalent to `rs1` modulo liveness (at the path entry), for all stack `stk2` equivalent to `stk1` modulo liveness, there exists a path step

```
Definition eqlive_reg live rs1 rs2: Prop := ∀ r, (Regset.In r live) → rs1#r = rs2#r
Inductive eqlive_states : state → state → Prop :=...
Definition eqlive_stacks: (list stackframe) → (list stackframe) → Prop :=
Lemma path_step_eqlive f path stk1 sp rs1 m pc t s1 stk2 rs2:
  liveness_ok_function f →
  (f.(fn_path)) ! pc = Some path →
  path_step ... path.(psize) stk1 f sp rs1 m pc t s1 →
  eqlive_stacks stk1 stk2 →
  eqlive_reg path.(input_regs) rs1 rs2 →
  ∃ s2, path_step ... path.(psize) stk2 f sp rs2 m pc t s2 ∧ eqlive_states s1 s2
```

Fig. 17. Equivalence of RTLpath Steps Modulo (Register) Liveness

```
Record match_function (dm: PTree.t node) (f1 f2: RTLpath.function): Prop := {
  preserv_entrypoint: dm!(f2.(fn_entrypoint)) = Some f1.(fn_entrypoint);
  dupmap_path_entry1: ∀ pc1 pc2, dm!pc2 = Some pc1 → path_entry (fn_path f1) pc1;
  dupmap_path_entry2: ∀ pc1 pc2, dm!pc2 = Some pc1 → path_entry (fn_path f2) pc2;
  dupmap_correct: ∀ pc1 pc2, dm!pc2 = Some pc1 → sexec_simu dm f1 f2 pc1 pc2;
  (* + a few extra properties expressing that f1 and f2 share the same interface *)
}
```

Fig. 18. Matching of RTLpath Function through the Symbolic Test Simulation

starting from `stk2` and `rs2` instead, emiting the same trace `t` and a final state `s2` equivalent to `s1` modulo liveness.

5.4 Design of the Certified RTLpath Scheduler

The Coq interface of our RTLpath scheduler is declared by the following directive:

```
Axiom untrusted_scheduler: RTLpath.function → code * node * path_map * (PTree.t node)
```

Given an original function from RTLpath, the oracle devises a schedule for each superblock (each path from the function). It returns a tuple $(c, e, pm, \texttt{dm})$ where c is the scheduled RTL CFG, e is its main entry-point, pm is its associated pathmap, `dm` is the reverse mapping from entry paths of the scheduled CFG of the original CFG (like the duplicate mapping ϕ in Section 4.1). Given an original RTLpath function `f1`, our formally-verified code turns the (c, e, pm) returned by the oracle into a RTLpath function `f2`, after verifying the well-formedness conditions through a variant of Algorithm 1 which ignores liveness information. Then, it applies various checks in order to ensure that `f2` "matches" `f2` for `match_function` of Figure 18: (1) properties `dupmap_path_*` express that `dm` is a mapping from path entries of `f2` to path entries of `f1`; (2) property `dupmap_correct` expresses that for each path entry `pc2` of `f2`, given `pc1` its matching path entry in `f1`, our formally-verified simulation test (detailed in Section 5.5) validates that the symbolic execution of the `f2` path starting at `pc2` simulates the symbolic execution of the `f1` path starting at `pc1` modulo live registers of `f1`. Formally, the forward simulation proof of our scheduler corresponds to a lockstep simulation "RTLpath-to-RTLpath" modulo live registers of `f1` (see Figure 10).

5.5 Certifying the RTLpath Simulation Test by Refinement

Like [Six et al. 2020] our simulation test by symbolic execution is defined in an intermediate representation, RTLpath, which is generic w.r.t. the target architecture. Our formal proof is organized in a similar way: we first define an abstract model of symbolic execution, allowing for a definition of a specification for the simulation test in this symbolic semantics, then, we refine this model into an efficient implementation that uses their formally-verified hash-consing technique. Indeed, as recalled in Section 2.1.2, this technique, based on the embedding of a trusted pointer equality into Coq and an untrusted hash-table, requires their monad of "impure" computations. The core of our simulation test, function `imp_simu_check` below, is implemented in this monad and proved

correct by lemma `imp_simu_check_correct`, expressing that when `imp_simu_check` normally terminates then property `dupmap_correct` of Fig. 18 holds. The primary goal of their refinement technique is to circumscribe the reasoning on impure computations as much as possible.[11]

```
Definition imp_simu_check (dm: PTree.t node) (f tf: RTLpath.function): ?? unit :=...
Lemma imp_simu_check_correct dm f tf:
 WHEN imp_simu_check dm f tf ⇝ _ THEN ∀ pc1 pc2, dm!pc2=(Some pc1) → sexec_simu dm f tf pc1 pc2
```

However, our abstract model of the simulation test, detailed in Section 5.6, is more complex than the one of [Six et al. 2020] on several points. (1) We consider *superblocks*: the symbolic state thus must represent several execution paths (where [Six et al. 2020] only deal with a single execution path). (2) Our simulation test compares symbolic state on each path exit modulo liveness. (3) Our notion of RTL/RTLpath state is richer than their notion of Asm/Asmblock state. For example, our notion of symbolic state partly abstracts away the RTLpath shadow stack, with a dedicated category of symbolic values, because each path step runs at constant stack, except on the final instruction.

Moreover, our implementation of the simulation test, detailed in Section 5.7, is also more complex. (1) Our rewriting rules are directly integrated in the symbolic execution engine, whereas their rewriting rules happens during the "Asmblock-to-AbstractBasicBlock" compilation (see Figure 4). As explained in Section 3.2, our approach gives simpler proofs about these rewriting rules. But, this makes the symbolic execution implementation a little more tricky. (2) In CompCert's memory model, the arithmetic operators comparing pointers fail when these pointers are allocated in distinct memory blocks. This feature is convenient for proving the correctness of compilation passes: we do not have to prove that the compilation preserves comparison of pointers allocated in distinct memory block, because the proof already assumes that the input program does not perform such comparison (the input program is assumed to not fail). However, some arithmetic operators may contain a read dependency on memory, which may lead a naive implementation of the simulation test to reject some desirable schedules. The solution of [Six et al. 2020] to this technical issue cannot be simply applied in our case: their solution works on the last IR of CompCert (assembly), but not in an intermediate IR such as RTLpath. Fortunately, we have found another very simple solution, easily proved in our design by refinement.

Let us quantify this increased complexity by comparing the size of the respective Coq developments. The model of symbolic execution (with the bisimulation theorems of symbolic execution w.r.t concrete execution) in [Six et al. 2020] represents around 300 lines of Coq. In our case, it represents almost 1500 lines of Coq. Their specification of the simulation test is a few lines of Coq. In contrast, our detailed model of the simulation test (with the proof that these detailed specifications are indeed correct) represents 700 additional lines of Coq. Finally, our implementation (including its proof w.r.t. the model, but excluding the rewriting rules specific to each processor target) also represents around 1.5 Kloc of Coq. Their implementation represents around 700 lines of Coq.[12] In short, around 1.4 Kloc of Coq for their development vs 3.7 Kloc for ours, our model being seven times bigger than their one, whereas our implementation is only two times bigger.

[11] At some point, `imp_simu_check` is coerced into a pure function returning a "`option unit`". We now argue that such a trusted coercion is safe as soon as the "`None`" case, which corresponds to the case where the function does not terminate normally, can never be observed from a pure computation. Thus, even if the function non-determistically diverges, this non-determinism cannot be observed. In other words, the trusted coercion has to catch exceptions and diverge instead.

[12] This line number on their implementation excludes 300 lines of Coq for a debugging system that we have not ported, because it would be too complex in our case. Its purpose is to provide debug traces when the simulation test fails: this helps to debug the scheduling oracles, or to discover expressiveness issues in the simulation test itself. In our case, we succeeds to debug with the help of `ocamldebug` plus a few traces manually inserted in the sources or in the extracted code.

Representing the left-hand side of Fig. 7

```
if (r4 ≥ r2) {
  r32 ≔ (r4 ≥ r2);
  goto .L101
}
  r0 ≔ (r0+(lws.ws(sxwd(r4),r1))
‖ r3 ≔ (lws.ws(sxwd(r4),r1))
‖ r4 ≔ (r4 + 1)
‖ r5 ≔ (sxwd(r4))
‖ r32 ≔ (r4 ≥ r2);
goto .L100
```

Representing the right-hand side of Fig. 7

```
if (r4 ≥ r2) {
    r3 ≔ (lws.s.ws(sxwd(r4),r1))
  ‖ r5 ≔ (sxwd(r4))
  ‖ r32 ≔ (r4 ≥ r2);
  goto .L101
}
  r0 ≔ (r0+(lws.s.ws(sxwd(r4),r1)))
‖ r3 ≔ (lws.s.ws(sxwd(r4),r1))
‖ r4 ≔ (r4 + 1)
‖ r5 ≔ (sxwd(r4))
‖ r32 ≔ (r4 ≥ r2);
goto .L100
```

Fig. 19. Respective Symbolic States of Figure 7 Superblocks.

5.6 Model of the Symbolic Execution and the Simulation Test

The symbolic representation of [Six et al. 2020] for a basic block is a preconditioned parallel assignment ended by `goto` (PPAG) as recalled in Example 2.1. We now detail our generalization for RTL "superblocks". Figure 19 illustrates our symbolic representations for KVX superblocks of Figure 7. We represent each execution path of the superblock as a PPA ended by `goto` (PPAG): a superblock is represented by a *sequence of guarded PPAG*. On Figure 19, all PPAG have a valid precondition, because all possible failure of the original superblocks are still present in parallel assignments. Figure 20 sketches the abstract syntax of our (model of) symbolic states.

Actually, Figure 7 represents the symbolic states returned by our *model* of the symbolic execution: the rewriting of trapping `lws.ws` instruction into the non-trapping `lws.s.ws` one is not represented here, because these rewriting is only achieved in the *implementation* of the symbolic execution. Hence, our *specification* of the simulation test (predicate `sexec_simu` of Figure 18) expresses that the right-hand side simulates the left-hand side by a *semantical* comparison of the *symbolic states* modulo register liveness. Like [Six et al. 2020], we prove on this model, that the (semantical) simulation of symbolic states (modulo liveness) suffices to prove the semantical simulation of RTLpath steps (modulo liveness). This actually results from Theorems `sexec_correct` and `sexec_exact` of Fig. 21, which together expresses that:

> "given a path entry `pc` and given a symbolic state `st` returned by the symbolic execution model on `pc`, then semantics of `st` *bisimulates* the RTLpath step from `pc`."

Simulation *modulo liveness* is proved by combining this strong bisimulation (which does not itself consider liveness) with predicate `sexec_simu` (which defines a semantical view of our simulation test). This predicate `sexec_simu` is itself refined into a *detailed model* performing a pairwise comparison of the guards of the two symbolic states: guards must appear in the same order, their conditions are compared for syntactical equality, their list of arguments are compared for semantical equalities of symbolic values, and the PPAG are semantically compared modulo register liveness.

5.7 Implementation of the Simulation Test

Our implementation of the symbolic execution uses the formally-verified defensive programming technique of [Six et al. 2020] for hash-consing, based on an untrusted hash-table, but a trusted pointer equality (as recalled in Section 2.1.2). Figure 22 sketches the Coq definition of the implementation of our symbolic values (`hsval`). Replicating the approach of [Six et al. 2020], our implementation duplicates the definitions of the model (e.g. `sval`) while inserting a hash-identifier `hid` of type `hashcode` on each node. Here are some important invariants of the hash-consing mechanism,

```
(* Model of Symbolic Values and Memories *)
Inductive sval := Sinput (r: reg)
  | Sop (op:operation) (lsv: list_sval) (sm: smem)
  | Sload (sm:smem) (trp:trapping_mode) (chk:memory_chunk) (addr:addressing) (lsv:list_sval)
with list_sval := Snil | Scons (sv: sval) (lsv: list_sval)
with smem := Sinit
  | Sstore (sm: smem) (chunk:memory_chunk) (addr:addressing) (lsv:list_sval) (srce:sval)

(* Model of Preconditioned Parallel Assignment (PPA) *)
Record sistate_local := {
  si_pre: RTL.genv → val → regset → mem → Prop; (* precondition on input memory and regs. *)
  si_sreg: reg → sval; si_smem: smem        (* parallel assignment *)
}

(* Model of Guarded PPA ended by Goto *)
Record sistate_exit := {
  si_cond: condition; si_scondargs: list_sval; (* Symbolic Guard *)
  si_elocal: sistate_local; si_ifso: node  (* PPA ended by Goto (PPAG) *)
}

(* Model of Sequences of Guarded PPAG *)
Record sistate := { si_exits: list sistate_exit; si_local: sistate_local; si_pc: node }

(* Model of Symbolic Final Value (modifiying the stack or emitting observational events) *)
Inductive sfval := Snone
  | Scall (sig:signature) (svos:sval+ident) (lsv:list_sval) (res:reg) (pc:node)
  | Sreturn: option sval → sfval
 ...

(* Model of Symbolic State *)
Record sstate := { internal:> sistate; final: sfval }
```

Fig. 20. Abstract Syntax of our Symbolic Representation of "Superblocks"

```
(* Model of Symbolic Execution *)
Definition sexec (f: function) (pc:node): option sstate :=...

(* Semantics of Symbolic State Models *)
Inductive ssem pge ge sp (st: sstate) stack f rs0 m0: trace → state → Prop :=...

(* Symbolic State Simulates Path Step *)
Theorem sexec_correct f pc pge ge sp path stack rs m t s:
  (fn_path f)!pc = Some path →
  path_step ge pge path.(psize) stack f sp rs m pc t s →
  ∃ st, sexec f pc = Some st ∧ ssem pge ge sp st stack f rs m t s

(* Path Step Simulates Symbolic State *)
Theorem sexec_exact f pc pge ge sp path stack st rs m t s1:
  (fn_path f)!pc = Some path →
  sexec f pc = Some st →
  ssem pge ge sp st stack f rs m t s1 →
  ∃ s2, path_step ge pge path.(psize) stack f sp rs m pc t s2 ∧ equiv_state s1 s2
```

Fig. 21. Bisimulation of Symbolic States Semantics w.r.t RTLpath Semantics

which do not, however, require a formal proof (because they are only a matter of "performance", not of "formal correctness"): (1) a `hid` has either the special value `unknown_hid` or has been allocated by the hash-consing oracle; (2) a node contained a `hid` distinct from `unknown_hid` (being thus allocated) never contains a subtree with an `unknown_hid`. Actually these invariants were already introduced in [Six et al. 2020]. However, we exploit invariant 2) in a new way, in order to simplify the proofs about rewriting rules, as further detailed in Section 5.8.

Except for the `hids`, the only difference between `hsval` (Fig. 22) and `sval` (Fig. 20) is that the `HSop` constructor does not depend on any "symbolic memory", whereas `Sop` depends on a symbolic memory `sm`. We relate our "concrete symbolic values" of type `hsval` to their model of type `sval` by the `hsval_proj` function, which removes `hid` information and uses `Sinit` as the

```
(* Implementation of Symbolic Values and Memories *)
Inductive hsval :=
  | HSinput (r:reg) (hid:hashcode)
  | HSop (op:operation) (lhsv:list_hsval) (hid:hashcode)
  | HSload (hsm:hsmem) (trp:..) (chk:..) (addr:..) (lhsv:list_hsval) (hid:hashcode)
with list_hsval := HSnil (hid:hashcode) | HScons (hsv:hsval) (lhsv:list_hsval) (hid:hashcode)
with hsmem := HSinit (hid:hashcode)
  | HSstore (hsm:hsmem) (chk:..) (addr:..) (lhsv:list_hsval) (srce:hsval) (hid:hashcode)

(* Abstraction of Concrete Symbolic Values and Memories *)
Fixpoint hsval_proj hsv: sval :=
  match hsv with
  | HSinput r _ ⇒ Sinput r
  | HSop op hl _ ⇒ Sop op (hsval_list_proj hl) Sinit
  | HSload hm t chk addr hl _ ⇒ Sload (hsmem_proj hm) t chk addr (hsval_list_proj hl)
  end
with hsval_list_proj hl: list_sval :=...
with hsmem_proj hm: smem :=...
```

Fig. 22. Implementation of the Symbolic Values

```
(* Implem. of PPA *)
Record hsistate_local := { hsi_ok_lsval:list hsval; hsi_sreg:PTree.t hsval; hsi_smem:hsmem }
(* Implem. of Guarded PPAG *)
Record hsistate_exit :=
  { hsi_cond:condition; hsi_scondargs:list_hsval; hsi_elocal:hsistate_local; hsi_ifso:node }
(* Implem. of Sequences of Guarded PPAG *)
Record hsistate := { hsi_pc:node; hsi_exits:list hsistate_exit; hsi_local:hsistate_local }
(* Implem. of Symbolic Final Value *)
Inductive hsfval := HSnone
  | HScall (sig:signature) (svos:hsval+ident) (lsv:list_hsval) (res:reg) (pc:node)
  | HSreturn (res:option hsval)
 ...
(* Implem. of Symbolic State *)
Record hsstate := { hinternal:> hsistate; hfinal: hsfval }
```

Fig. 23. Abstract Syntax of our Concrete Symbolic States

symbolic memory of all `Sop` nodes. Here, `Sinit` represents the initial memory when executing the underlying RTL superblock. Indeed, as explained in Section 5.5, the semantics of arithmetic operators may access the allocation table of the current memory. But within a superblock, the only instruction that modify the memory are "store" instructions which do not change this allocation table. Hence, the semantics of arithmetic operators do not change if we use the initial memory of the superblock instead of the current memory. Formally, this idea is expressed through a dedicated invariant of our data-refinement relation that relates "concrete PPA" of type `hsistate_local` (in Fig. 23) to their model of type `sistate_local` (in Fig. 20). Except for this additional invariant, this data-refinement relation is very like the `smem_model` relation of [Six et al. 2020].

Moreover, we have also refined each type of the abstract syntax given in Figure 20 into those of Figure 23. See our Coq sources online (its non-anonymous URL is associated to our submission) for details on these (not so straightforward) data-refinement relations. Like [Six et al. 2020], the proof of our implementation simply reduces to ensuring that each elementary computation of the symbolic execution preserve the data-refinement relations w.r.t. its abstract model, and finally that the physical or syntactic equalities involved in the implementation of the simulation test implies the semantical equalities involved in its detailed model.

5.8 Implementing Verified Rewriting Rules during the Symbolic Execution

As explained in Section 5.5, the implementation of the symbolic execution with hash-consing is written and proved correct within the Impure monad, also used by [Six et al. 2020]. Our rewriting function is defined as a transformation over hash-consed terms during the symbolic execution.

However, the design of this rewriting engine discharges the formally-verified rewriting rules of producing hash-consed terms within the Impure monad. Indeed, it exploits the fact that: (1) our rewriting rules are always applied at the top of hash-consed terms (like a smart constructor); (2) a node contained a `hid` distinct from `unknown_hid` (being thus allocated) never contains a subtree with an `unknown_hid`. Thus, our rewriting rules are defined as pure functions which produce new top nodes marked with `unknown_hid`. The rewritten terms are then turned into proper hash-consed terms by a dedicated (impure) function that only transform the top nodes of these terms. In other words, the management of hash-consing during rewriting is delegated to this dedicated function.

Currently, we store all potentially trapping terms of each PPAG into the `hsi_ok_lsval` field without any rewriting. Hence, rewriting is only applied within the parallel assignment expressed in `hsi_sreg` (see Fig. 23). Formally, a rewriting rule must turn a term t_1 into a term t_2 such that for all register and memory states such that evaluation of t_1 does not fail, t_2 evaluates to the same value as t_1. This allows us to rewrite a `lws.ws` operation of the KVX into a `lws.s.ws` in parallel assignment, the trap of `lws.ws` being still present in the precondition: our formally-verified simulation test validates the example in Figure 7.

As explained in Section 3.2, on the RiscV backend, we succeeded to move most of assembly expansions expressed at the "Mach-to-Asm" pass in vanilla CompCert (see Fig. 2) as rewriting rules on RTLpath. This required overcoming a little issue: while the forward simulation proof of "Mach-to-Asm" supports that expansions replace "`vundef`" value by any other value, this is not supported in the proof of our rewriting rules.[13] Our simple workaround is to introduce within rewriting rules some dedicated pseudo-instructions able to generate the necessary "`vundef`" (hence, acting like defensive tests): these extra pseudo-instructions are further removed in the "Mach-to-Asm" pass. Note that these extra pseudo-instructions do not disturb the scheduling because they are assigned 0 latency and 0 resource.

On complex ISA, like AArch64, many expansions of "Mach-to-Asm" pass cannot be expressed at RTL level. This is due to limitations of RTL, which does not support arithmetic instructions modifying several pseudo-registers in parallel, such as instructions with side effects on flags. Even on RiscV, expansions that involve stack-accessing instructions cannot really be expressed at RTL level, because the layout of stackframes is not yet defined at this level (see Fig. 2): stack accesses are only handled in a very abstract way.

6 SUPERBLOCK SCHEDULING ORACLE IMPLEMENTATION

Our oracle for superblock scheduling extends the principles of the one from Six et al. [2020] for basic block scheduling. It assigns to each instruction j, including exit branches, a date $t(j)$ expressing the number of clock cycles at which it is estimated that the instruction is executed: 0 is the first instruction in the superblock. Dependencies between reads and writes on register and memory are computed and adorned with latencies expressing the minimum number of clock cycles needed between the two events, then a schedule is computed that respects these dependencies and latencies. (1) A register read as an operand introduces a "read-after-write" dependency from the last write to that register, with a latency δ defined by the instruction that performed that write: δ is the number of clock cycles between the time when the instruction reads its operands and the time when it writes its output; (2) a register written to introduces a "write-after-write" dependency from the last write to that register, with a latency of 1 clock cycle (no simultaneous writes to the same register), and "write-after-read" dependencies from the last reads from that register, with a latency of 0

[13] In CompCert, a "`vundef`" value, e.g. generating by reading in an uninitialized register, does not abort execution. In contrast, accessing to an invalid address "traps": it aborts execution.

(one can read then write to the same register using instructions issued at the same cycle); (3) live variables at side exits are considered as extract reads by the branch instruction.

A dependency with a latency of δ clock cycles between an instruction j and an instruction j' is thus expressed as an inequality $t(j') - t(j) \geq \delta$, or, equivalently, as an edge $j \xrightarrow{\delta} j'$ in the latency graph. Path lengths in that graph matter for prioritizing instructions when scheduling.

Memory is treated as one single big register; there is at present no alias analysis that would allow, for instance, swapping two writes to non-overlapping locations. In order for such an analysis to be added the symbolic execution engine would have to be modified to reason modulo commutativity of memory accesses to memory locations proved not to overlap by this analysis.

In addition, the schedule should obey resource constraints. Each instruction j of instruction class $K(j)$ consumes a vector $\mathrm{u}(K(j))$ of resources: each coordinate of that vector expresses the number of resources taken from a certain class (e.g. load/store units, arithmetic units capable of doing addition/subtraction, floating-point units). The total of resources used by all instructions issued at the same clock cycle is bounded by a vector r: $\forall i \ \sum_{j|t(j)=i} \mathrm{u}(K(j)) \leq \mathrm{r}$.

Compared to postpass scheduling [Six et al. 2020], our prepass scheduler reasons at a higher level of abstraction: not only do we have an unbounded amount of registers, but also certain pseudo instructions have not yet been expanded into sequences of elementary assembly instructions. We however still have to assign latencies and resource uses to instructions. On the Kalray KVX, we generate the assembly instruction sequence, and then call the functions of the postpass scheduler that give latency and resource usage. On AArch64 and RiscV, for a limited number of cores (Cortex-A53, Cortex-A35; Rocket, SweRV EH1), we implemented these functions directly, with numbers from the available documentation or from relevant scheduling parameters in the LLVM compiler.

One difficulty is that the format of latency and resource constraints, originally from [Six et al. 2020], was designed for fully pipelined processors: processors in which there are resources constraints on which instructions can be issued at the same clock cycle, but no constraints across different clock cycles. That is, on a fully pipelined processor, such as the KVX, if a multiplication was issued at a cycle t, then it does not prevent another multiplication from being issued at cycle $t+1$. In many processors, some units, especially dividers, are not pipelined: an instruction entering the unit monopolizes it until completion. A general solution would be to introduce multiple-cycle resource reservations. We are waiting to have more core descriptions to introduce a more general format (with added functionality such as operands read at different cycles, etc.). Meanwhile, we handle this by adding a constraint $t(j') - t(j) \geq \delta$ when j and j' are two successive uses of the same non pipelined unit, where δ is an estimated number of cycles of use.[14] A drawback is that this does not allow reordering operations using the same non-pipelined unit with respect to one another.

Postpass optimization within a basic block [Six et al. 2020] has a single objective: reducing the makespan, defined as the date when the last value produced by the basic block is available. In contrast, superblock prepass optimization may have multiple, conflicting, objectives: (1) reducing the makespan (with respect to the final exit); (2) reducing register pressure (the number of physicals registers needed), or, as a proxy, the live ranges of values; (3) pushing side exits as early as possible.

For solving the constraint system, we reuse the algorithms of [Six et al. 2020]. (1) *Forward list scheduling*: time slots are greedily filled by increasing date (clock cycle), adding instructions to each slot as long as they respect the resource and latency constraints, with a priority for instructions that finish the longest path in the latency graph. Forward scheduling tends to place exit points as soon as possible, but may increase the live ranges of variables. (2) *Backward list scheduling*: time slots are instead greedily filled by decreasing date. (3) In addition, we introduced *zigzag scheduling*: forward scheduling is used to place exit points, and then backward scheduling places other instructions.

[14] The cycle count used by divider units typically depends on the number of bits in the quotient.

	AArch64			KVX			RiscV		
Setup	Q1	Med	Q3	Q1	Med	Q3	Q1	Med	Q3
+Postpass	+2.56%	+9.11%	+20.62%	+14.13%	+31.25%	+46.62%	-	-	-
+LICM	+6.11%	+16.29%	+41.06%	+17.07%	+35.50%	+61.18%	-1.65%	+3.11%	+9.67%
+Prepass	+8.71%	+22.44%	+50.44%	+18.67%	+37.73%	+62.34%	+4.34%	+9.23%	+15.99%
+Loop unroll.	+13.52%	+23.64%	+59.09%	+18.84%	+42.85%	+79.13%	+6.20%	+12.37%	+20.47%
+Loop rotate	+14.14%	+26.15%	+58.40%	+18.78%	+47.34%	+85.49%	+5.44%	+15.18%	+22.39%
+RTL expans.	-	-	-	-	-	-	+6.34%	+15.24%	+24.52%

Table 1. Improvement of Cumulated Optimizations wrt Vanilla CompCert on 3 Benchmarks

GCC-O1	-9.17%	+1.35%	+10.61%	-23.17%	-6.13%	-0.66%	+9.27%	+19.61%	+34.73%
GCC-O2	-3.27%	+2.55%	+10.55%	-0.12%	+5.47%	+11.88%	+42.73%	+63.21%	+74.02%

Table 2. Improvement of GCC wrt "Best CompCert" on Polybench

GCC-O1	-3.05%	+9.47%	+34.17%	-19.66%	+1.71%	+15.11%	+5.38%	+23.68%	+46.54%
GCC-O2	+11.43%	+53.09%	+88.64%	+27.10%	+51.58%	+84.81%	+24.20%	+60.07%	+102.81%

Table 3. Improvement of GCC wrt "Best CompCert" on TACLeBench

Backward	-1.99%	+0.32%	+2.17%	-1.78%	-0.04%	+0.27%	-3.52%	+0.11%	+2.34%
Zigzag	-3.96%	-0.76%	+0.99%	-2.35%	-0.20%	+0.04%	-3.90%	-0.63%	+1.60%

Table 4. Improvement of Alternative Prepass Scheduling wrt List Scheduling on 3 Benchmarks

7 EXPERIMENTAL EVALUATION

Tables 1 to 4 summarize our experiments on several architectures: **AArch64** corresponds to ARM Cortex A53 (AArch64) inside a Raspberry Pi 3 running Ubuntu 18.04.5 LTS;[15] **KVX** is a KV3 "Coolidge" core in actual hardware; **RiscV** is a "Rocket" RiscV core in FPGA. In each case, we tie the process to one core of the machine, and we measure clock cycles using hardware counters. We run three different suites: the benchmark of [Six et al. 2020], the computational oriented Polybench [Pouchet 2012], and the embedded systems oriented TACLeBench [Falk et al. 2016].

First of all, Table 1 measures the cumulative impact of each gradually introduced optimization compared to vanilla CompCert 3.8.[16] Note that the postpass is only active on AArch64 and KVX, and the RTL expansions are only compatible with RiscV. The postpass scheduling for KVX is the one of [Six et al. 2020]. And LICM (Loop Invariant Code Motion) is the one of [Monniaux and Six 2021]. The Q1, Med and Q3 values respectively denote the first quantile, the median, and the third quantile on the entirety of our benches. Here are a few conclusions. (1) For both the AArch64 and KVX cores, postpass scheduling has a significant impact. For the KVX, this impact is bigger, as expected on a VLIW architecture. (2) LICM is another meaningful optimization, producing a gain of 20% on some benchmarks. (3) Prepass scheduling [17] (without any loop unrolling) also helps, increasing by 5-10% for the AArch64 and RiscV cores. This is mostly due to removing the false dependencies (compared to the postpass scheduling). The KVX core, on the other hand, is not affected much since it features 64 user registers. Using both prepass and postpass scheduling together on AArch64 is the best setting, mainly because the latter act as a fine-tuning for in-between expanded instructions (that occur at the Asm level). (4) Loop unrolling, combined with prepass scheduling, increases performance by another 5-10%. (5) Loop rotation used alone has a small impact on AArch64 and RiscV (about +3% on the latter), but the postpass (on AArch64) benefits from it as the rotation may provides more scheduling opportunities. It shines mostly for the KVX architecture, since it results in a more efficient bundling of the loop header in postpass. (6) RTL expansions does not have a

[15] This dual-issue, in-order core was chosen because it is similar to other in-order ARM cores used in embedded systems; also it is used as little core in "big.LITTLE" settings.

[16] For the KVX target, not available in vanilla CompCert, we use the one from [Six et al. 2020] without postpass scheduling (nor bundling).

[17] On the KVX, due to a temporary technical issue not related to CompCert, we disabled the support for non-trapping loads.

significant impact on average (for all benchmarks) but they can, on some programs, result in a large performance gain when combined with the prepass. For instance, we measured that they increase the prepass yield from +7.8% to +10.5% on the Polybench tests.

We also compared our best version of CompCert to GCC. Table 2 (resp. 3) shows the gain (can be negative) of using GCC with the given optimization flag, versus our best version of CompCert, for Polybench (resp. TACLeBench). On Polybench, we are getting closer to GCC -O2 with our optimizations on AArch64 and KVX. On RiscV, we still have a margin of progression: we suspect the lack of postpass scheduling. Also, less work went on our RiscV prepass backend. On TACLeBench however, GCC -O2 beats us by a large margin.

Another experiment, in Table 4, shows that the prepass scheduling algorithms of Section 6 produce almost equivalently efficient code. List scheduling seems generally a little better than its variants for KVX, but not for AArch64 and RiscV where backward seems to give better results.

Finally, we measured on KVX that branch prediction from profiling information, instead of our static heuristics, gives only negligible benefit (+1% on Q3, 0% on Q1 and Median). To evaluate the impact more finely, we first profiled all our benchmarks, then we modified the compiler code to count the number of times that our heuristics gave a wrong prediction. Out of the 17816 branches that were able to be profiled, only 5% of them were wrongly predicted, and 14% of them had a pattern not caught by our used heuristics.[18] This gives us confidence in our static branch prediction.

8 RELATED AND FUTURE WORKS

Trace scheduling, as introduced by Fisher [1981], is more general than *superblock scheduling*. Indeed, trace schedulers may move code across *side exits* or *side entrances* within a given trace, modulo a quite complex *bookkeeping* (i.e. instruction duplications) in other traces. Superblock scheduling [Hwu et al. 1993; Lee et al. 1993] provides a simpler approach—in particular for *formally-verified* compilers—by dissociating the bookkeeping from the actual scheduling, through a preliminary pass of tail duplication. Additionally, tail-duplication may offer new opportunities of redundant code elimination: it is thus interesting to apply CompCert optimizations such as CSE3,[19] constant propagation or deadcode elimination, between tail-duplication and the actual scheduling.

Previously, Tristan and Leroy [2008] attempted to implement some form of trace scheduling *à la* Fisher within CompCert. But, their implementation suffered from exponential complexity, partly due to their lack of control of tail-duplication within the scheduling. Moreover, in contrast to Fisher's trace scheduling, their implementation systematically duplicated instructions that are moved across side-exits; our superblock scheduling, on the other end, can move instructions across side-exits without any duplication (under certain conditions depending on a liveness analysis).

We have not taken register pressure explicitly into account in the scheduling oracle. We only note that backward list scheduling naturally tends to minimize variable lifetime and thus register pressure. Goodman and Hsu [1988] proposed a modified list scheduler, switching to a different strategy when register pressure becomes high. We may in the future introduce such a scheduler, but so far have not found a need for it: backward list scheduling before register allocation does not seem to generate schedules that produce register spills that were not already present in the original program. Shobaki et al. [2013] propose an optimal scheduling algorithm in the presence of register pressure.[20] We did not seek an optimal scheduling algorithm: the problems we need to

[18] That's only half of the branches encountered: this is because our benchmarks do not have a 100% code coverage, only a part of each benchmark code is actually executed.

[19] CSE3 is a common subexpression elimination that analyzes accross branches [Monniaux and Six 2021].

[20] As the authors of that article rightly point out, optimality when solving the combinatorial optimization problem does not necessarily translate into optimal runtime behavior, because runtime behavior depends on other compiler phases and microarchitectural aspects that are not reflected in the model in which optimization is performed.

solve are NP-complete and optimal algorithms therefore have intolerable worst-case complexity. Furthermore, Six et al. [2020] observed that, for postpass scheduling, their optimal and costly algorithm yields a better makespan than heuristics only in a very small fraction of the cases, and the makespan is then only marginally better.

Necula [2000] and Tristan et al. [2011] previously established that symbolic simulation is effective to validate state-of-the-art compilers. While, it seems difficult to turn their powerful debuggers into a fully formally-correct checker, our approach could be extended to certify more complex optimizations. A few next steps: support a more general notion of block, support simulation modulo invariants at block entry (and in particular global renaming of registers), include an alias-analysis.

ACKNOWLEDGMENTS

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

We also wish to thank Benoît Dupont de Dinechin, Justus Fasse and Xavier Leroy, for their helpful remarks during this work. Justus contributed to develop Asmblock and the Asmblock-to-Asm translation for AArch64 during his L3 internship at UGA.

REFERENCES

Thomas Ball and James R Larus. 1993. Branch prediction for free. *ACM SIGPLAN Notices* 28, 6 (1993), 300–313.

Timothy Bourke, Lélio Brun, Pierre-Évariste Dagand, Xavier Leroy, Marc Pouzet, and Lionel Rieg. 2017. A formally verified compiler for Lustre. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 586–601. https://doi.org/10.1145/3062341.3062358

P. P. Chang and W. W. Hwu. 1988. Trace Selection for Compiling Large C Application Programs to Microcode. In *Proceedings of the 21st Annual Workshop on Microprogramming and Microarchitecture* (San Diego, California, USA) *(MICRO 21)*. IEEE Computer Society Press, Washington, DC, USA, 21–29.

Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Bo Sørensen, Peter Wägemann, and Simon Wegener. 2016. TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research. In *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016) (OpenAccess Series in Informatics (OASIcs), Vol. 55)*, Martin Schoeberl (Ed.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2:1–2:10.

Joseph A. Fisher. 1981. Trace scheduling: A technique for global microcode compaction. *IEEE transactions on computers* 7 (1981), 478–490.

J. R. Goodman and W.-C. Hsu. 1988. Code Scheduling and Register Allocation in Large Basic Blocks. In *ACM International Conference on Supercomputing 25th Anniversary Volume* (Munich, Germany). Association for Computing Machinery, New York, NY, USA, 88–98. https://doi.org/10.1145/2591635.2667158

Wen-mei Hwu, Scott Mahlke, William Chen, Pohua Chang, Nancy Warter, Roger Bringmann, Roland Ouellette, Richard Hank, Tokuzo Kiyohara, Grant Haab, John Holm, and Daniel Lavery. 1993. The Superblock: An Effective Technique for VLIW and Superscalar Compilation. *The Journal of Supercomputing* 7 (05 1993), 229–248. https://doi.org/10.1007/BF01205185

James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (1976), 385–394. https://doi.org/10.1145/360248.360252

M. Lee, P. Tirumalai, and T. . Ngai. 1993. Software pipelining and superblock scheduling: compilation techniques for VLIW machines. In *[1993] Proceedings of the Twenty-sixth Hawaii International Conference on System Sciences*, Vol. i. 202–213 vol.1. https://doi.org/10.1109/HICSS.1993.270744

David Monniaux and Cyril Six. 2021. Simple, light, yet formally verified, global common subexpression elimination and loop-invariant code motion. In *International Conference on Languages Compilers, Tools and Theory of Embedded Systems*. To appear.

George C. Necula. 2000. Translation validation for an optimizing compiler. In *Programming Language Design and Implementation (PLDI)*. ACM Press, 83–94. https://doi.org/10.1145/349299.349314

Louis-Noël Pouchet. 2012. *the Polyhedral Benchmark suite*. http://web.cs.ucla.edu/~pouchet/software/polybench/

Ghassan Shobaki, Maxim Shawabkeh, and Najm Eldeen Abu Rmaileh. 2013. Preallocation Instruction Scheduling with Register Pressure Minimization Using a Combinatorial Optimization Approach. *ACM Trans. Archit. Code Optim.* 10, 3, Article 14 (September 2013), 31 pages. https://doi.org/10.1145/2512432

Cyril Six, Sylvain Boulmé, and David Monniaux. 2020. Certified and efficient instruction scheduling: application to interlocked VLIW processors. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 129:1–129:29. https://doi.org/10.1145/3428197 arXiv:hal-02185883

Jean-Baptiste Tristan, Paul Govereau, and Greg Morrisett. 2011. Evaluating value-graph translation validation for LLVM. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*. ACM, 295–305. https://doi.org/10.1145/1993498.1993533

Jean-Baptiste Tristan and Xavier Leroy. 2008. Formal Verification of Translation Validators: a Case Study on Instruction Scheduling Optimizations. In *Principles of Programming Languages (POPL)*. ACM Press, 17–27. https://doi.org/10.1145/1328438.1328444 arXiv:inria-00289540

Jean-Baptise Tristan. 2009. *Formal verification of translation validators.* Ph.D. Dissertation. Université Paris 7 Diderot.

Jean-Baptiste Tristan and Xavier Leroy. 2010. A simple, verified validator for software pipelining. In *Principles of Programming Languages (POPL)*. ACM Press, 83–92. http://gallium.inria.fr/~xleroy/publi/validation-softpipe.pdf