Formally Verified Superblock Scheduling

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

Formally Verified Superblock Scheduling

Cyril Six
Kalray S.A.
Montbonnot, France
csix@kalrayinc.com

Léo Gourdin
Univ. Grenoble Alpes, CNRS,
Grenoble INP, Verimag
France
Leo.Gourdin@univ-grenoble-alpes.fr

Sylvain Boulmé
Univ. Grenoble Alpes, CNRS,
Grenoble INP, Verimag
France
Sylvain.Boulme@univ-grenoble-alpes.fr

David Monniaux
Univ. Grenoble Alpes, CNRS,
Grenoble INP, Verimag
France
David.Monniaux@univ-grenoble-alpes.fr

Justus Fasse
Univ. Grenoble Alpes, CNRS,
Grenoble INP, Verimag
France
Justus.Fasse@kuleuven.be

Nicolas Nardino
ENS de Lyon
France
Nicolas.Nardino@ens-lyon.fr

Abstract

On in-order processors, without dynamic instruction scheduling, program running times may be significantly reduced by compile-time instruction scheduling. We present here the first effective certified instruction scheduler that operates over superblocks (it may move instructions across branches), along with its performance evaluation. It is integrated within the CompCert C compiler, providing a complete machine-checked proof of semantic preservation from C to assembly.

Our optimizer composes several passes designed by translation validation: program transformations are proposed by untrusted oracles, which are then validated by certified and scalable checkers. Our main checker is an architecture-independent simulation-test over superblocks modulo register liveness, which relies on hash-consed symbolic execution.

CCS Concepts: • **Software and its engineering** → **Formal software verification**; **Retargetable compilers**; • **Theory of computation** → *Scheduling algorithms*; • **General and reference** → *Performance*; • **Computer systems organization** → Superscalar architectures; Very long instruction word.

Keywords: Translation validation, Symbolic execution, the Coq proof assistant, Instruction-level parallelism.

1 Introduction

In-order processor cores execute assembly instructions in their syntactic order. If one instruction computes a register and the next instruction uses this register, then the core *stalls* until the value computed becomes available, which may take several clock cycles. An optimizing compiler thus reorders instructions to minimize stalling.[1]

CompCert[2] [Leroy 2009a,b] is a compiler for the C programming language with a machine-checked proof of correctness: if compilation succeeds, then the semantics of the assembly code matches that of the source: 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…). CompCert does not reschedule instructions, thus producing suboptimal assembly code for in-order cores.

Motivations. Six et al. [2020] added instruction scheduling and some peephole optimizations to CompCert (thus creating CompCertSched) at the assembly level, in *postpass* (after register allocation and final transformations, Fig. 1 and 2). One of their goals was to form instruction "bundles", to be executed in parallel, for the Kalray KVX, a VLIW (Very Long Instruction Word) processor. However, due to that late position within compilation, they were limited to scheduling inside basic blocks (instruction sequences with one single entry point and one single exit point). Dependencies induced by register reuse may thus prevent finding good schedules. Moreover, the porting effort of their approach is rather high for every architecture.

[1] The alternative is an *out-of-order* core, dynamically reordering instructions. Their complexity and lower predictability (e.g., for bounding worst-case execution time), excludes them from some safety-critical systems.
[2] `compcert.org`, official versions: https://github.com/absint/CompCert

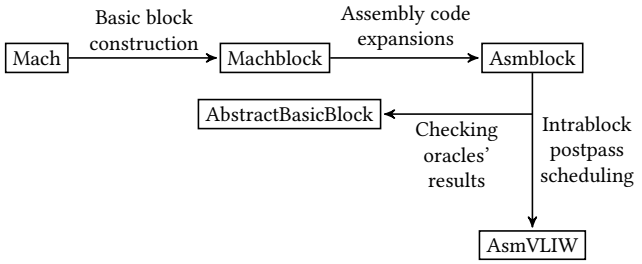

Figure 1. The KVX backend of Six et al. [2020]

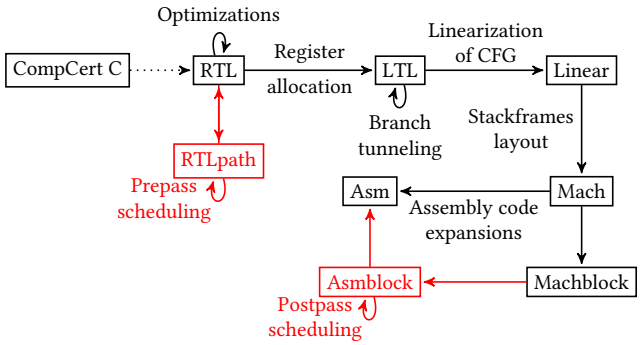

Figure 2. CompCertSched with our Schedulers

Scheduling over superblocks (where instructions may move across branches) is desirable because such blocks are often larger than basic blocks, thus with more opportunities to reorder instructions. Reordering before register allocation, at the RTL level[3], is also desirable as it can take advantage of the infinite number of pseudo-registers, in contrast to postpass scheduling which works with a limited number of registers, some of them having already been spilled. While postpass scheduling (anyway necessary for generating bundles on VLIW processors) benefits from the precise view of the final assembly code, e.g., with register-spilling instructions, prepass scheduling greatly leverages various heuristics that increases semantic parallelism, such as renaming with fresh pseudo-registers after loop unrolling. RTL also provides a representation generic over all processor architectures, thus with a reduced per-architecture development effort.

Contributions and structure of the paper. Section 2 recalls the approach of Six et al. [2020] and how, **as a secondary contribution,** we have ported their postpass instruction scheduler to the AArch64 architecture. This demonstrates which parts of their postpass are generic, and which need to be ported. Our postpass scheduler, like theirs, also performs a peephole optimization just before scheduling (in the scheduler's front-end). But, ours is slightly more advanced (for leveraging AArch64 features). Here, our main result is that we have simply reused their translation validation procedure: this strengthens the case for this approach to verification, which still works after tweaks to heuristics whereas a direct proof would surely require much reengineering.

Our main contribution is a portable verified instruction scheduler, working on a portable intermediate representation (RTL, see Figure 2). It operates over *superblocks*: a generalization of basic blocks, such that each instruction of a given block has still at most one successor in this block, but may also branch to another superblock [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.* Section 3 recalls the specificities of superblock (vs basic-block) scheduling.

In addition, we provide a certified checker for path duplications, which we use to prove tail duplication and loop unrolling optimizations. These optimizations increase scheduling opportunities (at the price of code size increase). Our checker of path duplications illustrates the interest of translation validation designs: a simple formally verified checker enables to certify the correctness of an important class of transformations, modulo small hints provided by oracles. This checker and the heuristics and methods for selecting relevant superblocks are explained in Section 4.

Next, Section 5 summarizes the several preprocessing transformations that are applied on selected superblocks to increase scheduling opportunities. As our prepass scheduling operates on RTL, the pass is technically available for every architecture. However, the scheduling process itself is only useful on in-order cores and needs to be parametrized by a description of the micro-architecture. Thus, our scheduling oracle is decomposed into a front-end that is specific to the architecture, and a generic backend (similarly to Six et al. [2020]). Our verifier is completely generic, but parametrized by rules specific to the target backend. This allows validating some target-specific rewritings in the scheduler's front-end (e.g., the RISC-V expansion of Section 5.2). Hence, similar rewritings could be easily set up for another target.

Section 6 details the implementation of our prepass scheduler's backend. Finally, Section 7 describes how we formally verify both instruction schedulings and rewritings, with a single simulation checker. Because our checker is significantly more powerful than the one of Six et al. [2020] (support of superblocks, rewritings during symbolic execution, simulation

[3]RTL, or "Register Transfer Language", is the intermediate representation on which most optimizations take place within CompCert. Optimizing from RTL to RTL facilitates compatibility, so our work can be easily slotted into other CompCert-related projects.

modulo register liveness), it is an important step forward for *certifying* compilers from symbolic executions. Lastly, Section 8 provides a thorough experimental evaluation of our optimizations, and Section 9 concludes with related works.

Six's PhD [Six 2021] details more in depth many parts of this paper. We use "CompCert" to denote official releases, and CompCertSched for our version of CompCert with scheduling. Our source code is available on our gitlab server, in the `CPP22_main` branch (a release consistent with this paper):

gricad-gitlab.univ-grenoble-alpes.fr/certicompil/compcert-kvx/-/tree/CPP22_main

2 Postpass Scheduler for AArch64

Six et al. [2020] created CompCertSched, a postpass (post register allocation) scheduler and peephole optimizer for CompCert, operating on basic blocks (one entry point, one exit point) in KVX assembly code. First, their peephole optimizer rewrites the assembly code to merge loads (resp., stores) to consecutive locations into double or quadruple loads (resp., stores). Then, a dependency analysis is performed and a scheduler, chosen by a command-line option among several available ones, computes a reordering of instructions. Lastly, the previous transformations being performed by untrusted oracles, the final transformed code is validated against the original one by a certified checker. This checker performs a symbolic execution of both codes, that computes the final contents of registers and of the memory as symbolic expressions over their initial contents. The two codes are considered equivalent if these expressions are structurally equal. For compilation efficiency, structural equivalence of expressions is reduced to pointer equality through *hash-consing* (i.e. memoizing expressions such that two structurally equal expressions are uniquely allocated in each compiler run).[4] Since certain processor operations may trap (e.g., reading from invalid memory locations), the symbolic execution engine also checks that the set of possibly invalid expressions of the original code includes that of the transformed code.

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, 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] encodes such a precondition as a list of potentially trapping terms, hence relaxing the implication of preconditions as a list inclusion.

[4] Thus, Six et al. [2020] distinguishes two notions of equality: first, the physical (or pointer) equality, bound to the OCaml == operator, that compares these expressions by their allocation address in the compiler run; second, the structural one, corresponding to the OCaml = operator and the Coq definitional equality, that recursively compares their abstract syntax.

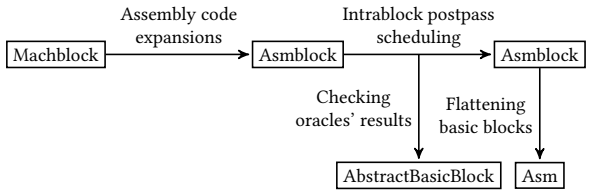

Figure 3. Our AArch64 backend

```
1  ldr  w4, [x6, #0]
2  add  w2, w4, w3
3  ldr  w1, [x6, #4]
4  ldr  w5, [x3, #0]
5  add  w3, w7, w1
6  ldr  w7, [x3, #4]
7  sxtw x3, w0
8  ldr  x19, [sp, #16]
9  ldr  x30, [sp, #8]
10 movz x1, #0, lsl #0
11 str  w2, [x1, #0]
12 movz w0, #0, lsl #0
13 str  w2, [x1, #4]
```

```
1  ldp  w4, w1, [x6, #0]
2  add  w2, w4, w3
3  add  w3, w7, w1
4  ldp  w5, w7, [x3, #0]
5  sxtw x3, w0
6  ldp  x30, x19, [sp, #8]
7  movz x1, #0, lsl #0
8  movz w0, #0, lsl #0
9  stp  w2, w2, [x1, #0]
```

Figure 4. Examples of Load/Store Compactions on AArch64

CompCert provides a backend for the AArch64 architecture (non-VLIW processors) 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-Machblock" of Six et al. [2020] with the passes of "Machblock-to-Asm" of Figure 3. The overall implementation of our formally verified postpass scheduler on AArch64 represents a bit more than three person·months of development. This port for a single architecture represents about 7000 LOC of Coq, with many bureaucratic proofs.

In contrast to the peephole optimizer of [Six et al. 2020], ours, also applied in the scheduler's front-end, can merge non-consecutive loads or stores within the original basic block, as long as they respect the semantic dependencies and offset constraints on double load/store specific to the AArch64 ISA. Figure 4 illustrates four situations found by our peephole, now described from top to bottom. (i) Backward load pairing, with increasing offset (the offset of the second load is *greater* than that of the first one): the pairing must happen backward because, we need to preserve the write into `w4` on line 1, before its read on line 2. (ii) Forward load pairing, with increasing offset: the pairing must happen forward because, we need to preserve the read into `w7` before its write. (iii) Consecutive load pairing, with decreasing offset (the offset of the second load is *lower* than that of the first one). (iv) Forward store pairing, with increasing offset.

Like Six et al. [2020], our formally verified simulation test validates these rewritings by performing the reverse rewriting (i.e. from double loads/stores to pairs of simple loads/stores) in the Asmblock-to-AbstractBasicBlock pass (see Fig. 3). Currently, the main benefit of our peephole optimizations for AArch64 is code size reduction: it reduces

the number of generated memory transfer instructions by about 10%, approximately 3% of the total code length (on average across all our benchmarks). This optimization opens the door for future similar replacements: e.g., selection of `ands` or the `bics` instructions[5].

Our experimental evaluation (see Sect. 8) shows that prepass and postpass schedulings of CompCertSched are complementary, both on KVX and on AArch64.

3 Superblock vs Basic-Block Scheduling

A major improvement of our prepass scheduling over the basic-block scheduling of Six et al. [2020] is that it operates on superblocks (one entry point, possibly several exit points).

The scheduled superblock must *simulate* the original superblock. We check this simulation property with a formally proved *simulation test* summarized in Section 7. Similar to the postpass verifier of Six et al. [2020], this test is based on symbolic execution with formally verified *hash-consing*. Additionally, our symbolic execution *normalizes symbolic values* in order to validate some rewritings of instructions applied in the scheduler's front-end. And, the simulation between superblocks is proved modulo *register liveness*: instead of comparing the symbolic values of *all assigned* registers, we only compare, for a *given exit*, those which are live at this exit (i.e. only those read by any execution starting at this exit). Simulation modulo liveness is both more expressive (since assignments of non-live registers are ignored) and more efficient (since we only compare a small subset of assigned registers).

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 as there are several possible partitions of a given function into superblocks: in particular, on each conditional branch, we may choose to extend the current superblock to one of the successors, or to end it when there is no clearly better choice. Moreover, 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.

This selection pass has a deep impact on the overall performance, since intra-superblock scheduling amounts to optimizing some execution path at the expense of other execution paths. For example, moving the `lws` instruction above the loop exit toward label `.L101` in Figure 8 eliminates a stall at each loop iteration, but slows down the path toward `.L101`. Currently, our superblocks are selected to be the "most likely path" inside the code, and instruction duplications are controlled through compiler options. This is detailed in Section 4.

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

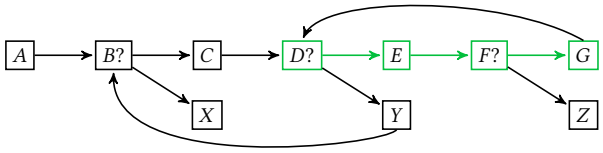

Figure 5. Innermost loop within a superblock

4 Selection and Extension of Superblocks

We describe here how we transform each function into a carefully chosen partition of superblocks. The first step to form superblocks is to identify "*CFG paths*"—called *traces*[6] by Fisher [1981]—likely to be executed. Depending on the accuracy of this *static branch prediction*, our optimizations may result in a performance gain or loss.

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. Finally, the selection of our superblocks for scheduling combines branch predictions (presented in Sections 4.1 and 4.2) and these path duplications (see Sect. 4.3).

4.1 Principles of our Static Branch Prediction

We optionally attach one Boolean prediction per conditional branch: if present it indicates the truth value for the condition that we consider the most likely, and thus the next step in the superblock; if absent, it indicates no prediction at this branch, and thus the end of the superblock.

4.1.1 Detecting Innermost Loops. Fig. 5 depicts a program consisting of two nested loops. The innermost, 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: "`F?`" should be predicted as branching on "`G`", 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. 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.

4.2 Acquiring Prediction Information

Prediction information can be acquired by user annotations, profiling (Sect. 4.2.1) or by static analysis (Sect. 4.2.2). Annotations use `__builtin_expect`, as in GCC and LLVM. Profiling consists in instrumenting the code to record execution statistics into a file. Then, that file is used on subsequent

[6]CompCert already defines a *trace* as a sequence of observational events. Instead, we use "CFG path", where CFG stands for "Control-Flow Graph".

compilations to guide branch prediction. Absent annotations and 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.

4.2.1 Prediction by Profiling. We developed a profiling system in CompCertSched. Classically, our profiling system is used in two steps. First, the program is compiled with special instrumentation: (i) counters are inserted into each object file as local data blocks; the link between the counter and the instruction in the program is given by a hash code depending on the function being compiled and the location within the function, so that counters can be retrieved during recompilation; (ii) right after each branch, a special CompCert builtin is inserted, which increments the appropriate counter; (iii) at program exit, counters are written to a file, through special linker sections added to each compiled object file. The program is then run on representative input, and branch counts are accumulated in a log file.

After profiling information has been recorded, the software is recompiled. The compiler loads the logging file and the profiler-based heuristic consists in assigning the prediction to the branch more taken if the relative difference between the two branches counts exceeds a given threshold.[7]

4.2.2 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. Heuristics are run sequentially, until one of them decides a prediction, otherwise preserving the default "`None`" prediction. (i) 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`. (ii) If a given branch leads to a `return`, then that branch is unlikely to be taken. (iii) 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. (iv) 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 there are a few slightly disappointing corner cases, in particular for heuristic (i) on conditions.

4.3 Path Duplication and Selection from Predictions

After branch prediction, we apply path duplications while selecting superblocks: (i) *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 superblock ending on this join point (see Fig. 6); (ii) *loop body unrolling* consists in unrolling a whole *innermost* loop in order to make its unrolled loop body a big superblock, and thus enable scheduling across the original loop iterations; (iii) *first iteration peeling*, essentially consisting in replacing `while(e) {b}` by `if (e) {b; while(e) {b}}`; (iv) *loop rotation* consists in turning "while-do" loops into "if-do-while", 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" from the loop.

In addition to these selections with duplications, path selection is performed in two other passes of the compiler: (i) for superblock scheduling, the pass "RTL-to-RTLpath" of Fig. 11 involves a superblock selection, which partitions each function into superblocks; (ii) the linearization pass, "LTL-to-Linear" in Fig. 2, also partitions functions into superblocks: it lays out basic blocks that are likely to flow from one to the next in consecutive memory locations, which typically benefits code fetch in processors.

Among these path selection phases, 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 growing it backward (through the "best predecessors"). Then, the path stops, and the next unvisited node is selected to grow a new path. Here are key differences between our implementation and their algorithm. (i) 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`. (ii) 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.

4.4 Formally Verified Checker of Path Duplications

All the duplications of Sect.4.3 are *path duplications*: they duplicate paths in the CFG, while preserving syntactically the CFG structure modulo renaming of nodes. In our compiler, these duplications are perfomed by dedicated *untrusted oracles* followed by a common certified checker. The latter can check in a single run any combination of these transformations. Moreover, it is both simple and small (only 650 lines of Coq, including its correctness proof).

Each oracle is expected to return: (i) the resulting CFG; (ii) the new entrypoint in this new CFG; (iii) a mapping from nodes of the *new CFG* to nodes of the *original* CFG, that we call the *duplicate mapping*, noted ϕ in Fig. 6.

Fig. 6 illustrates path duplication: the original is on the left; the transformed code is on the right; and the duplicate mapping ϕ, in red, indicates which node originated where.

[7] Right now, the threshold is 1: if the execution count of the "if" branch is strictly greater than that of the "else" branch, then the "if" branch is privileged. It would be interesting in the future to change it to a relative threshold instead (e.g., privileging a branch if at least 70% of the executions pass through it).

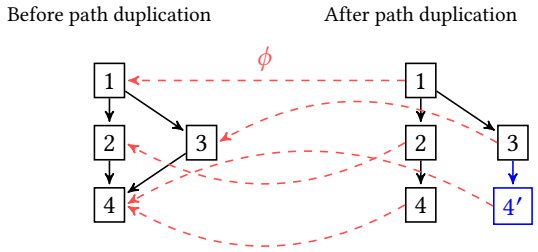

Figure 6. Example of Path Duplication

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 $4'$, 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 $4'$.

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 (Fig. 7). 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).

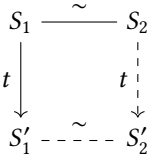

Figure 7. Lockstep

First, our checker verifies that the entrypoint e_2 of f_2 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 a node in f_1's CFG, then n_2 is also a node in f_2's CFG such that: (i) the instruction at node n_1 in f_1's CFG syntactically matches the instruction at node n_2 in f_2's 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); (ii) n_1 (in f_1's CFG) and n_2 (in f_2's CFG) have the same number of successors; (iii) for any i-th successor n_1' of n_1 (in f_1's CFG), the i-th successor n_2' of n_2 (in f_2's CFG) 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. 7.

This checker also accepts oracles that prune unreachable parts of the CFG (from e_1), as long as these pruned nodes do not appear in ϕ, and oracles that negate conditions of conditional branches: in this case, the two successors must also be exchanged. This is useful on architectures such as the KVX where branching costs more than not branching.

Finally, the same checker could also be applied to check code factorizations, because intuitively, it ensures a bisimulation. However, we have not yet formally proved the reverse simulation, nor yet designed a pass leveraging this feature.

5 Rewriting for Better Scheduling

In order to remove some semantic constraints on instruction reorderings, our scheduler's front-end sometimes rewrites instructions into other instructions, which possibly introduces "fresh" registers. The verifier (detailed in Sect. 7) thus checks the simulation by normalizing symbolic expressions and by reasoning modulo register liveness.

5.1 KVX Speculative Loads

In addition to normal load instructions that trap (usually aborting the program) if an unmapped address is accessed, the Kalray KVX provides special load instructions known as "speculative", "dismissible" or "non-trapping", which instead return a default value. In our semantics, these loads return the "undefined" (`Vundef`) value when accessing an incorrect location.[8] Speculative loads can be freely moved before a conditional branch, whereas a normal load cannot unless one can prove that it cannot trap; they are thus very interesting for superblock scheduling.

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

Consider for instance the superblock starting at label `.L100` of Figure 8: 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 post-pass 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 on the update of `$r0`.[9] The gain is of $2n - 1$ cycles where n is the number of loop iterations (there is a 1 cycle loss if there is no 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 a gain of almost 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 should label `.L101` be taken. This is the case, because the scheduler's front-end

[8] Reading from an incorrect location w.r.t CompCert semantics may return an arbitrary value if that memory location is accessible to the CPU, regardless of what was in that location. This is a valid refinement of "undefined".
[9] Only one stall remains in the improved scheduling, because loads have a latency of 3 cycles on the KVX.

```
int sum(int *t, int n) {
  int s=0; for (int i=0;i<n;i++) s += t[i];
  return s;
}
```

Sequence of bundles as emitted by Six et al. [2020] (makespan of 8 cycles)

```
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
```

Effect of our prepass scheduler in between (makespan of 6 cycles)

```
.L100:
 sxwd $r5 = $r4
 compw.ge $r32 = $r4, $r2
;;
 lws.s.xs $r3 = $r5[$r1]
;;
 cb.wnez $r32? .L101
;; // 1 STALL
 addw $r0 = $r0, $r3
 addw $r4 = $r4, 1
 goto .L100
```

Figure 8. Scheduling and bundling a loop body on the KVX

has rewritten the trapping load `lws.xs` into a non-trapping (speculative) load `lws.s.xs`.

We have thus slightly extended RTL to support trapping load instructions. Formally, CompCertSched intermediate representations model these instructions for all architectures. But, they are only selected on KVX (i.e. except for KVX, the compiler will fail to produce assembly code, if an intermediate pass selects them).

5.2 Expanding Operations on RISC-V

Figure 9 presents a fragment of C code and the resulting RISC-V superblock, both for CompCert and for CompCertSched. Registers `x10`, `x11` and `x12` respectively correspond to variables `x`, `y` and `t` of the input program. CompCert does not attempt to minimize pipeline stalls: on line 1, the `lw` instruction dereferencing `x12` in `x7` may induce pipeline stalls at line 3, where `x7` is added to `x10`, with the result written to `x6`. Moreover, 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 RISC-V, `x0` is a read-only register equal to 0). Additionally, 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 RISC-V, our prepass scheduler's front-end performs an expansion of comparisons with immediate (branching or not), and of some other instructions (arithmetic operations on immediates, casts, loads of constants, and length conversions). Intermediate values generated by expansions are stored into fresh pseudo-registers (before register allocation), and the untrusted preprocessor uses a dynamic value numbering system to avoid redundant instructions

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

```
1  lw x7,0(x12)
2  // x7 MAY STALL
3  addw x6,x10,x7
4  addiw x31,x0,7
5  bge x6,x31,.L10
6  addiw x31,x0,7
7  bge x11,x31,.L10
```

```
lw x7, 0(x12)
addiw x12, x0, 7
addw x6, x10, x7
bge x6, x12, .L10
bge x11, x12, .L10
```

Figure 9. On the left: the RISC-V assembly produced by CompCert (3.8). On the right: the RISC-V assembly produced with our prepass' front-end (one `addiw` & one potential `lw` stall have been gained, resulting in a potential gain of two cycles).

(a Common-Subexpression Elimination limited to the superblock, operating on every instruction in the path). Fortunately, in Figure 9, 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 immediates should only have a limited impact on register pressure.

Performing these expansions within the scheduler's front-end 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.

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 that 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). For example, let us consider the following rewriting on RTL conditional branch instructions (written in pseudocode):

$$L_1\colon\ \texttt{if (GEs 7)}[r_1]\ \texttt{goto}\ L_2\ \texttt{else goto}\ L_3 \quad\to\quad \begin{cases} L_1\colon\ r_2 := \texttt{(ADDiwx0 7); goto}\ L_4 \\ L_4\colon\ \texttt{if GE}[r_1, r_2]\ \texttt{goto}\ L_2\ \texttt{else goto}\ L_3 \end{cases}$$

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

$$\texttt{(GEs 7)}[v] \ \to\ \texttt{GE}[v, \texttt{(ADDiwx0 7)}]$$

where v is the symbolic value of r_1.

Moreover, verifying that rewritings of the untrusted preprocessor correctly deal with "fresh" registers is just a particular case of the simulation test modulo liveness: the expansion on the right-hand side of Figure 9 is correct because `x12` is not live at label `.L10`. Hence, embedding the rewriting rules within the symbolic execution allows the proof of these rewrite rules to ignore liveness issues. In contrast, expressed in a preprocessing of the simulation test (like in the postpass of [Six et al. 2020]), rewriting rules would be required to satisfy a bisimulation modulo register liveness.

5.3 Register Renaming

Register allocation may introduce a lot of name dependencies (e.g., write-after-write and write-after-read) by reusing the same register for previously independent pseudo-registers. This may reduce the number of possible schedules. On the contrary, before register allocation name dependencies are rare because "fresh" registers can be generated at any time. This is a big advantage of pre- over postpass scheduling.

However, the path duplications of Section 4.4 may induce name dependencies. In particular, as illustrated on the left-hand side of Fig. 10, loop unrolling produces two almost exact copies of the loop body (shown here in assembly instead of RTL), thus leading to name dependencies between instructions of the original and duplicated loop body. These name dependencies are due to the reuse of register names (e.g., `d1` and `d2` in Fig. 10) and can be removed by renamings with "fresh" names (resp. `d4` and `d3`). We implemented an oracle providing such renamings, and we check the correctness of its results with our certified simulation test modulo liveness over superblocks. Therefore, renamed registers must either not be visible outside the superblock (this is the case in Fig. 10), or else the expected data flow be restored by assigning the correct value to the expected name before exits, at the price of extra-moves.

The result is a superblock-local register renaming pass, which allows for greater scheduling flexibility within the superblock while preserving the expected data flow with respect to the successor superblocks. Note that CompCert's register allocator [Rideau and Leroy 2010], with coalescing and live-range splitting, eliminates most of the extra-moves created by renaming.

5.4 If-Lifting: Moving Up Side Exits in Superblocks

For architectures such as AArch64 which do not provide speculative loads, load instructions cannot move above side-exits without adding a potential trap (which is incorrect). Even after the register renaming on the left-hand side of Fig. 10, this "trap-after-exit" dependency between lines 9 and 7 prevents the desired interleaving of the two occurrences of the initial loop body. A workaround (Fig. 10, right) is to "lift" the side-exit of line 7 above the two arithmetic operations of lines 5 and 6. Interleaving these operations with those of the second body is now possible. On ARM Cortex A53 (dual

```
double sumsq(double *x, int len) {
  double s = 0.0;
  for (int i=0; i < len; i++) s += x[i]*x[i];
  return s;
}
```

```
1  .L101:// loop start
2    ldr  d2,[x0,w2,sxtw #3]
3    add  w2, w2, #1
4    cmp  w2, w1
5    fmul d1, d2, d2
6    fadd d0, d0, d1
7    b.ge .L100
8    // end body 1
9    ldr  d2,[x0,w2,sxtw #3]
10   add  w2, w2, #1
11   cmp  w2, w1
12   fmul d1, d2, d2
13   fadd d0, d0, d1
14   b.lt .L101
15   // end body 2
16 .L100:// loop exit
17   // only d0 is live here
18
```

```
.L101:
  ldr  d2,[x0,w2,sxtw #3]
  add  w2, w2, #1
  cmp  w2, w1
  b.ge .L102
  ldr  d3,[x0,w2,sxtw #3]
  add  w2, w2, #1
  fmul d1, d2, d2
  cmp  w2, w1
  fmul d4, d3, d3
  fadd d0, d0, d1
  fadd d0, d0, d4
  b.lt .L101
  b .L100
.L102:
  fmul d1, d2, d2
  fadd d0, d0, d1
.L100:
```

Figure 10. Interleaving of unrolled loop-bodies on AArch64.

issue, where each of these operations takes 6 cycles), this reduces the initial makespan from 25 to 21 cycles (a gain of 16%). The price to pay is the compensation code at the "fresh" label `.L102` which duplicates these two arithmetic operations for when the side-exit is taken. The correctness of this transformation is verified by combining existing verifiers of CompCertSched (modulo minor extensions).[10] See [Justus Fasse 2021] for details.

6 Oracle of the Superblock Scheduler

After superblock selection, each superblock is scheduled separately. Our superblock scheduling oracle, after preprocessing the superblock according to Section 5, schedules the transformed superblock by extending the principles of Six et al. [2020] for basic block scheduling. The scheduling problem of a given superblock is expressed through a system of constraints, where each instruction j, including exit branches, is associated to a time-slot $t(j)$ expressing the number of clock cycles at which it is estimated that this instruction is executed. Constraints on $t(j)$ are of two kinds: resource constraints (i.e. feasible allocations of pipeline units) and latency constraints (i.e. semantic and time dependencies between instructions). In order to find schedules that preserve the semantics in presence of side-exits, we need to extend the latency constraints of Six et al. [2020], with latency constraints specific to branching instructions: (i) we represent live variables at side exits, by extra reads of the branch instruction; (ii) we forbid trapping instructions to move above side-exit, by making each trapping instruction

[10] However, it is currently implemented through an intricate combination of passes: it is thus not yet integrated in our mainline CompCertSched. Its code is in the `CPP22_if_lifting` branch
gricad-gitlab.univ-grenoble-alpes.fr/certicompil/compcert-kvx/-/tree/CPP22_if_lifting

depend on the preceding side-exit with a latency of 1 cycle; (iii) similarly, we forbid side-exits to be reordered, through a latency of 1 between successive side-exits.

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 usage 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 RISC-V, for a limited number of cores (Cortex-A53, Cortex-A35; Rocket, SweRV EH1, SiFive U74), 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 resource 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 operand reads at different cycles). 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.[11] 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*: the time when the last value produced by the basic block is available. In contrast, superblock prepass optimization has multiple, conflicting objectives: (i) reducing the makespan (with respect to the final exit); (ii) reducing register pressure (the number of physical registers needed), or, as a proxy, the live ranges of values; (iii) pushing side-exits as early as possible.

For solving the constraint system, we reuse two algorithms of [Six et al. 2020]. (i) *Forward list scheduling*: time slots are greedily filled by increasing time (clock cycle), adding instructions to each slot as long as they respect the resource and latency constraints, with a priority for instructions that start 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. (ii) *Backward list scheduling*: time slots are greedily filled by decreasing time, with a priority for instructions that end the longest path in the latency graph.

A difficulty not present in postpass scheduling is that scheduling some instructions early, whereas their result is not used soon, increases the number of live pseudo-registers; the risk is that a superblock that originally used less pseudo-registers than the number of physical registers in the processor could, after scheduling, use more, resulting in costly *spill code* (loads and stores to stack slots) being inserted by the register allocation pass. The risk of exceedingly early scheduling is higher with the forward list scheduler than with the backward scheduler, thus we originally favored the latter for prepass scheduling. In addition, we introduced: (iii) *Zigzag scheduling*: forward scheduling is used to place exit points, and then backward scheduling places other instructions. (iv) *Register-pressure-aware scheduling*: the forward list scheduler is made aware, at each allocation step, of the number of live pseudo-registers in each register class (e.g., integer vs floating-point registers). When this number becomes close (e.g., the difference is less than three) to the number of physical registers in the processor, the strategy is modified: the scheduler favors instructions which decrease the number of live registers in that class, and chooses the one which decreases it the most. If no instruction decreases register pressure, we favor instructions which do not increase the number of live registers of the given class, with long paths to the exit in the latency graph. Finally, if all possible instructions increase the number of live registers, we wait for an instruction that does not, and if none becomes available after 5 cycles, we schedule the next instruction with the normal list scheduling strategy (priority for instructions that finish the longest path in the latency graph). When the number of live registers drops below the threshold, the normal list scheduling strategy applies again. See [Nicolas Nardino 2021] for details.

7 Formally Verified Superblock Scheduler

We now describe how our superblock scheduler is formally verified. For the sake of concision, this description remains informal, and in particular, without Coq definition. Full details are given in [Six 2021, Chap. 5&7] and in our Coq code (see the URL given on page 3).

Figure 11 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. Then, the untrusted scheduler

[11] The cycle count for division often depends on the quotient bit-length.

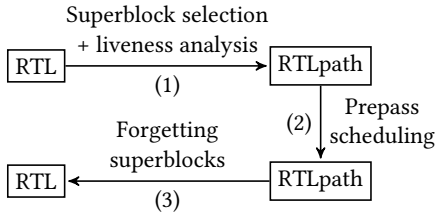

Figure 11. Our Formally Verified RTL Superblock Scheduler

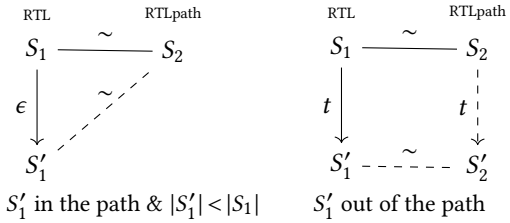

Figure 12. RTL to RTLpath

(Sect. 6) transforms each superblock, which are checked for simulation by the verified checker, described in this section. 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.

7.1 Definition of the RTLpath IR

RTLpath extends the RTL CFG 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". To each instruction we assign an optional default successor. Instructions with a default successor are either basic instructions, or conditional branches.[12] A CFG path connects a sequence of instructions to their default successors; the final instruction of a path has no default successor. The whole execution of a path of length p emits at most one observational event, in the CompCert semantic sense (call to an external function, access to a volatile variable): at its final node (this is important for forward simulation proofs). In fact, all functions calls are restricted to be at the final node. Note that one given CFG path represents several *execution paths*: the execution may exit early from the CFG path through an intermediate conditional branch.

RTLpath comes with a semantics: one step of RTLpath execution runs all RTL instructions from path entry to path exit. States are the same as for RTL, but store as well that all caller functions are well-formed RTLpath functions.

[12]For conditional branches, the default successor is the one that "continues the sequence" (i.e. the "else" branch). Indeed, it has the smallest latency in usual pipelines. It also corresponds to the one predicted in Sect. 4: conditions have been negated if necessary, as explained at the end of Sect. 4.4.

7.2 Bisimulation of RTLpath and RTL Executions

With these definitions, the forward simulation of RTLpath by RTL is reduced to a simple "Plus-simulation" (Fig. 13): an RTLpath step runs at least one RTL instruction, but at most $p + 1$ where p is the size of the executed path.

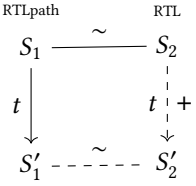

Figure 13. Back to RTL

The reverse simulation, of RTL by RTLpath, is less trivial. In an 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 (recall that a call instruction must be last in a path, so its return location is at the start of another path). 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 12, one RTL step is simulated by one RTLpath step on path exits. Otherwise, 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.

7.3 Path-Executions Equivalence Modulo Liveness

Given the original RTL program, the oracle pictured in pass (1) of Figure 11 produces an RTLpath program, annotated with a set of "input" (i.e. live) registers on each path entry. The well-formedness of this RTLpath program is checked by a certified simple forward analysis, that simultaneously checks the correctness of liveness information: the set of *input registers* given on each path entry must include the sets of registers actually read in all executions starting from this path entry.

For RTLpath programs with correct liveness information, we prove a "lockstep" simulation (see Fig. 7) 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 those that have been stored in the stack during function calls. Since RTLpath states (like RTL ones) comprise a well-formed list of stack frames modeling the call stack, it is easy to define this simulation relation that both involves registers and memory.

7.4 Design of the Certified RTLpath Scheduler

The scheduling oracle takes as input an RTLpath function, computes a schedule for each superblock (each path), and 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, and `dm` is the reverse mapping from entry paths of the scheduled CFG to the original CFG (like the duplicate mapping ϕ in Sect. 4.4). Given an original RTLpath function `f1`, our

Symbolic state for the left-hand side of Fig. 8

```
if (r4 ≥ r2) {
  r32 := (r4 ≥ r2);
  goto .L101
}
ok[lws.ws(sxwd(r4),r1)];
  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
```

Symbolic state for the right-hand side of Fig. 8

```
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
```

Figure 14. Comparing Symbolic States of the KVX Superblocks given in Figure 8.

certified pass turns the (c, e, pm) returned by the oracle into an RTLpath function `f2`, after verifying the well-formedness conditions (like in Sect. 7.3). Then, thanks to the `dm` mapping, it checks that `f2` "matches" `f1`: (i) `dm` is a mapping from path entries of `f2` to path entries of `f1`; (ii) for each path entry `pc2` of `f2`, given `pc1` its matching path entry in `f1`, 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` (with the simulation checker detailed in Sect. 7.5). Formally, the forward simulation proof of our scheduler also corresponds to a lockstep simulation, similar to the one of Sect. 7.3.

7.5 Certifying the RTLpath Simulation Test

Fig. 14 illustrates the comparison of symbolic states for validating the transformation of Fig. 8. Each symbolic state is a sequence of conditional exits followed by a final one. The state of memory and registers on each exit is represented by a *preconditioned parallel assignment* (following the terminology of Example 2.1), where the precondition is a list of possibly trapping expressions, within the "**ok**" keyword in Fig. 14. Our simulation test checks the implication of preconditions on each exit by testing list inclusion: here, the lists of the right-hand side on both exits are implicitly empty. And the simulation test compares the symbolic values of live registers on each exit: here, only `r0` is live at exit `.L101`, whereas only `r0`, `r1`, `r2`, and `r4` are live at exit `.L100`.[13]

Our formal development follows the general lines of that of [Six et al. 2020]: we 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, extended with our rewriting engine.

We retain their approach of modeling the impure computations involved in hash consing in a monad of "impure" computations [Boulmé 2021]. The core of our simulation test is implemented in this monad and proved correct by a lemma, expressing that when the test normally terminates then property (ii) of Section 7.4 holds. The primary goal of their refinement technique is to circumscribe the reasoning on impure computations as much as possible.

However, our abstract model of the simulation test is more complex than the one of [Six et al. 2020] on several points. (i) We consider *superblocks*: the symbolic state thus must represent several execution paths (where [Six et al. 2020] only deal with a single execution path). (ii) Our simulation test compares symbolic states on each path exit modulo liveness. (iii) Our notion of RTL/RTLpath state is richer than their notion of Asm/Asmblock state. Note that, since call instructions are always at the end of a path, return addresses in the call stack must always point to the start of a path. (iv) In CompCert's memory model, a pointer comparison fails when these pointers are allocated in distinct memory blocks.[14] Hence, arithmetic operators that may compare pointers (comparisons of integers of the same size as the pointers) 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 works on the last IR of CompCert (assembly), but not in an intermediate IR such as RTLpath. Alternatively, we solved this issue by proving that the memory within a path execution is only modified by store instructions, which do not change the results of pointer comparisons: thus, all pointer comparisons do not really depend on the current memory, but only on the initial memory of the path execution.

7.6 Verified Rewritings during Symbolic Execution

In contrast to rewriting rules of Six et al. [2020], which happens during the "Asmblock-to-AbstractBasicBlock" compilation, ours are directly integrated in the symbolic execution engine. As explained in Section 5.2, our approach gives simpler proofs about these rewriting rules, but makes the symbolic execution implementation a little trickier. In particular, we have to integrate them within the hash-consing mechanism.

Our rewriting rules are always applied at the top of hash-consed terms (like a smart constructor). The rewritten terms

[13] Superblock of Fig. 8 is a loop of entry `.L100` and exit `.L101`, where `r0` is the result of the loop, `r4` is the loop counter, and where `r1` and `r2` are parameters.

[14] Comparison of pointers pointing to different blocks has undefined behavior according to the C standard [ISO 2011, §6.5.8p5].

are then turned into proper hash-consed terms by a dedicated (impure) function that only transforms their top nodes. In other words, the management of hash-consing during rewriting is delegated to this dedicated function: this both simplifies the implementation of rewriting rules and its formal proof.

Formally, a rewriting rule must turn a term t_1 into a term t_2 such that for all register and memory states, if evaluation of t_1 does not fail, then t_2 evaluates to the same value as t_1. This allows us to rewrite t_1 as t_2 in *parallel assignments* but not in *preconditions*. For example, as illustrated in Fig. 14, rewriting `lws.ws` operation into `lws.s.ws` suffices for our certified simulation test to validate the transformation in Fig. 8.

As explained in Section 5.2, on the RISC-V backend, we succeeded to move most of the assembly expansions expressed at the "Mach-to-Asm" pass in CompCert (see Fig. 2) as rewriting rules on RTLpath. This required us to overcome a little issue: while the forward simulation proof of "Mach-to-Asm" supports that expansions replace the `Vundef` value by any other value, this is not supported in the proof of our rewriting rules. In CompCert, `Vundef` represents a potential undefined behavior that has not yet been observed (e.g., read into an uninitialized register): it is a poison value which does not abort computation until it becomes observable. In our case, `Vundef` may appear when evaluating some RISC-V macro-operations on unexpected immediate arguments (e.g., the `shrximm` macro-operation used to model division by a power of two, expansed into a sequence of right shifts and additions). The "Mach-to-Asm" expansion typically replaces `Vundef` by a silent arithmetic overflow.

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 ISAs, such as 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 RISC-V, expansions that involve stack-accessing instructions cannot really be expressed at RTL level, because the layout of stack frames is not yet defined at this level (see Fig. 2): stack accesses are only handled very abstractly.

8 Experimental Evaluation

Tables 1 to 3 summarize our experiments on several architectures: **AArch64** is an ARM Cortex A53 (AArch64) in a Raspberry Pi 3 running Ubuntu 18.04.5 LTS;[15] **KVX** a Kalray KV3 "Coolidge" core; **RISC-V** a SiFive U74 RISC-V dual-issue, in-order core in a HiFive Unmatched board. In each case, we tie the process to one core of the machine, and measure clock cycles using hardware counters. We run different suites: the benchmarks of [Six et al. 2020], the computational oriented Polybench [Pouchet 2012], and the embedded systems oriented TACLeBench [Falk et al. 2016].

First, Table 1 measures the cumulative impact of each gradually introduced optimization compared to CompCert 3.8.[16] Note that postpass scheduling and peephole optimizations apply only to AArch64 and KVX, and RTL expansion only applies to RISC-V. The postpass scheduling for KVX is the one of [Six et al. 2020]; LICM (Loop Invariant Code Motion) is the one of [Monniaux and Six 2021]. The Q1, Med and Q3 values respectively denote the first quartile, the median, and the third quartile on the entirety of our benches. Here are a few conclusions. (i) 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. (ii) LICM is another meaningful optimization, producing a gain of 20% on some benchmarks. (iii) Prepass scheduling (without any loop unrolling) also helps, increasing by 5-10% for the AArch64 and RISC-V cores. This is mostly due to removing the false dependencies (compared to the postpass scheduling). In contrast, the KVX core, with 64 user registers, is barely affected (in absence of optimizations for building large superblocks such as loop unrolling). Using both prepass and postpass scheduling together on AArch64 is the best setting, mainly because the latter fine-tunes the placement of spills and other instructions that have been expanded between RTL and Asm. (iv) Tail duplication and loop unrolling, combined with prepass scheduling, increase performance by about 5% on KVX. Those are also useful on specific benches thanks to their ability to enlarge the size of superblocks. (v) Loop rotation used alone has a small impact on RISC-V, but the postpass (on AArch64) benefits from it as the rotation may provide more scheduling opportunities. It shines mostly for the KVX architecture, since it results in a more efficient bundling of the loop header in postpass. (vi) RTL expansions (RISC-V only) have no significant impact on average (for all benchmarks) but they can, on some programs, result in a large performance gain when combined with prepass scheduling. We observe an improvement of 4% on the third quartile.

We also compared our best version of CompCertSched to GCC. Table 2 shows the gain (can be negative) of using GCC with the given optimization flag, versus our best version of CompCertSched, for the three suites.

[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 CompCert, we use the one from [Six et al. 2020] without postpass scheduling (nor bundling).

Table 1
Improvement of Cumulated Optimizations wrt CompCert on 3 Benchmarks suites

	AArch64			KVX			RISC-V		
Setup	Q1	Med	Q3	Q1	Med	Q3	Q1	Med	Q3
+Peephole & Postpass	+4.6%	+13.7%	+21.8%	+16.5%	+32.6%	+54.5%	-	-	-
+LICM	+7.4%	+21.9%	+42.9%	+19.3%	+41.1%	+65.9%	+0.1%	+3.8%	+10.0%
+Prepass (List)	+10.4%	+27.5%	+47.8%	+19.7%	+43.9%	+69.0%	+5.6%	+14.0%	+30.0%
+Tail dupl.	+13.5%	+28.1%	+47.7%	+20.7%	+43.1%	+70.1%	+5.4%	+14.3%	+30.8%
+Loop unroll.	+16.1%	+30.1%	+62.3%	+23.3%	+47.5%	+88.1%	+6.5%	+13.6%	+31.7%
+Loop rotate	+18.2%	+36.0%	+63.5%	+23.7%	+53.7%	+97.3%	+6.9%	+13.5%	+32.2%
+RTL expans. ("Best")	-	-	-	-	-	-	+6.8%	+14.3%	+36.1%

Table 2
Improvement of GCC wrt "Best CompCertSched" on 3 Benchmarks suites

GCC-O1	-5.8%	+0.9%	+12.5%	-20.6%	-4.0%	+10.8%	-3.4%	+6.9%	+20.0%
GCC-O2	+6.2%	+15.6%	+39.9%	+4.7%	+21.5%	+62.6%	+17.7%	+29.9%	+67.1%

Table 3
Improvement of Alternative Prepass Scheduling wrt "Best" with List Scheduling on 3 Benchmarks suites

"Best" & Regpres	+17.4%	+36.6%	+66.4%	+23.7%	+53.7%	+97.3%	+7.5%	+14.4%	+37.2%
"Best" & Backward	+16.2%	+34.1%	+62.0%	+23.6%	+52.3%	+91.1%	+6.7%	+15.7%	+33.6%
"Best" & Zigzag	+16.7%	+32.0%	+66.3%	+23.3%	+50.8%	+96.6%	+7.3%	+14.8%	+34.6%

We are getting closer to GCC -O2 with our optimizations on AArch64 and KVX, and we are better than GCC -O1 in most cases on KVX. On RISC-V, we still have a margin of progression: we suspect the lack of postpass scheduling and of strength reduction. RISC-V needs several instructions for operations, such as accesses to consecutive indices in an array, feasible in one instruction on KVX or AArch64 and also strength-reducible, thus the lack of strength reduction is more evident on a RISC-V core than on architectures with more powerful individual instructions. More generally, less work went into the RISC-V backend.

Another experiment, in Table 3, 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 RISC-V where register-pressure-aware and backward (respectively) seems to give better results.

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 could be profiled, only 5% of them were wrongly predicted, and 14% of them had a pattern not caught by our used heuristics.[17] This gives us confidence in our static branch prediction.

We also measured that the preprocessing pass exploiting non-trapping loads on KVX yields an average gain of about +1.5% on benchmarks where replacements are made. This score would possibly be higher if followed by an if-conversion pass.

Finally, we have also measured the effect of our experimental "if-lifting" optimization of Section 5.4 (together with register renaming) on AArch64. On most benches, it has little effect with only a few percent of performance gain or loss when added to "Best". Nonetheless, on some, we observe a marked improvement of more than five percent, for a code size increase below 5%. Interestingly, the efficiency gain on TACLeBench's Susan exceeds 30%.

Our verified checkers ensure that if code compiles successfully, then it is compiled correctly. However, there is the possibility that the scheduler, peephole optimizer, expander…are incorrect and code is rejected by the checker. We thus test CompCertSched on benchmarks, applications, as well as hundreds of random programs generated by compiler fuzzers, such as CSmith[18] [Yang et al. 2011] and YarpGen [19] [Livinskii et al. 2020]. CSmith found one bug in the AArch64 peephole optimizer, which produced an incorrect replacement which was refused by the verified checker; the bug case was reduced using CReduce[20] [Regehr et al. 2012] and then fixed.

9 Related Works and Conclusions

Remarks on heuristics within compilers. We implement a variant of Ball and Larus [1993]'s static branch predictor, as do GCC (citing [Wu and Larus 1994]) and LLVM [Alovisi 2020]. Profile-guided scheduling, the "ground truth", performs better only on a small minority of examples, thus we do not expect that more advanced static prediction [Deitrich

[17] 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.

[18] https://github.com/csmith-project/csmith
[19] https://github.com/intel/yarpgen
[20] https://embed.cs.utah.edu/creduce/

et al. 1998] would improve the situation. From our experiments, nonprediction is preferable to misprediction: this is why we do not use some of Ball and Larus's heuristics. Similarly, we use heuristics to solve the instruction scheduling optimization problems. Six et al. [2020] observed that, for postpass scheduling, their optimal and costly algorithm based on integer linear programming yields a better makespan than heuristics only in a very small fraction of the cases, and the makespan is then only marginally improved. Shobaki et al. [2013] propose an optimal scheduling algorithm in the presence of register pressure, with high cost. As they rightly point out, optimality when solving the combinatorial optimization problem does not necessarily translate into optimal runtime behavior, because the latter depends on other compiler phases and microarchitectural aspects that are not reflected in the model in which optimization is performed.

State of the art on unverified compilers. As of version 11.1.0, GCC optionally has postpass superblock scheduling (`-fsched2-use-superblocks`) and can enlarge superblocks by tail duplications (`-ftracer`), as CompCertSched. Superblock scheduling was introduced in GCC 3.4 but is still described as "experimental"; it is not active at `-O3`, one has to pass the specific option. An option for trace scheduling (`-fsched2-use-traces`) exists but seems to be a dummy (trace scheduling is not in the list of active passes listed by `-fverbose-asm`); it appears to have been introduced in GCC 3.4 and deactivated later. GCC has an "interblock" prepass scheduler based on region scheduling [Gupta and Soffa 1990].

As of LLVM version 12.0.0, `MachineScheduler` and `MachinePipeliner` operate on basic blocks only. Swing modulo scheduling, a software pipelining technique, and an extension to superblocks were implemented for LLVM in 2005 [Lattner 2005] but this work was not integrated into official releases. Recently, [Rangasamy 2021] proposed a superblock scheduler for LLVM. They neither aim at supporting code motion below side exits with compensation code, nor do they currently take liveness into account in checking if code motion is valid without compensation code.

Trace vs. Superblock scheduling. Trace scheduling, as introduced by Fisher [1981], is more general than superblock scheduling: it allows for side entrances in addition to side exits. Instruction may then be moved above and below both side entrances and exits. Generating the necessary compensation code, especially when branches are moved across each other, is however complex.

Superblock scheduling [Hwu et al. 1993; Lee et al. 1993] is presented as a simpler alternative. By disallowing side-entrances, and movements of branches across one another, only the comparatively simple compensation code for moving instructions below side exits (e.g., "if-lifting" of Sect. 5.4) is necessary [Gregg 2001]. Moreover, the loss of side entrances may be compensated by tail duplication. Gregg [2001] actually argues that superblock scheduling gives comparable results to full trace scheduling while being much simpler to implement. We add that *superblocks*, in contrast to *general traces*, allows for a compositional reasoning on scheduling correctness, which makes formal proofs much easier.

Comparison with verified compilers. Tristan and Leroy [2008] certified a simplified postpass trace scheduling within CompCert (at the Mach level): they disallow moves of branches across each other and, because they do not consider register liveness, they systematically duplicate instructions (without register renaming). Moreover, their implementation suffered from exponential complexity [Tristan 2009, §6.7.1][Tristan and Leroy 2008, §7], and was never integrated into official releases of CompCert.

In contrast, our certified superblock scheduling leverages most of the power of superblock scheduling while remaining efficient even for large superblocks. Moreoever, it composes with existing RTL-level optimizations. We thus apply CompCert optimizations such as CSE3,[21] constant propagation or deadcode elimination, between our code duplications and the actual scheduling. We intend to extend our approach to certify more complex optimizations.

Apart from CompCert, the only other real-scale verified compiler backend is CakeML, which does not include an instruction scheduler. An alternative is to use a normal compiler and apply post-hoc translation validation. For example, Necula [2000] and Tristan et al. [2011] previously established that symbolic simulation is effective to validate state-of-the-art compilers. However, it seems difficult to turn their powerful debuggers into formally verified checkers. Alternatively, a translation validation approach by SMT-solving, applied from source to object code without modification to the compiler, was used for the seL4 secure operating system kernel [Sewell et al. 2013], with the GCC compiler and the ARM architecture, with some restrictions as to the form of the C programs to be compiled and the level of optimization. The fact that this approach was only used for one specific program seems to indicate that there are difficulties in applying it more widely.

Acknowledgments

This work has been partially supported by the LabEx PERSYVAL-Lab (ANR-11-LABX-0025-01) and the IRT Nanoelec (ANR-10-AIRT-05), funded by the French national program "Investissement d'Avenir".

We also wish to thank Delphine Demange, Benoît Dupont de Dinechin, Frédéric Pétrot, Xavier Leroy, and anonymous referees for their helpful advices.

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

References

Pietro Alovisi. 2020. *Static Branch Prediction through Representation Learning*. Master's thesis. KTH Stockholm. https://www.diva-portal.org/smash/get/diva2:1450658/FULLTEXT01.pdf

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

Sylvain Boulmé. 2021. *Formally Verified Defensive Programming (efficient Coq-verified computations from untrusted ML oracles)*. Habilitation Thesis. Université Grenoble Alpes. https://hal.archives-ouvertes.fr/tel-03356701

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.

Brian L. Deitrich, Ben-Chung Cheng, and Wen-mei W. Hwu. 1998. Improving Static Branch Prediction in a Compiler. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques, Paris, France, October 12-18, 1998*. IEEE Computer Society, 214–221. https://doi.org/10.1109/PACT.1998.727253

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.

David Gregg. 2001. Comparing Tail Duplication with Compensation Code in Single Path Global Instruction Scheduling. In *Compiler Construction, 10th International Conference, CC 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings (Lecture Notes in Computer Science, Vol. 2027)*, Reinhard Wilhelm (Ed.). Springer, 200–212. https://doi.org/10.1007/3-540-45306-7_14

Rajiv Gupta and Mary Lou Soffa. 1990. Region Scheduling: An Approach for Detecting and Redistributing Parallelism. *IEEE Trans. Software Eng.* 16, 4 (1990), 421–431. https://doi.org/10.1109/32.54294

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

ISO. 2011. *C11 Standard*. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf ISO/IEC 9899:2011.

Justus Fasse. 2021. *Code Transformations to Increase Prepass Scheduling Opportunities in CompCert*. Master Thesis of Science. Université Grenoble Alpes. https://www-verimag.imag.fr/~boulme/CPP_2022/FASSE-Justus-MSc-Thesis_2021.pdf

Tanya M. Lattner. 2005. *An Implementation of Swing Modulo Scheduling with Extensions for Superblocks*. Master's thesis. Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL. https://llvm.org/pubs/2005-06-17-LattnerMSThesis.html

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

Xavier Leroy. 2009a. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009). https://doi.org/10.1145/1538788.1538814

Xavier Leroy. 2009b. A formally verified compiler back-end. *Journal of Automated Reasoning* 43, 4 (2009), 363–446. http://xavierleroy.org/publi/compcert-backend.pdf

Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random testing for C and C++ compilers with YARPGen. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 196:1–196:25. https://doi.org/10.1145/3428264

David Monniaux and Cyril Six. 2021. Simple, light, yet formally verified, global common subexpression elimination and loop-invariant code motion. In *LCTES '21: 22nd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems, Virtual Event, Canada, 22 June, 2021*, Jörg Henkel and Xu Liu (Eds.). ACM, 85–96. https://doi.org/10.1145/3461648.3463850

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

Nicolas Nardino. 2021. *Register-Pressure-Aware Prepass-Scheduling for CompCert*. Bachelor Thesis of Science. ENS de Lyon. https://www-verimag.imag.fr/~boulme/CPP_2022/NARDINO-Nicolas-BSc-Thesis_2021.pdf

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

Arun Rangasamy. 2021. Superblock Scheduler for Code-Size Sensitive Applications. Slides presented at LLVM developers' meeting. https://llvm.org/devmtg/2021-02-28/slides/Arun-Superblock-sched.pdf

John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, Jan Vitek, Haibo Lin, and Frank Tip (Eds.). ACM, 335–346. https://doi.org/10.1145/2254064.2254104

Silvain Rideau and Xavier Leroy. 2010. Validating register allocation and spilling. In *Compiler Construction (CC 2010) (LNCS, Vol. 6011)*. Springer, 224–243. http://gallium.inria.fr/~xleroy/publi/validation-regalloc.pdf

Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. 2013. Translation validation for a verified OS kernel. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 471–482. https://doi.org/10.1145/2491956.2462183

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. 2021. *Optimized and formally-verified compilation for a VLIW processor*. Ph.D. Dissertation. Université Grenoble Alpes. https://hal.archives-ouvertes.fr/tel-03326923

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

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

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

Youfeng Wu and James R. Larus. 1994. Static branch frequency and program profile analysis. In *Proceedings of the 27th Annual International Symposium on Microarchitecture, San Jose, California, USA, November 30 - December 2, 1994*, Hans Mulder and Matthew K. Farrens (Eds.). ACM / IEEE Computer Society, 1–11. https://doi.org/10.1109/MICRO.1994.717399

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Programming Language Design and Implementation (PLDI)*. ACM Press, 283–294. https://doi.org/10.1145/1993498.1993532