

Master of Science in Informatics at Grenoble
Master Informatique
High-confidence Embedded and Cyberphysical Systems

Code Transformations to Increase Prepass Scheduling Opportunities in CompCert

Slightly revised post-defense report

Justus Fasse

December 6, 2021

Research project performed at Verimag* under the supervision of
David Monniaux & Sylvain Boulmé

The original thesis was defended before a jury composed of:

Laurence Pierre	President
Akram Idani	Examiner
Yannick Zakowski	External Expert

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

This internship complements previous work at VERIMAG on prepass superblock instruction scheduling in COMPCERT. A superblock-local register renaming pass, that does not require any modifications to the existing translation validation approach, was added. In addition, a technique to move code below side exits of superblocks, composed of several independent passes, was implemented. The semantics of the intermediate representation used for superblock scheduling had to be tweaked, requiring changes down to the hash-consed implementation of the symbolic execution test. With the exception of one lemma, these adaptions have been proven in COQ. These two contributions decrease the amount of constraints on prepass instruction scheduling, making it more powerful. A proof-of-concept oracle implementation demonstrates that selectively moving code below side exits can improve the schedules produced by prepass scheduling dramatically for a few benchmarks, while in general not negatively impacting the performance of others.

Contents

1 Introduction

1.1 CompCert and certified compilation

Compilers are an essential building block of virtually all software systems[1]. With rare exceptions it is much preferable to rely on the compiler's ability to analyze, transform and optimize programs than writing assembly manually. Unsurprisingly, compilers tend to be large and complex programs themselves. While a compiler's foremost responsibility is to generate *correct* code [16, pp. 31, 314], it has proven extraordinarily difficult to achieve this goal in practice. Despite the extensive review and testing regiment applied to mainstream compilers such as gcc and clang/LLVM, miscompilation still regularly occurs. For most code, where the software quality of the input program usually falls far short of that of the compiler, this is not a major issue.

Critical systems that require high-assurance software on the other hand may not be satisfied by placing this much trust in the compiler. Indeed, "in the past for highly critical applications compiler optimizations were often completely switched off" [23]. CompCert [27, 26] offers a compelling alternative: They give formal semantics to a dialect of C—CompCert C, which comprises a large subset of C99 and some C11 extensions—and the targeted assembly languages. These specifications, as well as all the intermediate languages are formalized in Coq and the transformations between them are proven correct. That is, if CompCert returns without signalling an error, the generated assembly code *preserves the semantics* of the source program[2]. More precisely, the generated assembly code's runtime behavior is one of the possible behaviors of the source code. This clarification is necessary because some aspects of C are deliberately nondeterministic. CompCert assumes that no undefined behavior is present in the source program.

Specifying the semantics in Coq and proving that the transformations preserve these semantics has been empirically shown to work: Yang, Chen, Eide, and Regehr did not find bugs in the verified parts of CompCert while discovering dozens of errors in both gcc and clang [41]. Earlier work had shown that of thirteen tested production-quality compilers, all of them produced errors compiling the volatile qualifier, noting that such errors are "disturbingly common" [9]. The volatile keyword is often essential in ensuring the correct communication with hardware.

There are two main styles to prove an optimization correct in CompCert: (1) The algorithm performing the transformation is programmed in Gallina, Coq's specification language, and proven correct directly in Coq. (2) Alternatively, an untrusted oracle, usually written in OCaml, can be used if a verified-in-Coq validator is provided that checks the results and only proceeds if the output of the oracle can be shown to preserve the semantics of the input. This approach presents a variation of *translation validation*.

[1] With the distinction to interpreters being arguably blurry.

[2] See section D of "Simple, light, yet formally verified, global common subexpression elimination and loop-invariant code motion" for a short discussion of CompCert's trusted computing base [32].

The latter approach relies on compile-time checks to ensure correctness. Its main advantage is the ability to use a more convenient programming language and programming model. Furthermore, the validator can be used for any implementation, allowing for relatively quick and adventurous iterations. Note that CompCert only guarantees semantic preservation; the effectiveness of optimizations is not covered.

Some examples of CompCert's existing optimizations include, among others, (1) function inlining (2) constant propagation (3) common subexpression elimination (CSE) (4) dead code elimination (DCE) and (5) a live-range splitting, graph-coloring (iterated register coalescing), register allocator. The register allocator [34] being an impressive example of the use of translation validation in CompCert. However, "[l]oop optimizations and instruction scheduling optimizations are not implemented yet." [22] These two notable exceptions are particularly important for in-order processors (see section 1.2) in order to expose instruction-level parallelism (ILP).

1.2 Instruction scheduling

Modern processors can be roughly divided into three groups: (1) "Statically scheduled superscalar processors" (2) "VLIW (very long instruction word) processors" (3) "Dynamically scheduled superscalar processors" [20, p. 218]

"All processors since about 1985 have used pipelining to overlap the execution of instructions and improve performance." [20, p. 168]. Moreover, nowadays internal components of the CPU are replicated "so that it can launch multiple instruction in every pipeline stage." [33, p. 343] For very long instruction word (VLIW) processors, "bundles" need to be explicitly formed to indicate which instructions should be issued simultaneously, whereas for superscalar processors this is done implicitly. Instruction scheduling is the art of ordering the instructions in such a way that the processor can safely (see section 1.2.1) execute as many instructions as possible in parallel via the aforementioned techniques.

Dynamically scheduled processors employ hardware techniques to reorder the program code at runtime to fit the hardware's capabilities. Particularly, in the desktop and server space, dynamically scheduled superscalar processors dominate. Even newer embedded system processors follow this design [20, p. 265]. However, supporting dynamic scheduling and other advanced features such as speculative execution requires more, complex, hardware. Conversely, designs with simpler control logic such as in-order VLIW processors use "less die space and energy for the same [theoretical] computing power" [37]. By combining pipelining and the ability to issue multiple instructions simultaneously, even simple processors possess tremendous potential for exploiting ILP. Due to their relative simplicity, VLIW and statically scheduled processors require extensive compiler support to realize their potential [20, p. 218]. For these processors it is the compiler's responsibility to discover and expose ILP. In this report we will focus on statically scheduled superscalar processors.

1.2.1 Correctness of instruction scheduling

This work assumes interlocked pipelines and is thus able to concentrate on the logical correctness of the instruction scheduling. That is, if instructions cannot be safely issued at the same time or

overlapped, the processor is assumed to stall the pipeline[3]. The stalling reduces or eliminates the overlap until it is safe to execute the following instruction. Thus, the expected sequential semantics are preserved. Of course, the timing and conflict information is important for good schedules since the instruction scheduler wants to maximize overlap and minimize stalls. Similarly, structural hazards and consequently the inability of the processor to overlap certain instructions is important. Both of these are taken into account by the actual instruction scheduling algorithms considered in this report. However, because we assume an interlocked pipeline they are less important for the current discussion, which focuses on correctness.

Consequently, we will consider [20, p. 170]

- true data dependences
- name dependences and
- control dependences

It is statically known which registers an instruction will read and/or write, and whether the memory will be affected. Due to a lack of integrated alias analysis the entire memory will be regarded as a single location in this report.

A true data dependence occurs when a later instruction uses a result of a previous instruction (read-after-write (RAW)). In contrast to name dependences there is a transfer of data between the two instructions. Name dependences may occur as antidependences, a location is written to after being read (write-after-read (WAR)), and output dependences where a location is written to twice (write-after-write (WAW)). Since renaming is able to resolve these dependences, they are also sometimes called "pseudo-constraints" [2, p. 469] of the scheduling problem. Instead of disallowing movement across branches (strict adherence to control dependences), it is expected that the exception behavior and data flow are preserved [20, p. 175]. The former is usually relaxed to forbidding schedules or other transformations to introduce any new exceptions.

In principle, any reordering, even across the whole program, that respects these constraints would be valid. Nonetheless, in order to make the problem tractable, instruction scheduling is usually limited to smaller regions of code. The smallest unit of such code is called a basic block.

1.2.2 Basic blocks

The most popular, and arguably intuitive, basis for instruction scheduling are basic blocks. A basic block is a sequence of instructions with a single entry and exit point. Thus, the basic block can be thought of as a single semantic unit: once execution starts at the entry point—barring runtime exceptions—all instructions of the basic block will be executed. Consequently, a basic block may be reordered arbitrarily as long as the dependences (cf. section 1.2.1) are not violated. As a result, any reordering that is valid according to the dependence rules, affects the modeled state of the system in the same way. Recall that CompCert assumes that the input program does not exhibit undefined behavior.

[3] Alternatively, the toolchain after CompCert, e.g. the assembler is assumed to verify and/or fix these issues.

1.2.3 Global code scheduling

It is estimated that "between three and six instructions execute between a pair of branches." [20, p. 169] Because of that, the amount of ILP available within each basic block is rather limited. However, as noted in section 1.2.1, it may be possible to exploit ILP across basic block boundaries. Indeed, many such approaches have been developed (e.g. [11, 21, 31, 19, 1]). The reason for this plethora of approaches to global code scheduling is the balance of flexibility for the instruction scheduler to obtain good schedules and the complexity in reasoning about it and actually realizing the desired schedule while preserving the program's semantics. Some schedules may require extensive modifications to the program beyond simple reordering. Two of the most well-established global code scheduling approaches are trace- and superblock scheduling.

Trace scheduling

In contrast to basic blocks (cf. section 1.2.2), traces [11] are considerably more flexible. They allow both multiple entrances as well as multiple exits and are ideally selected, guided by profiling information or presumably good heuristics, to be likely execution paths of the program at runtime.

Any instruction in the trace is only allowed one predecessor that is part of the same trace: the path covered by the trace is loop-free [11] and linear. Code motion above and below side entrances (joins) and exits (branches) is supported. Moreover, branches may be moved above joins and below splits as well [15]. As a result, so-called "bookkeeping" code needs to be inserted after trace scheduling in order to preserve the program's semantics. The rules for code motion past side exits are identical to those used in superblock scheduling and will be discussed in section 1.2.3. More generally for trace scheduling, "[t]he details of the bookkeeping phase are very complex and their formal presentation is unintuitive." [11, p. 7]. The assumption is that due to their, presumed or measured, importance to the runtime of the program, optimizing traces is worth the effort and possible pessimization (e.g. speculative execution of instructions) of other parts of the program.

Superblock scheduling

A superblock [21] is in a sense a compromise between the rigidness of basic blocks and the flexibility, but also complexity, of traces. Like basic blocks they can only have a single entrance, but akin to traces they may have multiple exits. Code may therefore still move above and below side exits during scheduling. The most complicated part of trace scheduling, reshaping the control flow graph (CFG) for branches moving above joins is thus avoided.

The types of movement allowed are as follows [21, 15, p. 5].

- Above side exits, possibly with renaming and/or substituting non-trapping versions for trapping instructions, e.g. dismissible loads. Instructions that are not live at the side exit and do not trap may be speculatively executed without any compensation.
- Below side exits with optional, interposed, duplication if the moved instruction is live with regard to the side exit.

An instruction is live with regard to an (side-) exit, if the location written to is or can be read by the remainder of the program. A location is either a register name or a memory location. Keeping

track of the liveness of registers is not too difficult, but differentiating memory location requires so-called alias analysis.

While superblock scheduling lacks the ability to move code across joins, tail duplication can be used to eliminate some of them. After selecting likely-to-be-taken-together traces (cf. section 1.2.3), the first side-entrance (join) may be identified and the rest of the trace (tail) be duplicated. Any other (outside) instructions branching to the original tail are then pointed to the duplicated code instead. The side-entrance is thus eliminated (cf. [36, p. 71]). Gregg reports that when combining tail duplication with superblock scheduling, neither superblock nor trace scheduling is clearly superior to the other. Even with regard to code size—where we would expect a much larger increase with superblock scheduling due to the anticipatory as opposed to on-demand duplication—they observe that trace scheduling's compensation code "often produces more code growth" [15, p. 11].

1.3 Certified, scalable, instruction scheduling

Six, Gourdin, Boulmé and Monniaux introduced scalable (cf. section 1.5) instruction scheduling at the level of basic blocks [37][4] and superblocks [38]. While the superblock scheduling happens before register allocation at the RTL level and in principle automatically applies to all architectures (target platform specific timing information is helpful, however), basic block scheduling must be ported for each targeted architecture because it operates directly on the platform-specific assembly code. Currently, it is available for KVX and AArch64. For both types of scheduling, they employ the well-known technique, for formal verification in general but also CompCert in particular (e.g. [34]), of translation validation (cf. section 1.1, see sections 1.3.2 and 4.3 for more details). Because the oracle—in this case the instruction scheduler—is not trusted, it has total freedom with regard to its implementation. Of course, the accepted transformations are limited by what is provably equivalent according to the verifier, which is proven sound but not complete.

The verifiers for verifying basic and superblock schedules are based on (1) symbolic execution for correctness and (2) hash-consing in order to scale the compile-times to large pieces of code (in particular large basic- and superblocks). We will expand on both of these in sections 1.3.2 and 1.3.3.

Loop transforming optimizations themselves are out of scope of this report but please note that "[t]he simplest and most common way to increase ILP is to exploit parallelism among iterations of a loop." [20, p. 170] Fortunately, loop unrolling, loop peeling and loop rotation have been added to this version of CompCert as well [36]. All of these optimizations, when applied to inner loops[5] increase the size of superblocks. Particularly in combination with tail-duplication [36], even inner loops with conditional branches contained in them can be covered by a single superblock.

1.3.1 RTLpath

[4] Bundles are also supported. A bundle is a set of instructions that can be issued at the same time by the VLIW processor used in the paper.

[5] Since they are usually predicted to loop.

```
Definition istep (ge: RTL.genv) (i: instruction) (sp: val) (rs:
↪  regset) (m: mem): option istate :=
  match i with
  | Inop pc' => Some (mk_istate true pc' rs m)
  | Iop op args res pc' =>
      SOME v <- eval_operation ge sp op rs##args m IN
      Some (mk_istate true pc' (rs#res <- v) m)
  | Icond cond args ifso ifnot _ =>
      SOME b <- eval_condition cond rs##args m IN
      Some (mk_istate (negb b) (if b then ifso else ifnot) rs m)
  (* ... *)
  end.
```

Listing 1: Excerpt of the definition of `istep`, defining how execution proceeds within a path.

The prepass scheduling is implemented as a pass from RTLpath → RTLpath, where RTLpath is an extension of mainline CompCert's RTL intermediate representation (IR). A path more or less corresponds to the notion of trace of section 1.2.3, but since traces have a special meaning in the context of CompCert, the term "path" will be used instead. While RTLpath itself does not require these paths to be non-overlapping, the scheduling does. We will therefore only consider this case, at which point the paths correspond to superblocks. In the following, path and superblock will be used interchangeably. As described in section 1.2.3, paths ("traces") are created by a selection heuristic or are based on profiling information. Moreover, tail duplication may be used to turn paths into essentially superblocks of larger size.

RTLpath extends RTL with

- Path metadata, notably the size of each path and verified liveness information.
- A `path_map`, partially mapping CFG nodes ("PCs") to the above metadata, in which case the node is the beginning of a path. This map is checked for well-formedness: each target of an exit must itself be the beginning of a path. Of course, the entrypoint of a function must itself be a path entry. Additionally, except for the final instruction of a path, instructions may only be basic, or two-way conditional branches (`Icond`) with a predicted successor. The latter create side-exits.

Essentially, RTLpath is a version of RTL with verified annotations that enable execution to proceed multiple instructions at a time. In contrast to RTL whose semantics execute instruction-by-instruction, RTLpath executes the code in `path_steps`: the entirety of a path is executed at once. A `path_step` either hits an early exit, in which case the condition of a side exit must have evaluated to true[6] or exit the path normally at the end. Note that this gives n possible executions, where n is the number of (side-) exits. In order to define the execution of a path, an internal state is defined (`istate`), which keeps track of the register and memory state (`irs` and `imem`, respectively), the next instruction to execute (`ipc`), and whether execution continues within the path or a side-exit is triggered (`icontinue`). The execution within a basic block is defined by

[6] During the generation of RTLpath, predicted conditional branches are normalized to always favor the `false` case.

a succession of `isteps`, bounded by the length of the path, given in the `path_map`. Listing 1 shows the definition of `istep` for three cases: 1) `Inop`, in which case the register and memory state are left untouched and execution proceeds to the successor named in the instruction. 2) `Iop` which evaluates the operation, modifies the register state[7] and continues to the unique successor. 3) `Icond`, which evaluates the condition and either triggers the path's side exit by setting the `icontinue` boolean to false or continues execution at the predicted successor.

RTL's and RTLpath's execution semantics are shown to be in bisimulation, which makes it easy to move back and forth between the two.

The liveness information of RTLpath can be used to derive a simulation between RTLpath states which only considers equality between live registers. This allows the speculative execution of non-live instructions, essential for superblock scheduling. In order to prove lock-step simulation between superblocks, as required by instruction scheduling, Six, Gourdin, Boulmé, and Monniaux define a symbolic execution semantics [38, 36].

1.3.2 Simulation test via symbolic execution

This section will only give a brief overview of the symbolic execution strategy used to establish simulation between RTLpath superblocks [38, 4]. Section 4.3 will give more details on selected aspects that were modified as part of the implementation of chapters 2 and 3

Transformations of the type RTLpath → RTLpath, in particular superblock scheduling but also some rewriting optimizations (see "Verified Superblock Scheduling with Related Optimizations" [38]), are proven via symbolic execution to establish simulation. That is, ultimately, symbolic execution will be used on the original, provably correct, superblock and a—supposedly matching—untrusted superblock returned by the oracle. Both superblocks will be executed from the same initial symbolic state. The resulting symbolic states should be verifiable by an efficient test, "such that its success would imply the simulation" [36, p. 141].

To ease the proof, the symbolic execution is divided into two parts.

Abstract symbolic execution First, an abstract version that simplifies the proof of bisimulation necessary for correctness with regard to RTLpath [36, p. 116]. Symbolic values are introduced, as well as modified semantics manipulating them. A symbolic state, `sstate`, of a path is made up of an symbolic internal state, `sistate`, and a symbolic final value `sfval`, which records information about a possible final instruction[8]. The `sistate` in turn is made up of a list of side exits, including the conditions under which they are taken, and an `sistate_local` keeping track of the symbolic register and memory state, as well as the preconditions necessary for a non-failing execution. The preconditions are given as a conjunction of assertions that the execution of the instruction does not fail. It also stores the current PC. See figure 7.2 of "Optimized and formally-verified compilation for a VLIW processor" for a visual summary and more detailed explanations [36, p. 120]. Abstract symbolic execution (`sexec`) proceeds by symbolically executing each instruction of the path while updating the `sistate` upto the last instruction. Then symbolically executing a possible final instruction and storing the two results in an `sstate`. Two lemmas then

[7] Note that only load and store instruction may modify the memory state within a path.
[8] In case a path ends without a necessarily path-terminating instructions this value can be `Snone`.

prove the bisimulation between RTLpath and the abstract symbolic execution. `sexec` is an over-approximation of `path_step` but also always represents a concrete execution of `path_step`, thus proving the bisimulation [38, p. 18]. An additional predicate refines this bisimulation to be less strict e.g. to accept register equality modulo liveness, required by useful superblock scheduling.

Hash-consed symbolic execution Second, a computationally efficient version of the symbolic execution based on hash-consing, thus enabling the scalable verification of even large superblocks. Data structures are defined to refine their abstract twin. Abstract symbolic values are refined by hash-consed counterparts, which simply add an additional field to store the hash-id, `hid` (see section 1.3.3 for more details on hash-consing). Their semantics are inherited from their abstract version. Dropping/ignoring the `hid` by projection, an equivalent, abstract, version is obtained whose semantics can be reused. A similar process is followed for the symbolic state definitions, which are refined by more efficient data structures. For example, the precondition in `sistate_local`, recording which instructions may not fail (and are guaranteed to not fail for the original superblock) at a given point in the execution of the superblock is originally defined as a conjunction of such assertions. This is hardly executable. Instead, `sistate_local`'s *hash-consed* refinement, `hsistate_local`, stores a list of `hsvals`, which must not fail[9]. Thus, "the proof of [Six, Gourdin, Boulmé, and Monniaux's] implementation simply reduces to ensuring that each elementary computation of the symbolic execution preserve[s] 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 impl[y] the semantical equalities involved in its detailed model." [38, p. 20]

This symbolic evaluation described in [38] and [36] can thus verify the following types of movement:

- Moving an instruction *above* a side exit, if it does not impact the data flow. A register write may therefore only be moved above a side-exit if the register written-to is not live at the predicted-not-taken branch. Furthermore, the move may not introduce a new exception, unless the instruction is substituted with an equivalent, non-trapping, one.
- Moving an instruction downwards, that is below a side exit, is possible if its effect on the data flow is not visible at the side exit (it is not live with regard to the side exit). Note that moving possibly trapping instruction downwards is possible because they do not introduce new failures to the superblock. Interestingly, it *does* add a new exception to the constituting basic block that is the target of the move. Since superblocks do not have side-entrances, however, it is guaranteed that if the moved (added to the basic block) instruction fails in the basic block, the basic block would (semantically) never have been reached anyways.

Note that the scheduling itself, as presented in [36, 38], does not introduce any code duplication.

[9] While the abstract symbolic execution asserts that none of the instructions may fail, the concrete symbolic execution only stores those instructions (symbolic expression/values) that *may* fail.

1.3.3 Hash-consing

In principle, the comparison of symbolic states requires expensive tests for structural equality since each symbolic value essentially records the entire "history" of computations relevant to it, starting from the initial symbolic state. The structure of these symbolic values resembles an abstract syntax tree (AST). A naïve implementation would pose problems with regard to both memory usage and the cost of structural checking. *Hash-consing*, a staple of imperative programming, alleviates both problems by using memoizing smart constructors for the symbolic values. Instead of allocating two or more structurally identical terms, subsequent invocations of a hash-consing constructor return the memoized object first constructed. Naturally, thus constructed values must be immutable. As a result, data structures with many repeating structures can be represented very compactly. Additionally, expensive structural equality tests can be replaced by constant time pointer equality tests [4, p. 58].

The name *hash*-consing is derived from a typical implementation detail: in order to quickly find and return already created objects, hash-tables or maps are generally used in the implementation. This very reason for its efficiency makes it difficult to employ this technique in the context of CompCert. Gallina, Coq's specification language, lacks many important imperative features to easily implement hash-consing. Nonetheless, multiple approaches are possible [6]. ASTs are a prime-use case for hash-consing [6].

Six, Gourdin, Boulmé, and Monniaux do not rely on the full power of hash-consing. Their system only relies on the property that physical equality i.e. pointer equality implies structural equality, and not that structural equality implies pointer equality (although this property is still highly desirable). This relaxation of the problem allows for a lightweight approach implemented in the Impure library [5]. Impure implements an efficient hash-consing factory. The only extra assumption[10] being made is that embedding OCaml's pointer equality is correct as axiomatized in listing 2. The axioms states that *if* `phys_eq` terminates, returning a boolean that is `true`, then its inputs can be considered equal for Coq's definition of equality. Note that as a user of Impure we cannot make any assumption on the return value of `phys_eq`, except that if it returns, it will return a boolean. Given a definition of how to compare hash-consed values[11], and some auxiliary functions, the hash-consing factory returns a memoizing function [4, p. 60]. The memoizing function doing the heavy lifting is itself untrusted (`xhCons`). It is essential, that this untrusted-oracle-provided function behaves observationally like an identity (cf. the discussion on proving that a certain relation between inputs and outputs is preserved. [4, pp. 37–38]). This is verifiably guaranteed by the `hCons` wrapper, which adds a runtime check that the returned value is equal to the given input with regard to the user-provided hashing equality.

1.4 Missed scheduling opportunities

If we compare the abilities of superblock scheduling in general (section 1.2.3) and in the context of CompCert in particular (section 1.3.2) we notice two missing scheduling opportunities.

[10] In addition to adding Impure (see [4, pp. 27–29, 44–47]).
[11] An example for hash-consed condition evaluations is given in listing 14.

```
1 Axiom phys_eq: forall {A}, A -> A -> ?? bool.
2
3 Axiom phys_eq_correct: forall A (x y:A), WHEN phys_eq x y ~> b THEN
  ↪  b=true -> x=y.
4
5 Extract Inlined Constant phys_eq => "(==)".
```

Listing 2: IMPURE's axiom on pointer equality and extraction to OCAML's pointer equality test (cf. [5, p. 10, 4, p. 32]).

1. Due to the lack of register renaming, certain instruction may not be moved due to name dependences.
2. The symbolic execution is not able to verify the movement—and therefore necessary duplication—of instructions below side exits, if they are live with regard to that side exit.

The first shortcoming is mitigated by the fact that the superblock scheduling is applied at the RTL level, that is before register allocation (*prepass*). As a result, name dependences are much less common than after register allocation when the compiler handles the lack of infinitely many (pseudo-)registers by spilling and restoring registers to and from the stack. However, they still do occur, most notably after certain loop optimizations as introduced in [36]. For example, loop unrolling is implemented by an ingeniously simple oracle. The verifier relies on the strict duplication of its instructions to achieve this remarkable brevity and flexibility[12]. Consider listing 5 which is the result of unrolling and then rotating the loop of the simple vector sum function of listing 3. Although the superblock is much bigger than initially (listing 4), the scheduling opportunities have increased very little:

1. The duplicated loop index increment cannot be moved above the side exit because it overwrites the register used for the exit condition.
2. Unless the hardware platform and operating system support dismissible loads[13] the load operations cannot be moved across the side exit either.
3. The store operation cannot be moved below the side exit since the memory is implicitly always considered live. Transitively, the floating point addition and two associated loads cannot be scheduled below the side exit either.

Figure 1.1 shows the dependency graph of listing 5. Unsurprisingly, we observe that prepass scheduling has no effect on this loop.

Generalizing the first point we notice that unrolled loops[14] only expose a fraction of the possible ILP due to name dependences, which are a direct result of the simple duplication of instructions without renaming. Therefore, one big advantage of superblock scheduling over basic block

[12] The same verifier can be used to verify loop rotation, loop peeling and will later be used to enable code motion past side exits (cf. section 4.2).
[13] Loads that do not trigger an exception when performing an invalid memory access.
[14] At least inner loops where iterations are independent from each other.

```c
void vector_sum(double *dest, double *v1, double *v2, size_t len) {
  for (size_t i = 0; i < len; i++) {
    dest[i] = v1[i] + v2[i];
  }
}
```

Listing 3: Simple vector sum function in C.

```
  i = 0L

loop:
  if (i >= len) goto exit else (prediction: fallthrough)
  t1 = v1[i]
  t2 = v2[i]
  t = t1 + t2
  dest[i] = t
  i = i + 1
  goto loop

exit:
  return
```

Listing 4: Simplified RTL code for vector sum (listing 3). The register names and numeric labels have been renamed and the instruction representation slightly prettified.

scheduling, speculative execution of instructions at compile-time, is severely hindered. Chapter 2 tackles this problem.

The second point cannot be fixed without support by both the hardware and operating system running the process and will not be discussed further for the purpose of this report but ties in with the next point.

The inability of CompCert's current superblock scheduling to move live instructions below side exits can miss advantageous interleavings of instructions (cf. fig. 4.2a and table 4.1).

Chapter 2 will discuss the general idea for a register renaming approach compatible with the symbolic execution semantics of section 1.3.2. Then, chapter 3, elaborates the ideas and design behind the code motion below side exits implemented. Their implementations—including changes to RTLpath and its symbolic execution, necessary for the latter and useful for the first—are discussed in chapter 4. The results are experimentally evaluated (section 5.1) and discussed (chapter 6) before concluding the report. Next, however, related work is discussed.

1.5 Related work

As noted in section 1.2.3, the size of basic blocks is often quite limited so that, for statically scheduled platforms, global code scheduling becomes essential. Many variants have been proposed and implemented. Trace scheduling [11] is theoretically even more powerful than super-

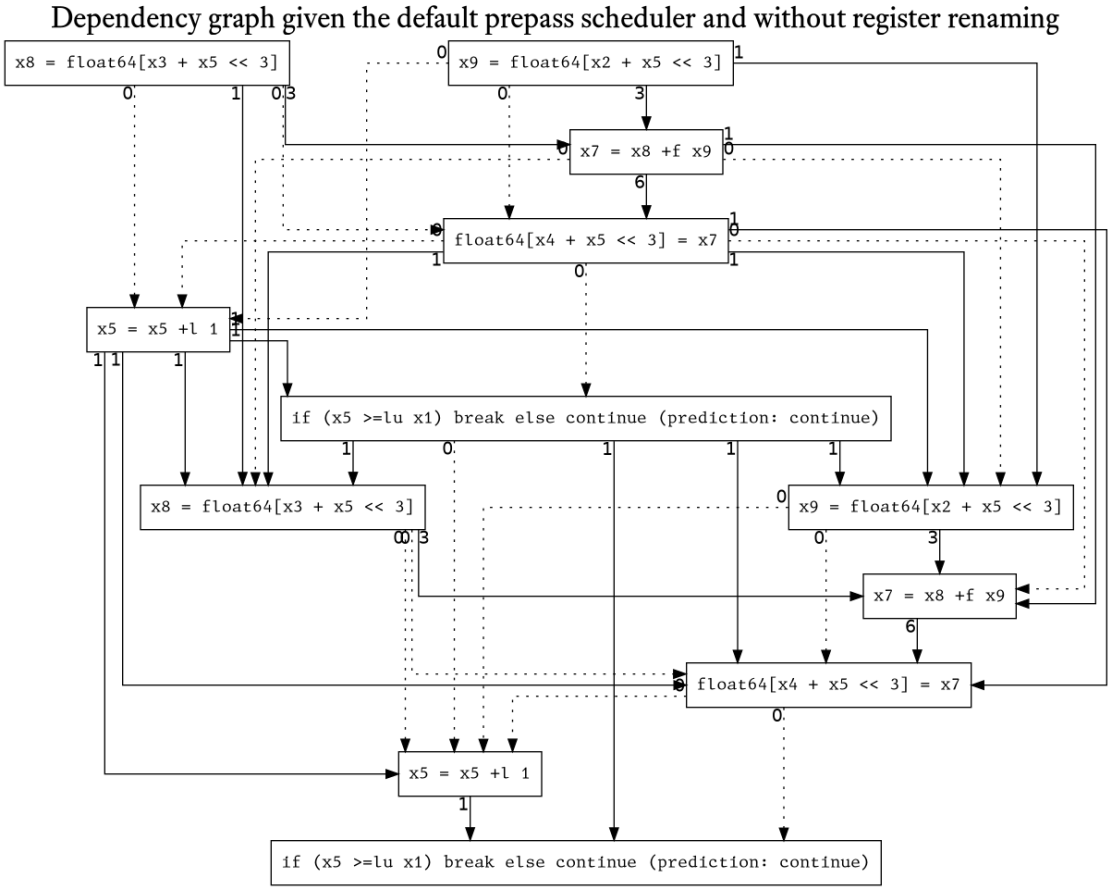


Figure 1.1: Dependency graph of unrolled and rotated vector sum. Due to name dependences almost no movement is possible. Pseudo-register `x5` corresponds to variable `i` in listing 5, `x8`, `x9` and `x7` to `t1`, `t2` and `t` respectively. Edges are annotated with their expected latency at the RTL level.

```
  i = 0L
  if (i >= len) goto exit else fallthrough (prediction: fallthrough)
loop:
  t1 = v1[i]
  t2 = v2[i]
  t = t1 + t2
  dest[i] = t
  i = i + 1
  if (i >= len) goto exit else fallthrough (prediction: fallthrough)

  t1 = v1[i]
  t2 = v2[i]
  t = t1 + t2
  dest[i] = t
  i = i + 1
  if (i >= len) goto exit else goto loop (prediction: fallthrough)

exit:
  return
```

Listing 5: Prettified RTL representation of loop-rotated and unrolled vector sum code.

block scheduling (cf. section 1.2.3). Nonetheless, many implementers have reached towards superblocks due to their relative simplicity. The paper introducing superblocks [21], further mentions five dependence-removing optimizations to aid scheduling. While both trace scheduling and superblock scheduling target linear sequences of code, Havanki, Banerjia, and Conte consider "Treegions". As the name implies, each treegion has a single entrance but can encompass a tree-like structure of the program's CFG [19]. In their experiments they report a 15% – 20% increase in performance over superblock scheduling. Software pipelining [2, 25, pp. 472–482] is a more advanced approach to instruction scheduling, specifically targeting loops. In essence the goal is to compute multiple iterations of the loop at once without necessarily unrolling the loop as many times. A prologue and epilogue need to be inserted in order to preserve the original loop's semantics.

Many approaches to verified compilation exist with differing goals. CakeML is another general-purpose verified compiler, compiling StandardML rather than C. Instead of using Coq it is written and proven with the Isabelle/HOL4 theorem prover. seL4 [24] on the other hand is a verified-correct microkernel that is proven with regard to semantic correctness and security. They go from an abstract presentation down to C using Isabelle/HOL, and, on some platforms prove the binary code produced. The latter means that they do not need to trust the compiler and linker. Its proof is automatically performed via translation validation by an SMT solver, able to handle all optimizations of gcc-O1 and much of seL4 at -O2. Note however, that the C source code being compiled comprises only 9500 lines of C code [35]. In a similar vain, ACL2 has been used to verify user-mode programs with regard to a formal x86 ISA model [14]. Some programs are verifiable at gcc's -O2 level while others are validated at the default optimization level.

Tristan and Leroy implemented a slight variation of trace scheduling for CompCert [40].

They restrict full trace scheduling by not allowing the reordering of branches. In contrast to Six, Gourdin, Boulmé, and Monniaux's work on superblock scheduling, their optimization applies to the Mach IR of CompCert, after register allocation. The same goes for their instruction scheduler of basic blocks, presented in the same paper [40]. Unfortunately, their implementation of symbolic execution was not hash-consed and exhibited exponential blowup. Moreover, they do not consider liveness and always duplicate instructions when moving code below side exits or above joins. Tristan, Govereau, and Morrisett demonstrate that symbolic execution is able to verify many kinds of transformations on LLVM using symbolic execution and a value graph [39].

Static single assignment form (SSA) presents a general solution to prepass register reuse by outlawing it. If multiple definition can be the most current version of a variable, ϕ nodes are inserted, representing the correct choice, depending on the predecessor. Given SSA it is possible to perform optimal register allocation in polynomial time [18]. On the hardware side, dynamic register renaming with Tomasulo's algorithm presents the state of the art. It is able to rename registers across branches. Additionally, the hardware may implement more "registers" (or reservation stations in Tomasulo's algorithm) than officially available in the ISA. With regard to the issue of register allocation and its interdependence with instruction scheduling, combinatorial approaches aim to solve both register allocation and instruction scheduling optimally. Recent approaches have been shown to scale up to 1000 instructions [29, 30].

2 Register renaming modulo liveness

As explained in section 1.3.2, the superblock verifier considers superblocks equivalent if their states match modulo register liveness at each exit. It follows that we can freely rename any, essentially temporary, registers if they are not live outside of the superblock. However, any renaming must be restored before any (side-) exit that may read the old register, expecting the renamed register's value. Due to the verified liveness information available at the RTLpath level, it is possible to only restore those registers which are live with regard to an exit.

For example, the registers holding the loaded values in listing 4 are not live and need not be restored. Moreover, the loop index variable `i`'s scope does not extend beyond the (trivially inner) loop. As a result, `i` is not live with regard to the side exit, nor its duplication, generated by rotating and unrolling the loop (cf. listing 5). However, `i` is live during the next iteration, so before leaving the superblock at its predicted exit, `i` must receive the correct value. Restoring registers to their, globally[1], expected value after renaming will be called *restoration code*. Because `i` is not live for the side-exits, the restoration code may be placed after the second conditional branch by changing the successor information accordingly. The effect of renaming registers on the dependency graph for this example is shown in fig. 2.1 (cf. fig. 1.1). Already, this small change decreases the expected time (as determined by the "list" instruction scheduler at the RTLpath level) when the final instruction of this particular superblock is started from 24 to 22 by removing many of the name dependences. Note, that the restoration code itself introduces a name (output/WAR) dependency again. Section 4.2 explains how to move or duplicate code from the superblock below a side-exit. This can also be used to offload the restoration code for all side-exits. Only the final restoration code, before the expected exit of the superblock must stay part of the superblock. Because it will stay part of the superblock, it will be inserted before applying the heuristic of chapter 3. This is a pessimistic choice because some of the restoration actions will be obviated by the register allocation. More generally, the impact of register allocation on the effect of this transformation can be quite significant due to the problem of register pressure (see section 2.2).

2.1 Register renaming and code movement below side exits

The register renaming approach of this chapter and the method for code motion past side exits of chapter 3 and section 4.2 are implemented to work independently from each other. For example, section 4.1.2 explains why we want to insert some restoration code before code motion below side exits, which, however violates the nice property that each definition in a superblock has only a single definition. Additionally, this property is trivially violated when simply not enabling register renaming while turning on code motion below side exits (cf. tables 5.2 and 5.3). In addition to

[1] Outside the current execution of the superblock.

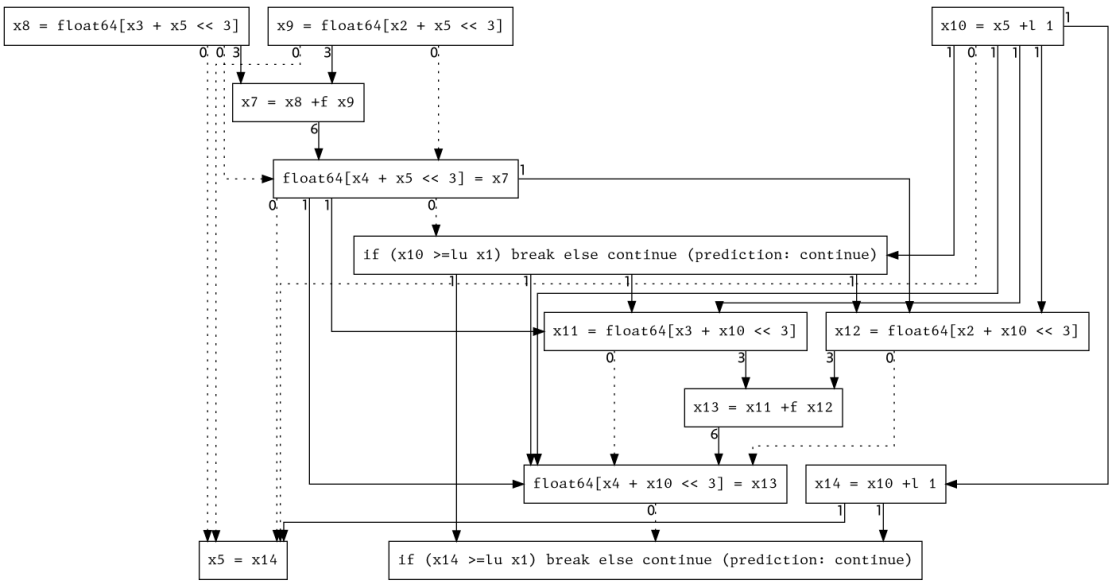


Figure 2.1: Dependency graph of unrolled and rotated vector sum. After renaming, a lot of dependencies are eliminated (cf. fig. 1.1). Note that the memory stores, and preceding calculations must occur before their respective side exit. Because on AArch64, loads cannot be turned into speculative loads, the second loop iteration can also not be interleaved with the first. The calculation of the loop indices (now `x10` and `x14`), on the other hand, have become flexible and could be pre-computed. Similarly for the restoration code (`x5 = x14`), which can be placed anywhere after its last use.

the systematic register renaming described here, a second, ad-hoc, mode is implemented. For example, whenever an instruction is duplicated twice or more in order to move below multiple side exits (cf. section 4.2) the arguments of the duplicated instruction must not be mutated between duplicate occurrences. Furthermore, the eventual insertion of all restoration code may invalidate naïvely duplicated code which is necessary for code movement below side exits. In this case, extra aliasing logic is applied: before restoring the register, a copy is made and used appropriately in the remainder of the superblock.

2.2 Register Pressure

During register allocation, the compiler has to cope with the finite number of registers available to it. It maps the pseudo registers, of which they were infinitely available and many used, to a small set of registers more closely resembling those of the hardware. If more registers are live at a given point than available, they need to be spilled, and restored on the next use. Excessive spilling and restoring is costly in two regards. For one, it introduces load and store instructions which are relatively costly. Two, this extra code increases the code size, sometimes significantly (see fig. 5.3). In turn, this code size increase may not only be problematic for limited device storage e.g. in small embedded systems but also for performance. The increased size may cause the code, or "hot" parts of it, to no longer fit into the instruction cache of the CPU. Together, "[t]he transformed code, while theoretically faster, may lose some or all of its advantage because it leads to a shortage of registers. [...] The problem becomes especially challenging in multiple-issue processors that require the exposure of more independent instruction sequences whose execution can be overlapped." [20, p. 182]

The problem of the register renaming approach presented earlier is twofold. The first is rather fundamental to the presented approach: register renaming introduces a lot of new registers. In principle, just the renaming does not have an influence on the number of *live* registers. Nonetheless, it can have an influence on the heuristics employed by the register allocator. Second and more importantly, applying prepass scheduling to increase ILP, the number of live registers and thus register pressure can increase dramatically. After all, exposing more ILP by overlapping instructions and thus generating more live registers was the entire point of combining a renaming pass with prepass scheduling. This latter issue can be mitigated by a register pressure aware instruction scheduling algorithm (see fig. 5.3).

The restoration code inserted by the renaming also increases the code size at the RTLpath/RTL level. Again, the problem is twofold. First, executing the restoration code costs cycles which is particularly irksome with regard to the superblocks we are trying to optimize. Second, it increases the code size. By moving the restoration code for side-exits out of the superblock (cf. chapter 3), the former issue can be avoided at the cost of exacerbating the latter. If the same renaming is live for multiple side exits, moving the restoration code out of the superblock, that is *after* the side exits, necessitates multiple copies of the restoration code. Otherwise the restoration code *before* an earlier side exit still applies to a later side exit of the same superblock. The upside is that the superblock executes less instructions. Because the side exit has been predicted to not be taken, this is assumed to be a worthwhile trade-off. In particular, Six, has adapted CompCert's linearization oracle, which is responsible for the generated code layout, to always lay out the basic

blocks of a superblock contiguously [36, p. 176]. In this light, avoiding as much restoration code in the superblock as possible, while likely increasing the global code size, also reduces the code size of the superblock itself, as laid out in memory. The process of code linearization has been slightly modified in this work, because it was observed that the restoration code created outside of the superblock would often be laid out directly after the superblock. In the case of inner loops spanned by a superblock, this turns the loop condition (which sits at the end for a rotated loop), looping back to the beginning of loop, into two jumps instead of one because the fallthrough successor is no longer the side exit but some restoration code. To counteract this for inner loops spanned by a superblock, we prefer the predicted-not-taken branch to be laid out after the superblock. Note that restoration code may or may not be eliminated by register allocation. Register allocation removes restoration code that has become redundant due to both higher-level registers having been mapped to the same lower-level register.

3 Live code motion below side exits

As mentioned in section 1.4, in addition to register renaming, we are also interested in moving live instructions below side exits. Once again, consider the dependency graph in fig. 3.1 for the unrolled and rotated vector sum example. Interestingly, the system used to generate these dependency graphs, GraphViz [13] and in particular `dot` [12], visualizes the advantage of removing dependencies in more than one way. In addition to removing edges, which makes the graphs visually cleaner, they become successively more compact[1] (height-wise, cf. figs. 1.1, 2.1 and 3.1). For the vector sum example only the memory stores are live and could be moved below side exits. While the loop index is live, it is a true dependence of the conditions guarding the side exits. Unfortunately, this added capability does not improve the expected performance[2] for this particular example.

Consider instead listing 6 and its generated assembly code in table 3.1. Prepass register renaming[3] of the floating point operations' target registers does not enable the desired interlaving of expensive computations because (a) the result (`s` in listing 6 and `d0` in the assembly code) is live after the loop, such that the first loop iteration cannot be merged into the second. (b) Merging the second into the first (speculative execution) is not possible because the computations depend on the load, which cannot be moved above the side exit as that would introduce an additional possible failure. Without dismissible loads the latter cannot be realized. The former, however, can at the cost of code duplication (see the suggested assembly code of table 3.1). We should thus be able to reduce the loop's final instruction's starting time from 19 to just 15.

While achieving the desired expected performance, the actual code generated (see table 3.2) is larger than desired due to a limitation in the heuristics employed (see section 3.1). For a slight

[1] Instruction scheduling in the context of trace scheduling is also sometimes called *trace compaction* [20, H-19].

[2] For these examples, expected performance will be roughly equated to the starting time of the final instruction in a given sequence (corresponding to a superblock). This does not correspond to actual performance, especially in the presence of loops and all the complications of actual hardware. Section 5.1 tries to evaluate performance in a more realistic manner.

[3] Note that at the RTL/RTLpath level, the same psuedo-registers are used for the two loop iterations. After register allocation, as displayed in table 3.1, this need no longer be the case.

```
1 double sumsq(double *x, size_t len) {
2    double s = 0.0;
3    for (size_t i = 0; i < len; i++)
4      s += x[i] * x[i];
5    return s;
6 }
```

Listing 6: C code to compute the sum of squares. Example due to Sylvain Boulmé.

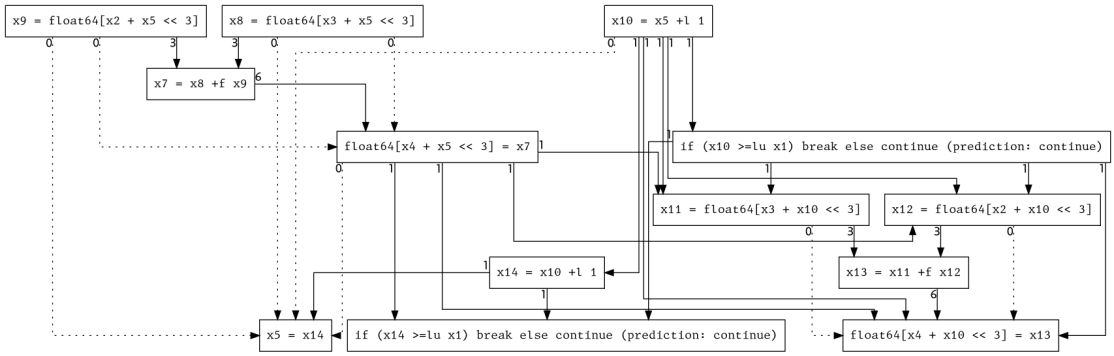


Figure 3.1: Dependency graph of unrolled and rotated vector sum. With the ability to insert compensation code, even less dependencies are mandatory, giving the prepass scheduler greater flexibility.

variation of this example, using two instead of just one buffer (`s += xi] * y[i]`), we are able to reduce the final starting time from 21 to 16.

The capabilities of the code motion below side exits have to be distinguished for

- Memory stores and
- Other instructions

Due to the way code motion below side exits is implemented (see section 4.2 and fig. 4.1), moving an instruction below multiple side exits requires several redundant copies in the superblock. This is verifiable for all instructions[4] but memory stores. In addition to not supporting alias analysis, executing a store instruction once versus twice always yields two different symbolic memories. Normally, duplicating a store instruction would not be interesting, were it not for the limitations discussed in sections 4.2 and 4.3.

3.1 Deciding which code to move below side exits

Ideally, the prepass instruction scheduling and duplication step necessary to realize schedules requiring moves below side exits would be integrated with each other. This is not the case in this work. Assuming all instructions could be speculatively executed, one approach would be to simply move as much code as possible below side exits, and let the subsequent instruction scheduling pass decide which instructions actually belong towards the top of the superblock. Unfortunately, some instruction can often not be speculatively executed e.g. in the absence of dismissible loads. Moreover, due to the aforementioned lack of memory aliasing analysis store instructions are always considered live and can therefore not be moved above a side exit. Consequently, too aggressive code motion downwards may hinder the instruction scheduler in finding good schedules. Instead, the actual instruction scheduling pass is run on a relaxed scheduling problem, removing the

[4] Since we are discussing transformation for instruction scheduling, final instructions (see section 1.3.2), which cannot be scheduled, are ignored.

Instruction	Start	Latency
`.L101: ; loop start`		
`ldr d2, [x0, w2, sxtw #3]`	0	3
`add w2, w2, #1`	0	1
`cmp w2, w1`	1	1
`fmul d1, d2, d2`	3	6
`fadd d0, d0, d1`	9	6
`b.ge .L100`	9	1
`ldr d4, [x0, w2, sxtw #3]`	10	3
`add w2, w2, #1`	10	1
`cmp w2, w1`	11	1
`fmul d3, d4, d4`	13	6
`fadd d0, d0, d3`	19	6
`b.lt .L101`	19	1
`.L100: ; loop exit`		

Instruction	Start	Latency
`.L101: ; loop start`		
`ldr d2, [x0, w2, sxtw #3]`	0	3
`add w2, w2, #1`	0	1
`cmp w2, w1`	1	1
`b.ge .L102`	1	1
`ldr d4, [x0, w2, sxtw #3]`	2	3
`add w2, w2, #1`	2	1
`fmul d1, d2, d2`	3	6
`cmp w2, w1`	3	1
`fmul d3, d4, d4`	5	6
`fadd d0, d0, d1`	9	6
`fadd d0, d0, d3`	15	6
`b.lt .L101`	15	1
`.L102: ; compensation code`		
`fmul d1, d2, d2`		
`fadd d0, d0, d1`		
`.L100: ; loop exit`		

Table 3.1: Compilation result (left) for listing 6, without register renaming or live code motion below side exits. Prepass register renaming, as discussed in chapter 2, which would enable the speculative increment of the loop counter does not improve the expected performance. Because AArch64 does not allow dismissible loads, the expensive computations `fmul` and `fadd` cannot be interleaved with the first iteration. As a workaround we can imagine the code on the right. The branch has been moved above the first iteration's expensive computations, thus interleaving the two iterations without requiring dismissible instructions. Note the insertion of compensation code to preserve the data flow.

Instruction	Start	Latency
`.L101: ; loop start`		
`ldr d2, [x0, w2, sxtw #3]`	0	3
`add w3, w2, #1`	0	1
`add w2, w3, #1`	1	1
`cmp w3, w1`	1	1
`b.ge .L102`	2	1
`fmul d5, d2, d2`	3	6
`ldr d16, [x0, w3, sxtw #3]`	3	3
`cmp w2, w1`	4	1
`b.ge .L103`	4	1
`fmul d6, d16, d16`	6	6
`fadd d7, d0, d5`	9	6
`fadd d0, d7, d6`	15	6
`b .L101`	15	1
`.L102: ; compensation code`		
`fmul d3, d2, d2`		
`fadd d0, d0, d3`		
`b .L100`		
`.L103: ; compensation code`		
`fmul d1, d16, d16`		
`fadd d4, d0, d5`		
`fadd d0, d4, d1`		
`.L100: ; loop exit`		

Table 3.2: Assembly code generated for listing 6 with both register renaming and live code motion below side exits enabled. Compared to the envisioned result (see table 3.1), the same final starting time is achieved. However, the code duplication incurred is much greater due to the relative obliviousness of the heuristics (see section 3.2).

scheduling constraints that we may remove via code motion below side exits. The resulting schedule is checked against the full constraints to see which instructions "want" to move downwards but cannot. Due to the lack of compensation-code aware instruction scheduler, the result is compared to a run on the original problem. If the original problem yields an equally good schedule[5], no code is duplicated. This simple work-around still allows many schedules involing significant amount of code duplication for marginal gains (see fig. 5.1). A slightly more fine-grained mitigation is discussed in the next section. Note however, that these final-time-comparing approaches still do not catch cases where an equivalent schedule, duplicating less instructions, exists (cf. the desired assembly code of table 3.1 and the generated one, with an equivalent final time but much more code duplication in table 3.2).

3.2 Code size

Code motion below side exits does not increase the size of the superblock itself at the RTLpath level[6]. However, the compensation code that must be interposed between the side exit and its target does increase the code size. Consider fig. 4.2. Many instructions, which previously could not have been moved, are moved below the superblock's first side exit to its predicted successor. The appropriate compensation code to maintain correctness has been inserted into the predicted-not-taken successor. Since some code is moved below the second side exit as well, which is thus no longer the last instruction of the superblock, *one* additional unconditional branching instruction is inserted inside the superblock. Globally, however, we can see that the predicted-not-taken successor's size increases by eight instructions[7]. Additionally, the movement of code below the second side exit further increases the code size by two instructions as a result of compensation code. Here the interposition of the code between the side exit and its original target is clearly visible. Moreover, it highlights again the obliviousness of the heuristic with regard to the duplication cost. The problem is particularly silly due to the necessary insertion of the unconditional branch at a lower IR. This branch is not accounted for at the RTL/RTLpath level and not predicted by the heuristic. Ideally, such trivially factorizable code sequences would be avoided.

Like the traces of trace scheduling, the superblocks of this work have been created by static or profiling based branch prediction, followed by trace selection and transformations meant to increase the size of superblocks (or even turning some traces (paths in the context of CompCert) into superblocks in the first place). Note that if a superblock does not contain side exits, i.e. does not make a prediction/assumption on the likely code execution path, neither speculation nor compensation code after code motion below side exits apply. Thus, the key assumption of trace scheduling—that the importance of the execution time of the superblock outweighs the possible cost for other code executions—stands to reason. However, until now, this only applied to the cost of speculative execution of instructions, including register pressure, but not directly duplicating instructions. The introduction of register renaming and possible code motion past

[5] In one case it was observed that the relaxed problem lead to a worse expected final time for the schedule using list scheduling. However, this case could be resolved by using the integer linear programming scheduler mentioned "Verified Superblock Scheduling with Related Optimizations".

[6] Its size may increase by a single unconditional branching instruction after linearization. An example is given below.

[7] In this case, since the predicted-not-taken successor is not a join, the compensation code was merged with the original code during code layout.

side exits further exacerbates the problem of instruction scheduling with regard to register pressure. Moreover, the latter may further directly generate additional code with an increased chance to negatively impact other execution paths and performance in general. With the additions of this work, CompCert becomes thus even more reliable on the accuracy of static branch predictions.

Two mitigations with regard to the code size increase are introduced by this work. A compiler flag allows the register renaming and code motion below side exits optimizations to only be applied to inner loops, spanned by a superblock, that are predicted to loop. These are expected to be especially likely to be executed *and* have an outsized effect on performance compared to other superblocks. Additionally, since the existing instruction schedulers [38, 36], and the recent addition of a register-pressure aware scheduler by Nardino during an internship at Verimag, are unaware of the duplication effort required to realize their schedules, an (expensive) heuristic tries to estimate the gain in performance by allowing code motion below side exits (see section 4.2.1). A compiler flag then allows to control the accepted code-duplication-to-expected-cycle-gain cost ratio.

The actual superblock scheduling pass of Six, Gourdin, Boulmé, and Monniaux, may move instructions previously pushed downwards back above the side-exits, if possible (recall section 3.1). Guiding the code motion below side exits with the actual instruction scheduling (see section 4.2.1) algorithm used later, while computationally expensive, should avoid most of these situations.

4 Implementation

In the untrusted OCaml code, a superblock is represented by the record shown in listing 7. The `instructions` array is ordered so that we can simply iterate over them instead of walking the CFG. Information about the live registers with regard to each (side-) exit is readily available (`liveins` and `s_output_regs` respectively. `s_output_regs` stores the union of live registers of all successors.). In order to verify the transformation, our oracle combining register renaming and code motion below side exits, returns (1) the modified code i.e. modified and added instructions (2) the possibly new function entry (3) the modified `path_map` and (4) another map that guides the verifier to compare the correct paths. In practice, the function and path entries are not modified. Instead, instructions are moved by reassigning the mapping of CFG nodes to instructions, a trick already explained and exploited in "Optimized and formally-verified compilation for a VLIW processor" [36]. Code insertion leverages this scheme further by designing a fake schedule that will insert the instructions in the correct order while maintaining the path entries and preserving the correct successor information of the last instruction in the path.

A new field was added to RTL, holding the untrusted change to the CFG and oracle-guiding information of Duplicate per function. Thus the oracle of the duplication step (see section 4.2) simply directly returns this information to the Duplicate verifier.

4.1 Register renaming

4.1.1 Basic superblock-local register renaming

Implementing the systematic renaming itself is quite simple: For each instruction in the superblock, replace redefinitions of registers by fresh registers and keep track of the last renaming with a simple map. The set of initially live registers can be obtained from the `path_info` entry in

```
1 type superblock =
2   { mutable instructions: P.t array
3   ; mutable liveins: Regset.t PTree.t
4   ; s_output_regs: Regset.t;
5   ; typing: RTLtyping.regenv }
```

Listing 7: Representation of superblocks in the OCaml oracles. `instructions` holds the PCs of the superblock's instructions. `liveins` maps from PCs of side-exits to registers which are live in the unpredicted part of the branch. `s_output_regs` contains the union of all successors of the last instruction, that is all registers that are live at the end of the superblock.

the `path_map` of the function. Each use of the old register name will then be replaced by the most recent renaming. At this point the superblock is internally consistent but not correct with regard to its successors. In order to repair the superblock, the currently live renames are recorded for each (side-) exit. Because the liveness information for each side exit is available, this snapshot only records the information necessary to restore live registers. Eventually, before each (side-) exit, a series of instructions restoring the expected value is inserted (*restoration code*, cf. fig. 2.1 and in particular the restoration instruction.). The final restoration code represents an exception, where restoration code may be placed after the originally final instruction, if it is either basic or a side-exit. In the latter case, restoration code that is not live with regard to the predicted successor can essentially be moved out of the superblock for free.

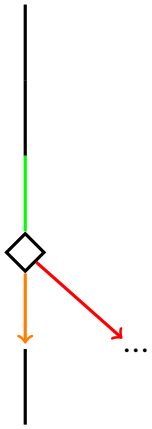
Up to this point, the register renaming is entirely provable—without any modifications—by the symbolic execution of section 1.3.2.

4.1.2 Moving restoration code outside of the superblock

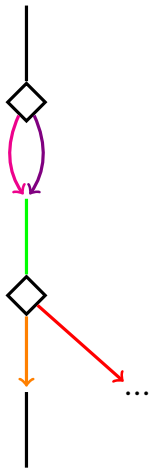
The same technique that will be described in section 4.2 is applicable to moving restoration code outside of the superblock. While this initially leaves a duplicate inside the superblock, the restoration code inside the superblock will become useless and will be removed by DCE eventually.

The final restoration code of the superblock cannot be moved outside of it. Although some or all of it *might* vanish due to register allocation, it might also remain. Therefore, the instruction schedulers of section 4.2 are conservatively given a superblock with the final restoration code, *but not the one for side exits*, inserted. The map of live renamings, storing information about the necessary renames per (side-) exits is changed to reflect the partial restoration. In combination with moving the restoration code of side exits outside of the superblock, this gives the actual prepass scheduling the opportunity to move the final and unavoidable restoration code as early as possible. However, care must be taken in case the code motion below side exits decides to move instructions below multiple side exits. Note that this can even happen when systematic register renaming is applied. A renamed instruction may be moved across multiple side exits, creating two or more copies before multiple side exits. The second occurrence may rely on its argument being constant, which was originally true for e.g. an input register to the superblock whose redefinition was then renamed. After inserting the restoration code, however, the value incorrectly assumed to be immutable has changed.

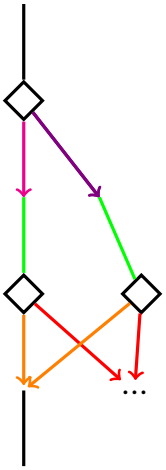
More generally, the arguments of duplicated instructions are not allowed to change before the duplicated occurrence. Therefore, when inserting the full restoration code, it is checked whether the restored register is read after its restoration. Currently, this check is implemented with a simple walk of the superblock, starting just after the insertion site of the restoration instructions. If the restoration code would change the data flow, an alias is created just before the restoration and the new name applied to (a) the remainder of the superblock when systematic register renaming is activated. Or (b) until the next definition is encountered, in the case that code motion below side exits is enabled without systematic register renaming.



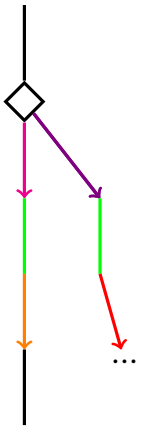
(a) The code to be moved below the side exit (shown in green) has been placed and appropriately marked just before the side exit (the diamond shape).



(b) The conditional branch of the side exit is cloned but changed to be "useless". This version of the superblock passes the modified symbolic execution semantics.



(c) The code that has been framed between the cloned and parental conditional branch is duplicated. This is easily verifiable by the existing Duplicate verifier.



(d) Finally, a modified pass of CSE3 can remove the now redundant parental `Icond` and its duplicate. For each of the branches the result is known and does not need to be recomputed. A pass of DCE is necessary in order to remove left-over instructions when instructions are moved across multiple side exits at once.

Figure 4.1: Sketch on how to move code below side exits. The diamond shape signifies a condition with the two arrows coming out of it being the two branches.

4.2 Code motion past side exits

Moving live code below an `Icond` (shown in green in fig. 4.1), and thus interposing it between the side exit and its target, is deceptively simple. First, clone the `Icond` and "frame" the code between the cloned `Icond`—whose two successor pointers both point to the beginning of the code to be duplicated—and the original `Icond` (fig. 4.1b). Afterwards, the code up to and including the original `Icond` may be duplicated. Intuitively, both initially useless branches point to the exact same code in the beginning. After duplication they still point to identical, if duplicated, code after the duplication (fig. 4.1c). This change is verifiable via the Duplicate verifier [36, pp. 109–111] without modifications. Finally, both the original `Icond` and its duplicate (NB: *not* its clone) can be removed since the result of the condition is known (fig. 4.1d) after computing the evaluating the cloned `Icond`. One of them will unconditionally branch to the original `ifso` successor and the other unconditionally branch to the `ifnot` successor. This step requires a restricted version of CSE3[1] [32, p. 9], that *only* removes redundant conditional branches. When instructions are moved downwards across multiple side exits, that their effect is live to, multiple duplicates are created that are intentionally redundant. Following the duplication step (cf. fig. 4.1c), they are no longer redundant, and their duplicate version inside the superblock will be dead. After the actual instruction scheduling, another pass of CSE3 is run, in case final restoration code of the superblock is moved so far up that its effect makes some restoration code inside the side exits redundant. Alternatively, between the code motion past side exits and actual prepass scheduling, the heuristic (section 4.2.1) and the actual schedule may disagree, in which case instruction that can still be moved up obviate previously inserted restoration code. This process, minus the cleanup phase, is visualized in fig. 4.1, and the changes necessary to verify are discussed in section 4.3.

The information, (modified code and information necessary to guide the Duplicate verifier to prove its correctness) necessary to perform the duplication of the if-branches is prepared at the RTLpath level by slightly adapting the existing function to clone code. Instead of returning the updated code, only the necessary changes are returned (a diff of sorts). Inside the trivial Duplicate oracle for this transformation the diff is simply applied.

Unfortunately, it is not possible to modify Duplicate [36, p. 109] to directly insert the `Icond` and duplicate values. Neither is it currently possible to transform multiple paths together at once, or change the control flow in the described way using the symbolic execution test. Adapting the former would require complicating the quite elegant and concise semantics, including its proven verifier. The decisive factor, in not adapting Duplicate, however, is the inability to create and insert new `Icond`s because the condition evaluation may fail. This is due to the fact that comparisons may fail if their arguments are not of the correct types. If this was the only problem, a work-around involving CompCert's guarantee that at the RTL level everything is well-typed would be possible. In this case unfortunately, pointers and integers share the same RTL type[2], and while integer comparison usually cannot fail, pointer comparisons can fail under various circumstances. The only pointer comparison allowed are comparisons between pointers of the same and correct size (64- or 32-bit according to the platform), which must point to the same mem-

[1] In contrast to CompCert's default CSE, CSE3 is a *global* CSE that can remove redundant code across conditional branches. The restricted version performs a strict subset of CSE3, but its proof is currently admitted.

[2] `Tlong` (64-bit integer) on 64-bit architectures, otherwise `Tint` on 32-bit platforms.

ory allocation and at most one index beyond the end of that allocation (in accordance with the C standard). Equality and inequality tests with NULL are also permitted. Otherwise a pointer comparison fails. As a result, the approach detailed in section 4.3 was chosen.

4.2.1 Instrumenting instruction scheduling as code duplication heuristic

As described in section 3.1, we want to instrument the actual instruction scheduler algorithm used later to infer which code duplication and eventual resulting movement below side exits yields schedules that are worth the incurred cost. Fortunately, the instruction schedulers are cleanly divided into two phases: (1) calculating the scheduling constraints (cf. section 1.2.1) and (2) the actual scheduling phase. By creating our own "frontend" to phase two, we can easily vary the dependency calculation to fit our needs.

Ignoring liveness

The existing dependency calculation walks the `instructions` array (listing 7), updating imperative state to keep track of the latest register and memory reads and writes, as well as updating the dependency constraints appropriately. When a side exit is encountered, the dependence calulation handles multiple issues:

1. Input registers that present a hard dependency ("actual" input registers) i.e. those that are required to compute the condition are added as RAW dependences.

2. Input registers, in the following called "fake", which are not read by the `Icond` itself but instead possibly read (live) for the part of the CFG pointed to by the predicted-not-taken successor are also added as RAW dependences.

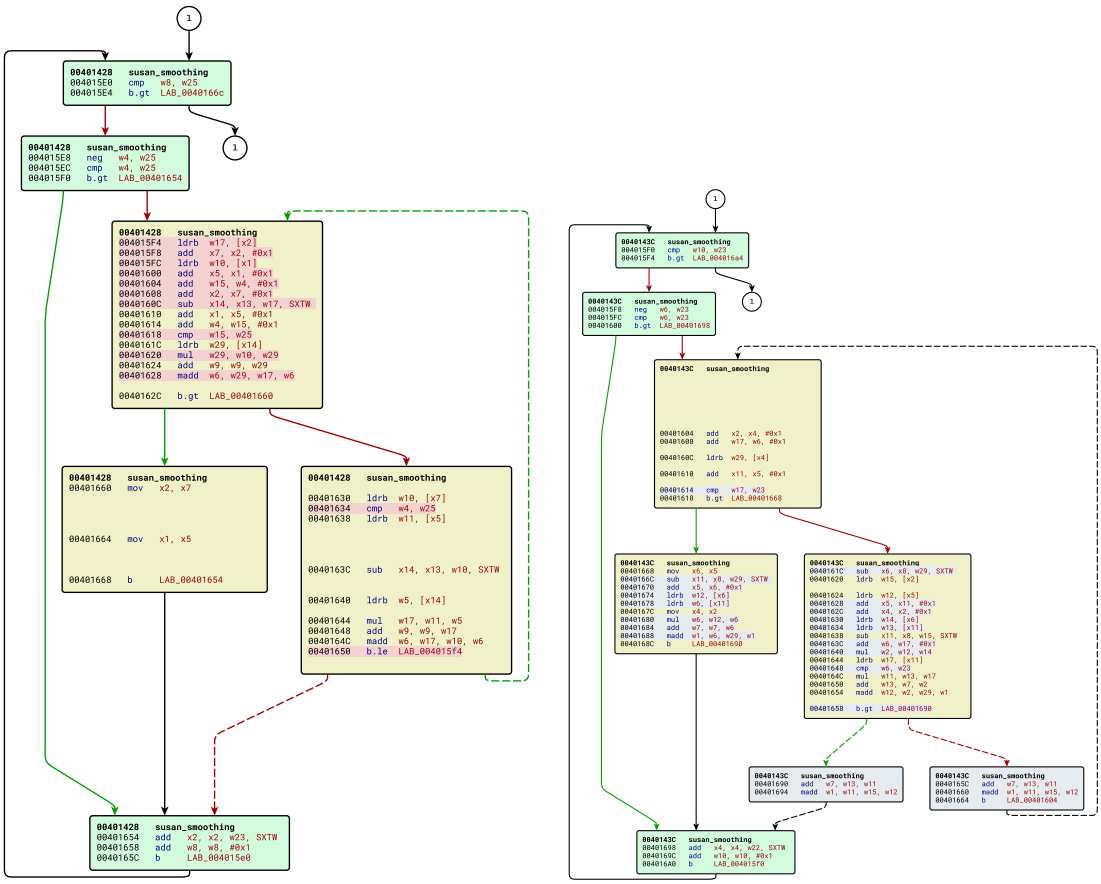
3. The side exit is recorded as the last read for both types of input registers with regard to the continuation of the superblock. This information is used to add WAR dependences to later writes in the superblock.

Simply erasing the liveness information with regard to side exits would allow impossible schedules in certain cases where a write from below the side exit is incorrectly allowed to be moved above it (see (3) above).

Instead only the RAW dependences for "fake" registers are omitted. Indeed, this relaxes the scheudling problem as desired. In order to avoid false scheduling wins when comparing the relaxed to the unrelaxed problem (see section 4.2.2), "fake" input registers are recorded with a latency of 0 which forces them to appear before side exits but without incurring the timing penalty of only scheduling the branch at a time slot for which the "fake" input is ready. Without this modification, ostensibly better schedules requiring compensation code would be incorrectly favored.

Partially ignoring memory store liveness

As explained in chapter 3, it is not possible to move stores below multiple side exits using the setup described in this chapter. Nonetheless, we *can* move memory stores across the next side exit by erasing the original instruction, a process that was subsequently applied to all instructions.



(a) Generated code when register renaming is enabled.

(b) In addition to register renaming (fig. 4.2a), the compiler is given the ability to move live instructions across side exits, at the cost of incurring compensation code.

Figure 4.2: TACLe's Smallest Univalue Segment Assimilating Nucleus (SUSAN) benchmark, compiled with tail duplication and loop unrolling and rotation. The generated code of the superblock spanning the inner loop is shown and evaluated in table 4.1. Note that register names between the two compiled version have changed. This diff was created using BinDiff [8].

Instruction		Start	Latency
ldrb	w17,[x2]	0	3
add	x7,x2,#0x1	0	1
ldrb	w10,[x1]	1	3
add	x5,x1,#0x1	1	1
add	w15,w4,#0x1	2	1
add	x2,x7,#0x1	2	1
sub	x14,x13,w17, SXTW	3	1
add	x1,x5,#0x1	3	1
add	w4,w15,#0x1	4	1
cmp	w15,w25	4	1
ldrb	w29,[x14]	5	3
mul	w29,w10,w29	8	4
add	w9,w9,w29	12	1
madd	w6,w29,w17,w6	13	4
b.gt	LAB_00401660	13	1
ldrb	w10,[x7]	14	3
cmp	w4,w25	14	1
ldrb	w11,[x5]	15	3
sub	x14,x13,w10, SXTW	15	1
ldrb	w5,[x14]	16	3
mul	w17,w11,w5	19	4
add	w9,w9,w17	23	1
madd	w6,w17,w10,w6	23	4
b.le	LAB_004015f4	**24**	1

Instruction		Start	Latency
add	x2,x4,#0x1	0	1
add	w17,w6,#0x1	0	1
ldrb	w29,[x4]	1	3
add	x11,x5,#0x1	1	1
cmp	w17,w23	2	1
b.gt	LAB_00401668	3	1
sub	x6,x8,w29, SXTW	4	1
ldrb	w15,[x2]	4	3
ldrb	w12,[x5]	5	3
add	x5,x11,#0x1	5	1
add	x4,x2,#0x1	6	1
ldrb	w14,[x6]	6	3
ldrb	w13,[x11]	7	3
sub	x11,x8,w15, SXTW	7	1
add	w6,w17,#0x1	8	1
mul	w2,w12,w14	9	4
ldrb	w17,[x11]	9	1
cmp	w6,w23	10	1
mul	w11,w13,w17	12	4
add	w13,w7,w2	12	1
madd	w12,w2,w29,w1	13	4
b.gt	LAB_00401690	13	1
add	w7,w13,w11	16	1
madd	w1,w11,w15,w12	17	4
b	LAB_00401604	**17**	1

Table 4.1: Evaluation of the superblocks of fig. 4.2 by hand. An important performance gain can be observed by improved interleaving of instructions that are now able to move below the side exit.

Encoding this constraint in the dependency calculation is slightly more complicated. Instead of adding each side exit as a blanket memory read, they are only added as memory reads to stores that have occurred one side exit before. Once again, this is implemented by splitting the constraint creating function for memory reads in two, once for actual memory reads and once for "fake" memory reads (side exits), which can be compensated. As before, fake reads are given a latency of 0.

4.2.2 Interpreting the results of relaxed scheduling problems

The actual instruction scheduling algorithms, oblivious to our trickery, report a schedule for the given superblock with the relaxed constraints. By comparing the resulting schedule with the original constraints we can observe (a) whether the relaxed problem's schedule is worth the possible code duplication (recall section 3.2) and (b) which instructions have been moved in an illegal way. Schedules are represented as arrays of PCs, in the order determined by the scheduler. Mapping the PCs of the superblock to their prior (original problem) and posterior (relaxed problem) index, it is easy to check whether a dependency of a side exit is no longer above it. Since the true input registers of side exits must be respected by the relaxed schedule as well, we can be sure that only movable instructions will be duplicated.

The following example should give the intuition. Given the dependency graph of the original code with the full constraints, we can obtain the dependencies of a side exit with PC n_{se} at index i_{se} of the instructions array. Consider the simplistic example of just one dependency with PC n_{dep} at index i_{dep}. Necessarily, $i_{dep} < i_{se}$. Having obtained the schedule (a permutation of the original array) for the relaxed problem, we can find the new indices i'_{se} and i'_{dep} of n_{se} and n_{dep} respectively. If $i'_{dep} > i'_{se}$ we can infer that the instruction at PC n_{dep} should be moved below this side exit. That is, in the relaxed problem, the instruction scheduler would not keep the compsenable dependency above the side exits.

Instead of actually recording the *instructions* to be duplicated, their PCs are. This is important because we may first need to apply the ad-hoc renaming step of chapter 2 and section 4.1.1. The connection between original and duplicated instructions is lost as soon as we insert the compensation code. Extra aliasing logic may still need to be applied after inserting the restoration code (section 4.1.1). Afterwards, the `Iconds` of side exits below which instructions are to be moved are cloned, placing the target code (to-be-moved) between them. This information is implicitly recovered by a prior insertion of an `Inop` before each side exit. If, after inserting the to-be-moved code, the `Inop` no longer directly precedes the side exit, it will be replaced by the cloned `Icond` thus correctly framing the code for duplication. Finally, the modified `code`, `path_map` and staged duplication information is returned.

4.3 Changes to RTLpath and the symbolic execution

4.3.1 Adding "useless" `Iconds`

A priori, the symbolic execution described in section 1.3.2 does not keep track of not-failing condition evaluations of `Iconds`. This was unnecessary because between the original and scheduled path (superblock): the number of side exits of the paths has to match, they must appear in

```
Definition istep (ge: RTL.genv) (i: instruction) (sp: val) (rs:
↪  regset) (m: mem): option istate :=
  match i with
  | Inop pc' => Some (mk_istate true pc' rs m)
  | Iop op args res pc' =>
      SOME v <- eval_operation ge sp op rs##args m IN
      Some (mk_istate true pc' (rs#res <- v) m)
  | Icond cond args ifso ifnot _ =>
      SOME b <- eval_condition cond rs##args m IN
      if Pos.eq_dec ifso ifnot then
        Some (mk_istate true (if b then ifso else ifnot) rs m)
      else
        Some (mk_istate (negb b) (if b then ifso else ifnot) rs m)
         ↪  (* ... *)
  end.
```

Listing 8: Modified definition of RTLpath's `istep` function (cf. listing 1). Note that for `Icond`s with identical successors, execution *always* stays within the superblock.

order, their "conditions are checked for syntactical equality, their list of arguments are compared for semantics equalities of symbolic values, and the PPAG [Preconditioned Parallel Assignment ended by Goto] are semantically compared modulo register liveness." [38, p. 18]. Normally, inserting an `Icond` would either terminate the superblock, if it is unpredicted, or introduce a new side exit, neither of which would be a legal transformation. The excerpt in listing 8 legalizes so-called "useless" `Icond`s: `Icond`s with two identical successors and thus not actually representing a meaningful branch. As explained in section 4.2, introducing such a useless `Icond` is the first step to enabling live code motion below side exits.

This change must be mirrored and proven correct for the well-formedness check since now not every `Icond` introduces a (side-) exit. Similarly, the RTLpath liveness generation needs a small adjustment. Both of these changes are rather small.

Moreover, the symbolic execution semantics need to be modified in order to specify when inserting such an `Icond` is legal. First, at an abstract level (`RTLpathSE_theory.v`): upon executing an `Icond` as part of `siexec_inst`, the non-failure of the condition evaluation is recorded as a precondition to the successful execution of the superblock (see listing 9, cf. assertion that instruction do not trap in section 1.3.2). These modified semantics must once again be proven to bisimulate the (modified) execution semantics of RTLpath (`sexec_correct` and `sexect_exact`).

Next, the data structures that will be used in the hash-consing implementation must be adjusted (`RTLpathSE_simu_specs.v`). A new kind of (ultimately hash-consed) data structure needs to be added to represent non-failing conditions, `hscond` (hash-consed symbolic condition, see listing 10 for its definition). Note that `hscond` does not depend on the memory: a pointer comparison—against whose failure we are guarding—does not need access to the content of the memory (listing 11). And since `alloc` and `free` are the only operation that can modify the allocated blocks/size simply always supplying `Sinit`, the initial symbolic memory, suffices. To

```
Definition slocal_eval_cond (st:sistate_local) (cond : condition)
  (args : list_sval) :=
  {| si_pre:=(fun ge sp rs m => seval_condition ge sp cond args
    st.(si_smem) rs m <> None
    /\ (st.(si_pre) ge sp rs m))
  ; si_sreg:=st.(si_sreg)
  ; si_smem:= st.(si_smem) |}.
```

Listing 9: Upon encountering an `Icond`, `siexec_inst` will record that its condition must not fail by adding the non-failure to the superblocks precondition. The symbolic register and memory state is left unchanged.

```
Record hscond :=
  { cond : condition
  ; lhsv : list_hsval
  ; hscond_hid : hashcode }.
```

Listing 10: Data structure to represent hash-consed condition evaluation. Note that unlike the abstract version, this data structure does not depend on the symbolic memory.

```
Notation "'seval_hscond' ge sp hsc" := (seval_condition ge sp
  hsc.(cond) (hsval_list_proj hsc.(lhsv)) Sinit)
  (only parsing, at level 0, ge at next level, sp at next level, hsc
  at next level): hse.
```

Listing 11: Symbolically evaluating an `hscond` always uses the initial symbolic memory `Sinit`.

```
Record hsistate_local :=
  { hsi_smem :> hsmem
  ; hsi_ok_lscond: list hscond
  ; hsi_ok_lsval: list hsval
  ; hsi_sreg:> PTree.t hsval }.
```

Listing 12: Hash-consed refinement of symbolic, local superblock state. The addition of a list of `hsconds` mirrors the approach taken for non-failing instructions (`hsi_ok_lsval`).

```
Definition hsilocal_simu_spec (alive: Regset.t) (hst1 hst2:
 ↪  hsistate_local) :=
     List.incl (hsi_ok_lsval hst2) (hsi_ok_lsval hst1)
  /\ List.incl (hsi_ok_lscond hst2) (hsi_ok_lscond hst1)
  /\ (forall r, Regset.In r alive -> PTree.get r hst2 = PTree.get r
   ↪  hst1)
  /\ hsi_smem hst1 = hsi_smem hst2.
```

Listing 13: Definition of what it means for two hash-consed symbolic local states to simulate each other. The second test, making sure that the transformed superblock's condition evaluations are included in the original one's has been added.

`hsistate_local` we must add the list of non-failing condition evaluations. This mirrors the approach taken for non-failing instructions (`hsi_ok_lsval`, see listing 12). The presence of an `hscond` in this list signifies that its evaluation (listing 11) does not fail[3]. In order to prove that this hash-consed representation of the local state refines the abstract symbolic local, two small lemmas had to be fixed. The refined definitions of the other states are not touched and the proofs pass without further modifications.

For the actual implementation of the simulation test, the hash-consed data structures are actually passed to the memoizing smart constructor factory of section 1.3.2. Listing 14 shows the hash-consing equality considered for our symbolic condition evaluations. The conditions, non-nested terms are compared for structural equality. The hash-consed symbolic values, on which the symbolic condition evaluation depends are simply compared by physical equality. *If* both OCaml-imported non-deterministic functions return true, the equality succeeds. Other data structures of the implementation are similarly adapted.

4.3.2 Known limitation

There is a small caveat with regard to the modified RTLpath semantics and its interaction with the scheduling oracles. In rare cases, the last side exit of a superblock is, inside the superblock, only followed by `Inops`. After scheduling it may thus be placed at the very end. If at that point both its

[3] While the `hscond` will be hash-consed, the list itself is not. Thus the list of non-failing conditions of side exits grows quadratically with the number n of side exits of a superblock. In general we expect n to be quite small. The list inclusion test used later is $\Theta(n)$, using an imperative hash-map, or $\Theta(n \log n)$, using a map based on a purely functional binary-search tree. [4, p. 39]

```
Definition hscond_hash_eq (hsc1 hsc2: hscond): ?? bool :=
  DO b1 <~ struct_eq hsc1.(cond) hsc2.(cond);;
  DO b2 <~ phys_eq hsc1.(lhsv) hsc2.(lhsv);;
  if b1 && b2 then RET true else RET false.
```

Listing 14: `hash_eq` for hash-consed conditions.

branches point to the same successor, it will be considered a useless `Icond`. Consequently, the transformed superblock has one less side exit, which is not legal. This issue is mostly[4] avoided by applying Goutagny's work on RTLtunneling, porting CompCert's previous tunneling[5], performed after register allocation, to RTL.

[4] Some programs generated by YARPGen [28] still trigger this issue.

[5] "Branch tunneling shortens sequences of branches (with no intervening computations) by rewriting the branch and conditional branch instructions so that they jump directly to the end of the branch sequence." (https://compcert.org/doc/html/compcert.backend.Tunneling.html, last accessed on 2021-08-20)

Filename	Changes to to RTLpath"	Lines changed
RTLpath.v	"Useless" `Iconds` are not early exits and do not actually branch.	25
RTLpathproof.v	No changes.	0
RTLpathWFcheck.v	"Useless" `Iconds` are not side exits, neither successor must start a new path.	10
RTLpathLivegen.v	Take these changes into account during path generation.	10
RTLpathLivegenproof.v	Trivial.	18
RTLpathSE_theory.v	The condition is evaluated and asserted not to fail but `sistate_exit` is created.	168
RTLpathSE_simu_specs.v	Add conditions as eventually hash-consed data structure. When comparing symbolic evaluations, all conditions must have been executed on the original path as well.	22
RTLpathSE_impl.v	Impelement previous checks with hash-consing in the IMPURE monad.	370
RTLpathScheduler.v	Pass along untrusted information.	2
RTLpathSchedulerproof.v	"Useless" `Iconds` are essentially a new basic instruction after which we must check equivalence if it is the final instruction of the path.	35

Table 4.2: Summary of the changes to RTLpath and its symbolic execution. Note that `hsiexec_inst_correct` in RTLpathSE_impl.v has not been proven entirely correct. The number of lines changed include a proof stub (only the cases for `Iconds` are missing), including a commented out proof stub of presumably an important lemma, that was proven earlier in a less strong form.

5 Evaluation

The theoretical advantage of the presented work is clear: with a perfect oracle, less constraints should always lead to better schedules. Right now, this fork of mainline CompCert's fork at Verimag is able to at least theoretically leverage almost the full advantages of superblock scheduling.

5.1 Experimental evaluation

None of the theoretical guarantees given by the symbolic execution (section 1.3.2) or its adaption (section 4.3) claim that the verifier is complete with regard to the oracle or that the oracle itself is correct. Indeed, we have discussed a limitation of the verifier with regard to technically correct code movement by the oracle that had to be be forbidden. However, practically speaking the translation validation does not fail for any of the tests or benchmarks given at least the compilation options of tables 5.2 and 5.3. All CompCert "compilers" (compilation with different optimization options will be considered as separate compilers), share the same set of common flags: `-fcse3-trivial-ops -funrollbody 30 -flooprotate 10 -ftailduplicate 60` in addition to standard flags related to the benchmarks[1].

The benchmark suite, comprises 154 benchmarks taken from PolyBench[2], TACLeBench [10], MiBench [17], Lustre examples and some benchmarks curated at Verimag. A Raspberry Pi 3 Model B Plus Rev 1.3 with a Cortex A53 processor was used as test platform. It runs at 1.5 GHz with an in-order eight-stage deep pipeline, issues instructions dynamically up to two at a time [33, p. 355]. The cycle estimates of table 4.1 are based on the same assumption as the postpass scheduler uses in Verimag's fork of mainline CompCert.

After running the entire benchmark suite 31 times, throwing away the first measurements to guard again thermal throttling issues during the benchmarks (or rather force equal thermal throttling). The raw performance metrics are measured in cycles, obtained via the CPU's cycle counter made easily accessible by an extra kernel module[3]. Afterwards, the relative standard deviation (RSD) was calculated as $100\frac{\sigma}{\mu}$ for each benchmark and compiler. Benchmarks with an RSD above 2 for any of the compilers were filtered out, leaving 44 benchmarks. For these benchmarks the 30 measurements where averaged and the relative relative difference calculated to a baseline version of the compiler (see figs. 5.2 and 5.3).

The value of 3, when limiting the amount of duplications caused by code motion below side exits, was picked rather arbitrarily. Intuitively, we want this number to be somewhat small in order

[1] For example, the flag `-DSMALL_DATASET` is passed, telling PolyBench to use its small data set (as opposed to mini, standard or large).

[2] http://web.cs.ucla.edu/~pouchet/software/polybench/, last accessed on 2021-08-22

[3] https://github.com/jerinjacobk/armv8_pmu_cycle_counter_el0

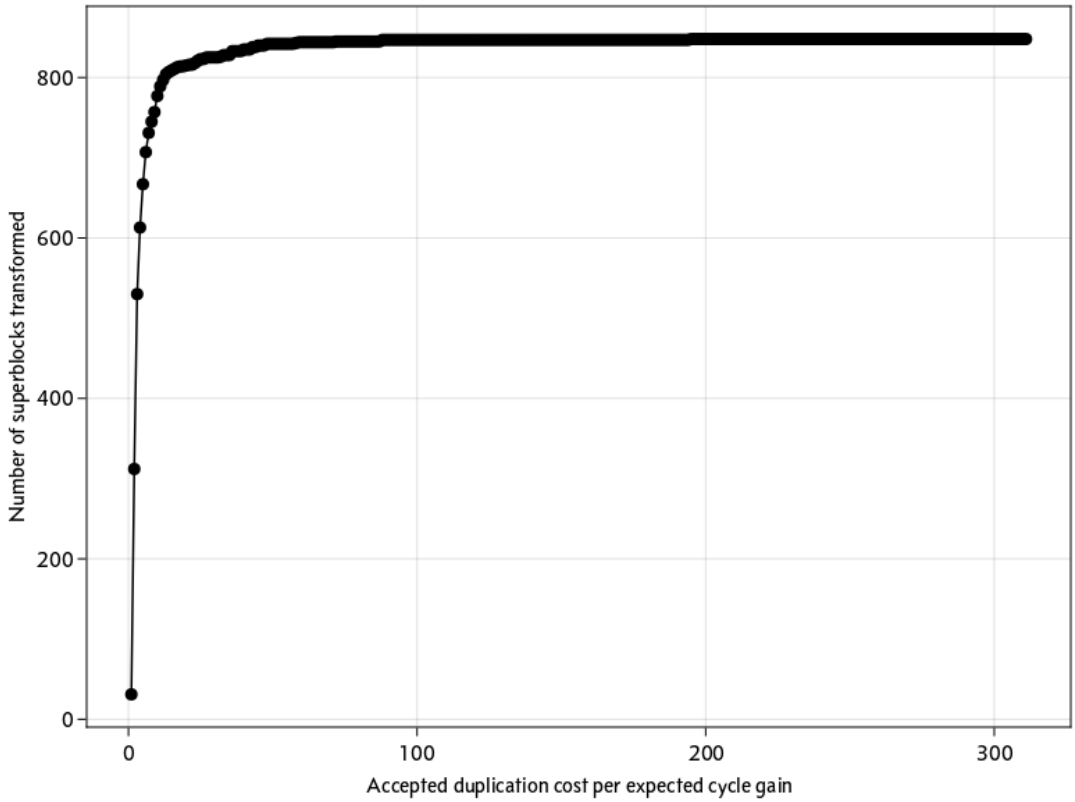


Figure 5.1: Number of superblocks transformed when limiting the amount of instructions duplicated per expected cycles gained due to the duplication. Note that the first data point sits at 1.

Abbreviation	Meaning
pp(**n**|**l**|**b**|**r**)	Prepass scheduling: **N**one, **L**ist, **B**ackwards, **R**egister-pressure aware.
rr	If present then register renaming is turned on.
li	"lift if", code can be moved below side exits. In combination with rr it moves restoration code of side exits out of the superblock.
ms	If present (must be combined with li/liN), instructions are allowed to move below side exits.
liN	When used with ms, specifies the maximum ratio between duplicated instructions and expected gain in cycles.
il	If present, the extra optimizations (rr/ms) are only applied to inner loops predicted to loop.

Table 5.1: Key to decipher compiler optimizations.

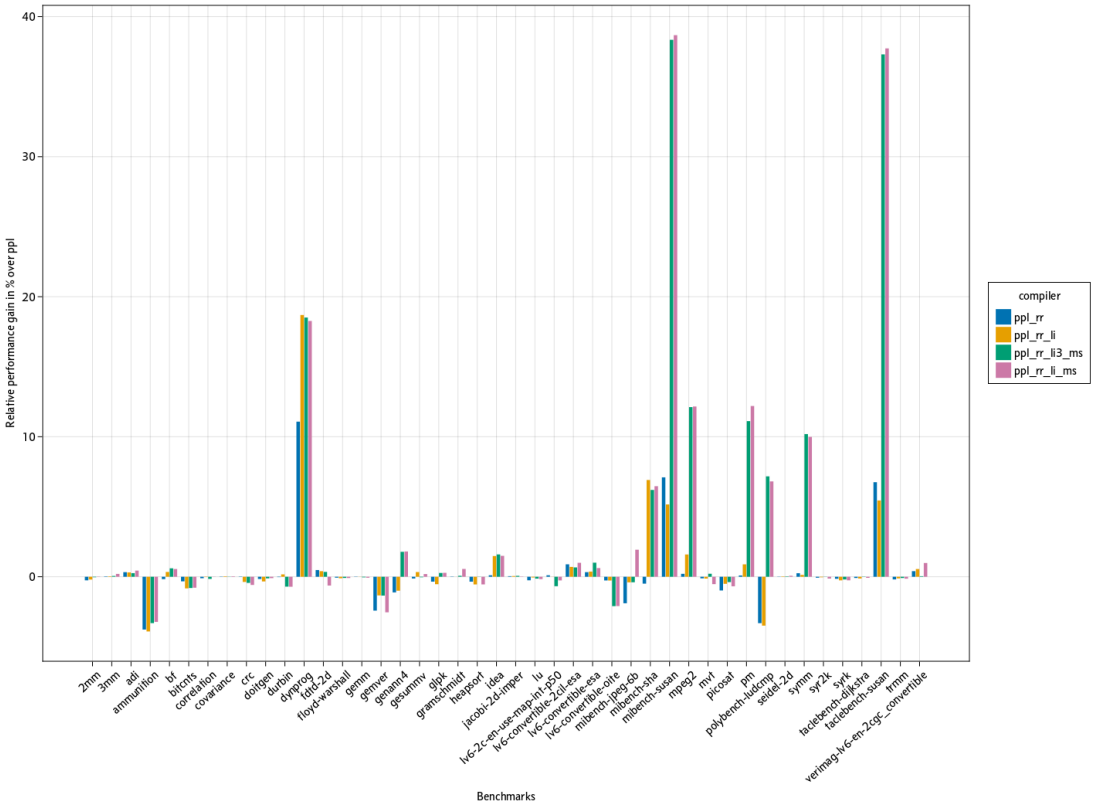


Figure 5.2: Selection of compilers plotted for easier visualization of results. Note the handful of benchmarks with important performance gains. Higher is better.

to avoid "excessive" code duplication. On the other hand, especially in this context, we still want to apply the optimization to many superblocks. Hence, 3 was picked as a trade-off (see fig. 5.1).

5.1.1 Runtime performance

Looking at fig. 5.2 we can see that register renaming and code motion past side exits can have an important influence on the runtime performance in certain cases (cf. the unimpressive mean and median columns of table 5.2). Note that SUSAN appears twice, once as part of MiBench, and once as part of TACLeBench. Predictably, and reassuringly, the performance results are very similar. SUSAN is particularly interesting because its code size increase is relatively small. To investigate where the performance gain comes form, the binaries of SUSAN when compiling just with register renaming ("ppl_rr_li") and when additionally allowing code motion ("ppl_rr_li_ms") were compared using BinDiff 6 [8] after exporting Ghidra's analysis results with the BinExport extension[4]. The code part of the CFG shown in fig. 4.2 has been discovered with BinDiff and we attribute to it the major gain in performance. Only one other part of the CFG showed noticeable

[4] https://github.com/google/binexport

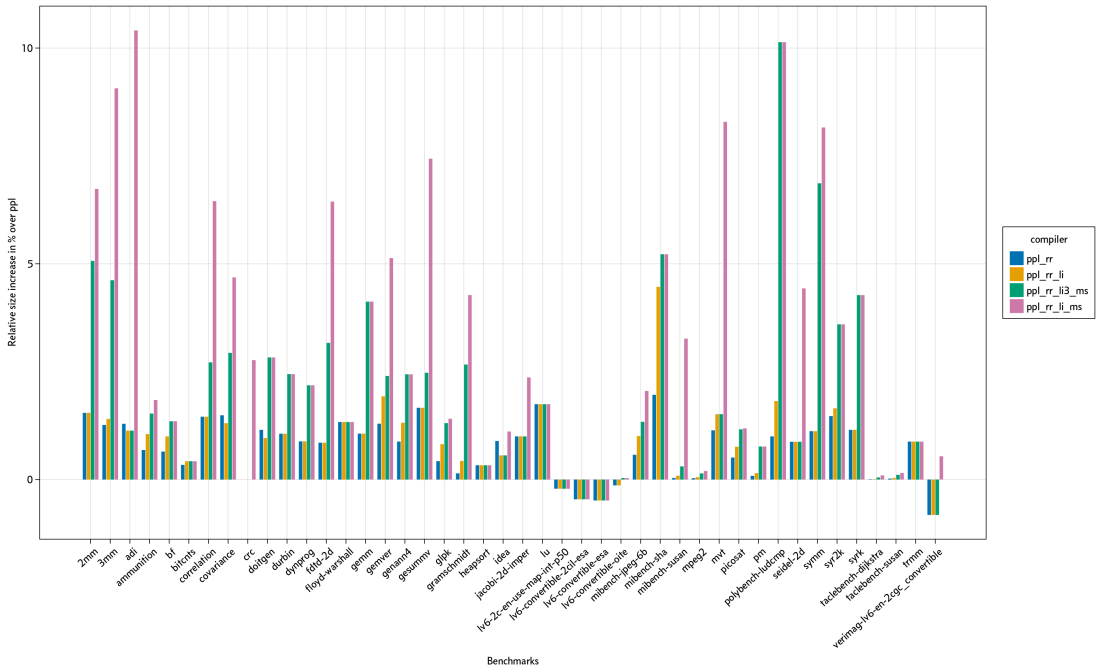


Figure 5.3: Relative code size increase to just the default compilation options for these benchmarks. Lower is better.

differences, but the same version compiled while limiting the accepted duplication cost to 3, only displayed the former change with similar performance. In table 4.1 the pipeline of the Cortex-A53 was simulated by hand and shows an important gain in cycles for this inner loop, showcasing the advantageous interleaving enabled by moving live code below a side exit. Other benchmarks showed important improvements for register renaming as well.

For many benchmarks there is no important change in performance to observe, suggesting that the heuristics of section 4.2.1 work decently well.

5.1.2 Code size increase

With regard to code size on the other hand, almost all of the benchmarks show important increase in the relative code size. Code size is measured by counting the lines of assembly as assembly and linking are outside the control of CompCert and add serious variability. It is interesting to note that the relatively large increase in code size does not seem to impact the performance too much. One possibility is that the code size increase mostly happens outside of important superblocks and/or loops so that the instruction cache is serious too much impacted. Figure 5.3 presents the increase in code size over compiling without register renaming and code motion below side exits.

Compiler	Mean	SD	Min	Q25	Median	Q75	Max
gcc_O1	7.6388	14.5364	-14.9958	-0.358799	5.85085	14.0446	64.8289
gcc_O2	20.6496	26.7684	0.134864	7.44226	11.3749	21.6997	150.406
ppn	-3.40771	4.26335	-16.6199	-5.36854	-3.45005	-0.15971	6.99419
ppn_rr	-3.3006	4.74266	-18.7174	-5.29553	-2.66983	-0.0992033	6.54556
ppb	-1.24036	4.56508	-22.831	-2.68629	-0.180271	0.308892	6.3524
ppr	-0.073096	0.274274	-0.739752	-0.139261	-0.0299136	0.0455517	0.778176
ppl_rr	0.253656	2.43581	-3.7699	-0.258335	-0.0663835	0.106016	11.0758
ppb_rr	-1.04864	4.49441	-20.9926	-2.8004	-0.21653	0.198317	6.73587
ppr_rr	0.182216	2.4542	-3.75183	-0.408277	-0.0743278	0.0484149	11.4343
ppl_rr_li	0.660213	3.31885	-3.90563	-0.285752	-0.0057986	0.349426	18.6909
ppl_rr_li_il	-0.070214	0.658057	-1.65598	-0.158776	-0.0399422	0.0535289	3.32184
ppb_rr_li	-1.8191	4.80041	-24.5397	-3.45424	-0.42696	0.0174452	6.74696
ppr_rr_li	0.562115	3.30297	-3.85031	-0.362995	-0.0120634	0.290906	18.5492
ppl_rr_li_ms	3.15595	8.80805	-3.22288	-0.256543	0.0262018	1.11994	38.6752
ppl_rr_li_ms_il	-0.0721444	0.703755	-1.91889	-0.219389	-0.0695988	0.0441982	3.45987
ppb_rr_li_ms	1.22705	9.27699	-19.6032	-1.91012	-0.146475	1.06919	39.5203
ppr_rr_li_ms	2.92591	8.51032	-3.22139	-0.367747	0.0252563	1.27369	37.1793
ppl_rr_li3_ms	3.1115	8.71975	-3.31072	-0.138454	0.0226524	0.750526	38.3435
ppb_rr_li3_ms	1.25772	9.17075	-18.8024	-1.66887	-0.145992	1.10096	39.2114
ppr_rr_li3_ms	2.86386	8.51221	-3.30564	-0.274683	-0.0235695	0.488564	37.3514
ppl_li_ms	0.27112	4.69161	-15.6833	-0.183128	0.0114277	0.763195	12.8895
ppb_li_ms	-0.491917	6.36201	-15.5361	-2.7751	-0.108018	0.299748	16.3388
ppr_li_ms	0.224586	4.69652	-15.7203	-0.259994	-0.012063	0.713992	12.8301
ppl_li3_ms	-0.0839534	4.3917	-15.3511	-0.123845	-0.0113259	0.116377	13.0092
ppb_li3_ms	-0.408243	6.37275	-15.5132	-2.70171	-0.0607288	0.355513	16.141
ppr_li3_ms	-0.103613	4.51028	-15.7118	-0.220621	-0.0612546	0.209797	12.9947

Table 5.2: Summary of performance benchmarks. The key is given in table 5.1.

Compiler	Mean	SD	Min	Q25	Median	Q75	Max
gcc_O1	-66.0654	245.387	-1210.79	-16.2458	-8.48005	-4.67762	8.96157
gcc_O2	-71.5867	249.573	-1249.74	-23.5247	-17.5077	-12.6662	38.5971
ppb	0.097385	0.948883	-5.39671	0.0	0.0955621	0.376648	2.4873
ppb_li3_ms	0.409494	1.17012	-5.5476	0.0670706	0.364445	0.706278	3.41463
ppb_li_ms	0.458691	1.19714	-5.5476	0.0808104	0.379297	0.822077	3.41463
ppb_rr	0.308986	1.10511	-5.5476	0.0550055	0.293114	0.554402	3.07443
ppb_rr_li3_ms	1.57274	1.83256	-5.66105	0.560658	1.17985	2.78572	5.51817
ppb_rr_li	0.492641	1.25282	-5.5476	0.113363	0.535142	0.68902	4.61783
ppb_rr_li_ms	1.58262	1.82967	-5.66105	0.560658	1.2363	2.78572	5.51817
ppl_li3_ms	0.315715	1.11001	-0.717213	0.0	0.0	0.202116	6.01266
ppl_li_ms	0.882762	1.51656	-0.485642	0.0	0.126799	1.58075	7.62053
ppl_rr	0.731947	0.663163	-0.820513	0.129327	0.880089	1.18018	1.96399
ppl_rr_li3_ms	1.95622	2.13583	-0.820513	0.401118	1.34449	2.74429	10.1362
ppl_rr_li	0.880711	0.874215	-0.820513	0.285794	0.980769	1.32018	4.46571
ppl_rr_li_il	0.0260701	0.108609	-0.0125219	0.0	0.0	0.0	0.668896
ppl_rr_li_ms	3.21504	3.00893	-0.485642	0.850749	2.40294	4.79656	10.4106
ppl_rr_li_ms_il	0.0260701	0.108609	-0.0125219	0.0	0.0	0.0	0.668896
ppn	0.00677172	1.01803	-6.38528	0.0	0.173663	0.360794	0.44843
ppn_rr	0.122665	0.989112	-5.69892	0.0	0.182006	0.566199	1.28913
ppr	0.0249667	0.225173	-0.340252	0.0	0.0	0.0	1.32932
ppr_li3_ms	0.392023	1.0948	-0.212993	0.0	0.0	0.265425	6.01266
ppr_li_ms	0.850847	1.49502	-0.854993	0.0	0.101193	1.58075	7.62053
ppr_rr	0.85587	0.674165	-0.212993	0.337667	0.888906	1.27247	2.98013
ppr_rr_li3_ms	2.08116	2.07185	-0.212993	0.777462	1.43303	2.85659	10.1362
ppr_rr_li	1.00211	0.859299	-0.212993	0.428924	1.00032	1.35112	4.46571
ppr_rr_li_ms	3.29533	3.04364	-0.958576	1.05497	2.40294	5.15452	10.4106

Table 5.3: Summary of code size increase on benchmarks. The key is given in table 5.1.

6 Discussion

The first part of this work, implementing a simple register renaming pass on top of RTLpath's symbolic execution, once again (cf. [38, pp. 8–9]) demonstrates the flexibility of the translation validation approach. Without any modifications, it is able to verify the systematic register rewriting, including the insertion of restoration code. Even when considering the adaption of RTLpath down to the implementation of hash-consed symbolic execution, the burden of refactoring and proof has been modest, considering the many aspects touched upon in this work.

When combined with loop optimizations verified via the Duplicate oracle, register renaming appears to have an important influence on the availability of ILP.

The problem of register allocation is central to compilation and in the form present in CompCert NP-complete [7]. An important drawback of the register rewriting presented here is its complex interactions with prepass instruction scheduling—where a register-pressure aware scheduler is an important step in the right direction—and register allocation. It seems very difficult to predict which renamings will be a boon to performance and which might actually make things worse. No effort has been made in this project to do so, although targeted register renaming (see section 4.1.1) could be used to single out likely candidates for effective renaming e.g. loop indices. In addition to being a more general approach to register renaming, SSA makes it possible to perform optimal register allocation in polynomial time [18]. CompCertSSA [3] does implement an SSA form for but its proof of correctness is quite large. "Their correctness proofs are harder [than non-SSA CompCert] and involve non-trivial invariants about control-flow graphs, dominance relations, etc." [32, p. 23] Nonetheless it has recently been ported to current versions of CompCert.

The benchmark results regarding the combination of register renaming and code motions below side exits are promising. They indicate that extension of prepass superblock scheduling to support code motion below side exits via compensation code is a worthwhile pursuit. At the same time there is still obvious room for improvement with regard to maximizing scheduling potential. In particular the lack of alias analysis in the symbolic execution forces many scheduling dependencies that are in principle not necessary. More immediately, enabling redundant writes to be normalized to the same symbolic memory would allow movement of stores below multiple side exits. However, how much additional flexibility it would bring and whether it could be meaningfully exploited is unclear. Previous benchmarks (not shown here) did not display remarkable performance differences between the two relaxations (section 4.2.1) that are currently compensable.

Both additions of this work are limited in their scope to superblocks, thus necessitating the impressive succession of independent passes to achieve code motion below side exits. While arguably decomposing a larger optimization into multiple smaller ones may be beneficial, in this case the setup seems more complicated than necessary. A tree or graph-like code region representation could remedy the situation. For example if both the likely successor and unlikely successor

were available, in a setup where symbolic execution is still practical and efficient, the code motion below side exits could be proved directly during scheduling. This would avoid most of the complicated aspects of this work with regard to both the composition and slight modification of many passes, as well as the relatively complicated oracle implementation presented earlier. Another interesting avenue to tackle the currently relatively myopic view of the symbolic execution would be the introduction of invariants to the concept of code regions. These and additional ideas are currently being explored by Léo Gourdin as part of his PhD thesis at VERIMAG co-supervised by Sylvain Boulmé and Frédéric Pétrot.

In conclusion, the extension of the symbolic execution semantics enabled a relatively complex, non-local, transformation where most of the complexity is handled by the oracle. The results hint at exciting possibilities with regard to oracle-guided translation validation, verified by scalable symbolic execution, particularly with regard to more powerful code region representations in the future.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley series in computer science / World student series edition. Addison-Wesley, 1986. ISBN: 0-201-10088-6. URL: https://www.worldcat.org/oclc/12285707.

[2] Andrew W. Appel. "Preface". In: *Modern Compiler Implementation in ML*. Cambridge University Press, 1997. DOI: 10.1017/CBO9780511811449.001.

[3] Gilles Barthe, Delphine Demange, and David Pichardie. "Formal Verification of an SSA-Based Middle-End for CompCert". In: *ACM Trans. Program. Lang. Syst.* 36.1 (2014), 4:1–4:35. DOI: 10.1145/2579080. URL: https://doi.org/10.1145/2579080.

[4] Sylvain Boulmé. "Formally Verified Defensive Programming. Efficient Coq-Verified Computations from Untrusted ML Oracles". Preliminary Draft for a future HDR. July 4, 2021.

[5] Sylvain Boulmé and Thomas Vandendorpe. "Embedding Untrusted Imperative ML Oracles into Coq Verified Code". working paper or preprint. July 2019. URL: https://hal.archives-ouvertes.fr/hal-02062288.

[6] Thomas Braibant, Jacques-Henri Jourdan, and David Monniaux. "Implementing and Reasoning About Hash-consed Data Structures in Coq". In: *J. Autom. Reason.* 53.3 (2014), pp. 271–304. DOI: 10.1007/s10817-014-9306-0. URL: https://doi.org/10.1007/s10817-014-9306-0.

[7] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. "Register Allocation Via Coloring". In: *Comput. Lang.* 6.1 (1981), pp. 47–57. DOI: 10.1016/0096-0551(81)90048-5. URL: https://doi.org/10.1016/0096-0551(81)90048-5.

[8] Thomas Dullien and Rolf Rolles. "Graph-based comparison of Executable Objects". In: Symposium sur la sécurité des technologies de l'information et des communications. Association STIC c/o CentraleSupélec, 2005. URL: https://www.sstic.org/2005/presentation/bindiff_compairson_struct_objets_exec/.

[9] Eric Eide and John Regehr. "Volatiles are miscompiled, and what to do about it". In: *Proceedings of the 8th ACM & IEEE International conference on Embedded software, EMSOFT 2008, Atlanta, GA, USA, October 19-24, 2008*. Ed. by Luca de Alfaro and Jens Palsberg. ACM, 2008, pp. 255–264. DOI: 10.1145/1450058.1450093. URL: https://doi.org/10.1145/1450058.1450093.

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

[11] Joseph A. Fisher. "Trace Scheduling: A Technique for Global Microcode Compaction". In: *IEEE Trans. Computers* 30.7 (1981), pp. 478–490. DOI: 10.1109/TC.1981.1675827. URL: https://doi.org/10.1109/TC.1981.1675827.

[12] Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Kiem-phong Vo. "A Technique for Drawing Directed Graphs". In: *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING* 19.3 (1993), pp. 214–230.

[13] Emden R. Gansner and Stephen C. North. "An open graph visualization system and its applications to software engineering". In: *SOFTWARE - PRACTICE AND EXPERIENCE* 30.11 (2000), pp. 1203–1233.

[14] Shilpi Goel. "Formal verification of application and system programs based on a validated x86 ISA model". PhD thesis. University of Texas at Austin, 2016. URL: https://repositories.lib.utexas.edu/handle/2152/46437.

[15] David Gregg. "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*. Ed. by Reinhard Wilhelm. Vol. 2027. Lecture Notes in Computer Science. Springer, 2001, pp. 200–212. DOI: 10.1007/3-540-45306-7_14. URL: https://doi.org/10.1007/3-540-45306-7%5C_14.

[16] Dick Grune, Kees van Reeuwijk, Henri E. Bal, and Koen Jacobs Ceriel J.H. d Langendoen. *Modern Compiler Design*. Springer New York, 2012. DOI: 10.1007/978-1-4614-4699-6. URL: http://dx.doi.org/10.1007/978-1-4614-4699-6.

[17] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown. "MiBench: A free, commercially representative embedded benchmark suite". In: *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*. 2001, pp. 3–14. DOI: 10.1109/WWC.2001.990739.

[18] Sebastian Hack and Gerhard Goos. "Optimal Register Allocation for SSA-form Programs in polynomial Time". In: *Information Processing Letters* 98.4 (May 2006), pp. 150–155. DOI: 10.1016/j.ipl.2006.01.008.

[19] William A. Havanki, Sanjeev Banerjia, and Thomas M. Conte. "Treegion Scheduling for Wide Issue Processors". In: *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture, Las Vegas, Nevada, USA, January 31 - February 4, 1998*. IEEE Computer Society, 1998, pp. 266–276. DOI: 10.1109/HPCA.1998.650566. URL: https://doi.org/10.1109/HPCA.1998.650566.

[20] John L. Hennessy and David A. Patterson. *Computer Architecture. A Quantitative Approach*. 6th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017. ISBN: 9780128119051.

[21] Wen-mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M. Lavery. "The superblock: An effective technique for VLIW and superscalar compilation". In: *J. Supercomput.* 7.1-2 (1993), pp. 229–248. DOI: 10.1007/BF01205185. URL: https://doi.org/10.1007/BF01205185.

[22] Daniel Kästner, Jörg Barrho, Ulrich Wünsche, Marc Schlickling, Bernhard Schommer, Michael Schmidt, Christian Ferdinand, Xavier Leroy, and Sandrine Blazy. "CompCert: Practical Experience on Integrating and Qualifying a Formally Verified Optimizing Compiler". In: *ERTS2 2018 - 9th European Congress Embedded Real-Time Software and Systems*. 3AF, SEE, SIE. Toulouse, France, Jan. 2018, pp. 1–9. URL: https://hal.inria.fr/hal-01643290.

[23] Daniel Kästner, Xavier Leroy, Sandrine Blazy, Bernhard Schommer, Michael Schmidt, and Christian Ferdinand. "Closing the Gap – The Formally Verified Optimizing Compiler CompCert". In: *SSS'17: Safety-critical Systems Symposium 2017*. Developments in System Safety Engineering: Proceedings of the Twenty-fifth Safety-critical Systems Symposium. Bristol, United Kingdom: CreateSpace, Feb. 2017, pp. 163–180. URL: https://hal.inria.fr/hal-01399482.

[24] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. "seL4: formal verification of an OS kernel". In: *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*. Ed. by Jeanna Neefe Matthews and Thomas E. Anderson. ACM, 2009, pp. 207–220. DOI: 10.1145/1629575.1629596. URL: https://doi.org/10.1145/1629575.1629596.

[25] Monica S. Lam. "Software Pipelining: An Effective Scheduling Technique for VLIW Machines". In: *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988*. Ed. by Richard L. Wexelblat. ACM, 1988, pp. 318–328. DOI: 10.1145/53990.54022. URL: https://doi.org/10.1145/53990.54022.

[26] Xavier Leroy. *Commented Coq development*. CompCert Version 3.9. May 10, 2021. URL: https://compcert.org/doc/ (visited on 05/26/2021).

[27] Xavier Leroy. "Formal verification of a realistic compiler". In: *Commun. ACM* 52.7 (2009), pp. 107–115. DOI: 10.1145/1538788.1538814. URL: https://doi.org/10.1145/1538788.1538814.

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

[29] Roberto Castañeda Lozano, Mats Carlsson, Gabriel Hjort Blindell, and Christian Schulte. "Combinatorial Register Allocation and Instruction Scheduling". In: *ACM Trans. Program. Lang. Syst.* 41.3 (2019), 17:1–17:53. DOI: 10.1145/3332373. URL: https://doi.org/10.1145/3332373.

[30] Roberto Castañeda Lozano and Christian Schulte. "Survey on Combinatorial Register Allocation and Instruction Scheduling". In: *ACM Comput. Surv.* 52.3 (2019), 62:1–62:50. DOI: 10.1145/3200920. URL: https://doi.org/10.1145/3200920.

[31] Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, and Roger A. Bringmann. "Effective compiler support for predicated execution using the hyperblock". In: *Proceedings of the 25th Annual International Symposium on Microarchitecture, Portland, Oregon, USA, November 1992*. Ed. by Wen-mei W. Hwu. ACM / IEEE Computer Society, 1992, pp. 45–54. DOI: 10.1109/MICRO.1992.696999. URL: https://doi.org/10.1109/MICRO.1992.696999.

[32] David Monniaux and Cyril Six. "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*. Ed. by Jörg Henkel and Xu Liu. ACM, 2021, pp. 85–96. DOI: 10.1145/3461648.3463850. URL: https://doi.org/10.1145/3461648.3463850.

[33] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware Software Interface ARM Edition*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2016. ISBN: 0128017333.

[34] Silvain Rideau and Xavier Leroy. "Validating Register Allocation and Spilling". In: *Compiler Construction, 19th International Conference, CC 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*. Ed. by Rajiv Gupta. Vol. 6011. Lecture Notes in Computer Science. Springer, 2010, pp. 224–243. DOI: 10.1007/978-3-642-11970-5_13. URL: https://doi.org/10.1007/978-3-642-11970-5%5C_13.

[35] Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. "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*. Ed. by Hans-Juergen Boehm and Cormac Flanagan. ACM, 2013, pp. 471–482. DOI: 10.1145/2491956.2462183. URL: https://doi.org/10.1145/2491956.2462183.

[36] Cyril Six. "Optimized and formally-verified compilation for a VLIW processor". PhD thesis. Kalray and Université Grenoble Alpes, July 13, 2021.

[37] Cyril Six, Sylvain Boulmé, and David Monniaux. "Certified and efficient instruction scheduling: application to interlocked VLIW processors". In: *Proc. ACM Program. Lang.* 4.OOPSLA (2020), 129:1–129:29. DOI: 10.1145/3428197. URL: https://doi.org/10.1145/3428197.

[38] Cyril Six, Léo Gourdin, Sylvain Boulmé, and David Monniaux. "Verified Superblock Scheduling with Related Optimizations". working paper or preprint. Apr. 2021. URL: https://hal.archives-ouvertes.fr/hal-03200774.

[39] Jean-Baptiste Tristan, Paul Govereau, and Greg Morrisett. "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*. Ed. by Mary W. Hall and David A. Padua. ACM, 2011, pp. 295–305. DOI: 10.1145/1993498.1993533. URL: https://doi.org/10.1145/1993498.1993533.

[40] Jean-Baptiste Tristan and Xavier Leroy. "Formal verification of translation validators: a case study on instruction scheduling optimizations". In: *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*. Ed. by George C. Necula and Philip Wadler. ACM, 2008, pp. 17–27. DOI: 10.1145/1328438.1328444. URL: https://doi.org/10.1145/1328438.1328444.

[41] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. "Finding and understanding bugs in C compilers". In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*. Ed. by Mary W. Hall and David A. Padua. ACM, 2011, pp. 283–294. DOI: 10.1145/1993498.1993532. URL: https://doi.org/10.1145/1993498.1993532.