Formally Verifying Optimizations with Block Simulations

Léo Gourdin, Benjamin Bonneau, Sylvain Boulmé, David Monniaux, Alexandre Bérard

Formally Verifying Optimizations with Block Simulations

LÉO GOURDIN, BENJAMIN BONNEAU, SYLVAIN BOULMÉ, DAVID MONNIAUX, and ALEXANDRE BÉRARD

CompCert (ACM Software System Award 2021) is the first industrial-strength compiler with a mechanically checked proof of correctness. Yet, CompCert remains a moderately optimizing C compiler. Indeed, some optimizations of "`gcc -O1`" such as *Lazy Code Motion* (LCM) or *Strength Reduction* (SR) were still missing: developing these efficient optimizations together with their formal proofs remained a challenge.

Cyril Six et al. have developed efficient formally verified translation validators for certifying the results of superblock schedulers and peephole optimizations. We revisit and generalize their approach into a framework (integrated into CompCert) able to validate many more optimizations: an enhanced superblock scheduler, but also *Dead Code Elimination* (DCE), *Constant Propagation* (CP), and more noticeably, LCM and SR. In contrast to other approaches to translation validation, we co-design our untrusted optimizations and their validators. Our optimizations provide hints, in the forms of *invariants* or *CFG morphisms*, that help keep the formally verified validators both simple and efficient. Such designs seem applicable beyond CompCert.

1 CONTEXT AND MOTIVATIONS

Compilers are complex pieces of software, and are thus likely to contain bugs. Some bugs result in the compiler crashing or aborting, some in missed optimizations, and some, more annoyingly, result in miscompilation: the generated code is incorrect. Many miscompilation bugs have been found in the optimization phases of major compilers such as GCC and LLVM [Zhou et al. 2021].

Translation validation, as coined and popularized by Pnueli et al. [1998], is a defensive approach for ruling out miscompilation. It consists in validating, during or after compilation, that the *target code* (of a single transformation or even for a whole compilation run) matches the *source code*. Hence, the correctness of the compiler is reduced to that of the validators. This is likely simpler and more maintainable than proving the correctness of each individual optimization, since a given validator may be able to validate a whole class of transformations, and may be robust to optimization tuning.

Checking semantic equivalence between two code fragments is undecidable in general. Yet, two major successes of software formal verification have leveraged translation validation. CompCert [Leroy 2009a] uses a specialized validator to check register allocation [Rideau and Leroy 2010]. The validator is specific to this compilation phase. In contrast, seL4 [Sewell et al. 2013] attempts to automatically match the C source and object code of the seL4 kernel (itself proved correct w.r.t. a high-level specification); the resulting verification conditions are discharged by an SMT solver.

These two projects had very different constraints. The seL4 validation team had to work with an existing compiler, which was not to be modified; but they could write the software to be compiled in a certain way that helped with the "matching", and they could tune per-module optimization options if needed. Their scheme is unlikely to work with other programs, or even with other compiler versions, unless these programs or the matching scheme are manually modified.[1] In contrast, CompCert was (informally) expected to compile arbitrary source programs without failure;[2] but code transformations and validators were designed together. In such a context, it is possible to have

[1] According to [Sewell et al. 2013, §4.2], the translation validation of seL4 is very unstable w.r.t. the version of GCC.
[2] This is another argument against general-purpose translation validation based on SMT-solving for compilation of many different and evolving code bases: SMT solvers tend to be brittle, changes in solver version or minor changes in the source program may result in the solver timing out on validation problems that it could previously discharge.

Authors' address: Léo Gourdin, Leo.Gourdin@univ-grenoble-alpes.fr; Benjamin Bonneau, Benjamin.Bonneau@ens.psl.eu; Sylvain Boulmé, Sylvain.Boulme@univ-grenoble-alpes.fr; David Monniaux, David.Monniaux@univ-grenoble-alpes.fr; Alexandre Bérard, alexandre.berard2@etu.univ-grenoble-alpes.fr.

the code transformation leave hints to the validator. The validator is then likely to be more robust (it need not guess how source and target match), simpler, and to perform fewer computations.

However, according to Leroy [2011], special-purpose translation validation is not a "*silver bullet*" either. Indeed, developing specific validators is tedious and expensive: they should be formally proved yet reasonably efficient, two characteristics that may be contradictory. Moreover, between ultra-specialized validators and fully-general ones, there is a continuum that remains to be systematically explored. Paving this way, Necula [2000] proposed a general-purpose but *predictable* translation validator (without a formal proof) combining *symbolic execution* [King 1976; Samet 1976] and *normalized rewriting*. Tristan and Leroy [2008] showed how to build a formally verified instruction scheduler from formally verified symbolic execution. Yet, their validator did not scale.[3] Then, they proposed [Tristan and Leroy 2009, 2010] formally verified translation validations of lazy code motion and other optimizations. None of their extensions were integrated into the public releases of CompCert.[4] Tatlock and Lerner [2010] designed an extensible CompCert by translation validation, but their validator was not formally verified, hence significantly augmenting the Trusted Computing Base.[5] The register allocator introduced by Rideau and Leroy [2010] was the first formally verified optimization by translation validation to be integrated into mainline CompCert. Its validator uses a specialized formally verified data-flow analysis, still used in current releases.

Since these works, from more than a decade ago, the efficiency of formally verified translation validation for advanced optimization, in particular in CompCert, did not progress a lot (see related works in §9), despite the interest in improving the performance of the generated code—a sign of the difficulty of the task. Recently, Boulmé, Monniaux and Six designed an efficient formally verified translation validation by symbolic execution thanks to formally verified hash-consing [Boulmé 2021; Six et al. 2020]. Then Six et al. [2022] extended the approach for validating an efficient superblock scheduling with peephole optimizations in CompCert.[6]

This paper generalizes the translation validation techniques of Six et al. [2022] to support a larger class of optimizations, including Lazy Code Motion (LCM) [Knoop et al. 1995, 1992] and Lazy Strength Reduction (LSR) [Knoop et al. 1993] which were not yet provided by CompCert. First, §2 gives a high-level introduction to our translation validators. Second, §3 details our central contribution: a new IR (Intermediate Representation) called BTL, along with a symbolic simulation validator, dedicated to "*inter-block transformations*", a concept introduced in §2.2.2. Then, three sections apply our validators to a distinct optimization: §4 to superblock scheduling, §5 to LCM, and §6 to LSR. Last, §7 briefly discusses how the translations need to be instrumented for being validated, §8 gives an experimental evaluation and §9 concludes. Our implementation represents around 12Kloc of Coq and 5Kloc of OCaml oracles, without comments and excluding the code that we directly reused from the CompCert version of Six-et-al. [7]

2 INTRODUCTION TO OUR VALIDATORS

We consider optimizations over RTL code [Leroy 2009b]. RTL (Register Transfer Language) is an IR of CompCert's backend representing each function as a CFG (Control Flow Graph) of instructions,

[3] Source code for their extensions at https://github.com/jtristan/CompCert-Extensions. Tristan and Leroy [2008, §7] mention two important algorithmic issues (including lack of hash-consing) that were solved in [Six et al. 2022].
[4] https://github.com/AbsInt/CompCert
[5] See [Monniaux and Boulmé 2022] for an extensive discussion of CompCert's TCB.
[6] Mullen et al. [2016] formally verified peephole optimizations for x86-32 within CompCert, based on the integer representation of pointers. Such low-level optimizations are out-of-scope of our work. In contrast, they do not support instruction reordering, nor loop optimizations. Moreover, they introduced a peephole execution engine with formally verified rewriting rules, but in a direct style, without translation validation.
[7] https://gricad-gitlab.univ-grenoble-alpes.fr/certicompil/Chamois-CompCert

themselves operating over an unbounded set of pseudo-registers. These pseudo-registers are allocated to the actual machine registers by a subsequent pass, called register allocation [Rideau and Leroy 2010], to another IR called LTL. Hence, RTL is convenient for "middle-end" optimizations, because they can easily introduce fresh (pseudo)registers for storing intermediate results.

We mainly combine two kinds of translation validators: the first one, described §2.1, targets code duplications or factorizations; the second one, described §2.2, targets what we call *inter-block transformations*. At high-level, each of our optimizations can be viewed as a composition of several transformations on the RTL code, with generally "preprocessing passes" (e.g. loop-unrolling or register renaming), the core of the optimization (e.g. superblock scheduling) and possibly some "post-processing passes" (e.g. code factorization). Each transformation must be checked by a validator. Distinct transformations may be checked by the same validator. If each transformation in a sequence can be checked by the same validator, then the oracles performing them can sometimes be composed into a single oracle requiring a single validator run at the end.

In CompCert, the correctness of a backend pass is established by a *forward simulation proof* [Leroy 2009b]. Each of our validators is thus formally proved to establish a forward simulation of the source code by the target code. Validation is helped by *hints* provided by oracles: information easy for the oracle to yield, but that would be hard to have the validators reconstruct.

2.1 CFG Morphisms for Duplications or Factorizations

By specialization of the notion of *graph homomorphism*, we define a *CFG morphism* between two RTL CFG as a mapping between the nodes of the CFG preserving the instruction contents with the successor relationship (and the successor orders), and also preserving the CFG entry point. Almost trivially, two CFG related by a CFG morphism are bisimulable (they are semantically equivalent). Checking that a given function is a CFG morphism is also very easy. Hence, we reuse the CFG morphism checker, initially introduced by [Six et al. 2022, §4.4].

Indeed, Six noticed that many code duplications useful for superblock scheduling—e.g. tail duplication, loop-unrollings—are CFG morphisms: their mapping is the function associating each duplicated node in the target CFG to its origin in the source CFG. Fig. 1 depicts various loop-unrollings on an abstract CFG: their CFG morphism corresponds to the mapping that forgets numeric indices on node contents.

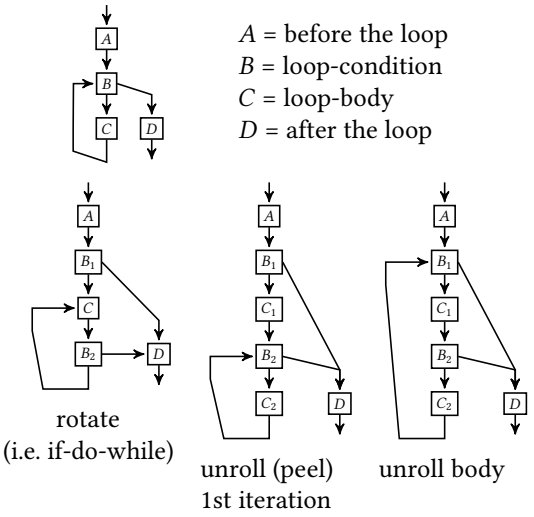

Fig. 1. Three loop-unrollings of a "while-do" loop

Our first improvement is to notice that the reverse transformations, i.e. code factorizations, can also be verified with the same validator (thanks to the reverse simulation). In particular, we formally verify a CFG minimization pass which mimics Moore's algorithm for DFA (deterministic finite automaton) minimization.

We have also generalized the notion of CFG morphism for mappings from BTL—our new IR detailed in §2.2—to RTL. Roughly speaking, in BTL, a CFG node syntactically corresponds to a loop-free fragment of a RTL CFG: a BTL program partitions a RTL program into blocks. A variant of Six's CFG morphism checker [Six et al. 2022, §4.4]—called the *BTL projection checker*—validates the

selection of these blocks in the RTL code, modulo code duplication or insertion of *synthetic nodes*[8]. While these transformations are validated during the RTL to BTL translation, the factorization from our minimization oracle is validated on the way back to RTL.

2.2 Block-by-block Symbolic Simulation Modulo Invariants

Given a RTL CFG, we characterize a *block of this CFG* as a loop-free sub-CFG that has a single entry-point from the remainder of the surrounding CFG, and where only exit-points may emit observational events.[9] Replacing a block by a semantically equivalent one does not change the global CFG semantics. More generally, by partitioning the CFG into blocks, we decompose the "global" simulation proof between two CFGs into a finite conjunction of "local" simulation proofs between block pairs. Then, by means of invariant annotations, the "local" simulations are aggregated at the global level, making sure that changes are consistent between blocks. Validation of these block simulations is fully automated by comparing their *symbolic executions* [Six et al. 2020].

2.2.1 Intra-Block Transformations. Symbolic execution of both blocks amounts to computing the final contents of registers and of the memory as symbolic expressions over their initial contents. If these expressions are structurally equal, then these concrete executions are observationally equivalent, assuming these executions do not trap. Since certain processor operations may trap (e.g. reading from invalid memory locations, division by zero), the symbolic execution of each block also computes a symbolic *precondition* (represented as a conjunction of symbolic clauses) such that the concrete execution traps if and only if the precondition is not satisfied. We require that the precondition of the source code implies the one of the target code. For compilation efficiency, structural equivalence of expressions is reduced to pointer equality through hash-consing (i.e. memoizing expressions such that two structurally equal expressions are uniquely allocated in each compiler run). See Ex. 2.1.

Example 2.1 (Symbolic simulation of basic blocks). Consider two basic blocks B_1 and B_2:
(B_1) $r_1 := r_1 + r_2;\ r_3 := \texttt{load}[m, r_1];\ r_3 := r_1;\ r_1 := r_1 + r_3$
(B_2) $r_3 := r_1 + r_2;\ r_1 := r_3 + r_3$
Both B_1 and B_2 lead to the same parallel assignment:

$$r_1 := (r_1 + r_2) + (r_1 + r_2) \parallel r_3 := r_1 + r_2.$$

B_1 is preconditioned by $\texttt{OK}(\texttt{Read}, r_1 + r_2)$—meaning "the location at address $r_1 + r_2$ is readable"—whereas B_2 has a true precondition. Hence, B_2 simulates B_1, but not the converse.

When applicable, this approach is very powerful, because it is simple, efficient and **does not require hints** from oracles (after the CFG is partitioned into blocks). A typical application is *instruction scheduling*, which reorders instructions to improve parallelism in the processor pipeline.

In their first work on instruction scheduling, [Six et al. 2020] only considered *basic blocks* (sequence of instructions with only one exit-point) of assembly code, and relied on a specific DSL (Domain-Specific Language) for symbolic execution. They later generalized their symbolic simulation validator to *superblocks* (sequence of instructions with intermediate conditional exit-points plus a final exit-point) to validate *superblock scheduling* [Six et al. 2022]. In some further improvements of their system, the source superblock may be transformed into an *extended block* (a sequence of trees without internal joins) which is not a superblock. They overcame the limitation of their symbolic simulation to superblocks with an intricate sequence of passes combining ad-hoc validators [Justus Fasse 2021].

[8] *Synthetic nodes* are "no-op" blocks inserted in the CFG in preprocessing of code motion [Knoop et al. 1992, §3.1].
[9] In CompCert, certain actions, such as calls to external functions and volatile variable accesses, generate *events*. Global correctness amounts to preserving observable event sequences. In contrast, registers etc. are internal affairs of the program.

In contrast, one of our contributions is to define our symbolic simulation validator on a very general syntax for blocks, even if its current applications only apply it to extended blocks. We have introduced an IR called BTL (Block Transfer Language) for representing CFGs of RTL blocks: it replaces the "RTLpath"—a decoration of RTL defining a block semantics—of [Six et al. 2022]. In contrast to RTLpath, BTL represents the control-flow within each block structurally (with an usual abstract syntax tree), instead of an explicit sub-CFG. This makes our symbolic simulation test both simpler and more efficient.[10] Proving the passes between RTL and the CFG of blocks becomes however less trivial: this leads us to generalize the notion of CFG morphism as sketched in §2.1.

2.2.2 Inter-Block Transformations. The simulation test sketched above is only applicable if the transformation preserves the semantics of each block. Indeed, it uses the equality of the source and target states (that is, the registers and memory) at the entry of each block as a simulation invariant. Inter-block transformations cannot be validated using this invariant for two reasons. Firstly, a transformation only needs to consider initial states at the block entry that are reachable by the program, whereas the above simulation test requires proving the preservation for any values of the registers and memory at the block entry. Secondly, the source and target states at the entry of the block dot not need to be equal. For instance, *Dead Code Elimination* (DCE) only maintains the equality of the live registers; some transformations use auxiliary registers, which hold meaningful values only in the target program.

To validate such global transformations, we generalize the *invariants* relating the source and target states at block entries. We progressively explain this idea, with the help of Fig. 2, providing a transformation on C pseudocode.[11] Both the source and the target code are CFGs of two (extended) blocks, labeled by `Entry` (their entry-point) and `Loop`. The target is obtained after a combination of *Constant Propagation* (from `a=7`) and *Strength Reduction*: the multiplication originally within `Loop` is moved to `Entry` and reduced in `Loop` to an addition on a *fresh* register `i_a`. The target also benefits from DCE, since register `a` and its assignment are eliminated.

To account for the differences between the source and target registers at the beginning of the block, we allow our symbolic expressions to mention the initial memory, the initial source registers (denoted with an "S" subscript) and the initial target registers (denoted with a "T" subscript). An invariant is then the conjunction of the equality of the source and target memories, a precondition expressed as a symbolic proposition, and equalities between some registers and symbolic expressions. The symbolic proposition and equalities are produced by our untrusted oracles for each block. Fig. 2 represents the equalities attached to each block of the target code, all preconditions are trivially true in this example.

Source CFG

```
Entry:
  s = 0;
  a = 7;
  goto Loop;
Loop:
  if (i > n)
    return s;
  s += i*a;
  i += 3;
  goto Loop;
```

Target CFG

```
Entry:
  // i_T = i_S ∧ n_T = n_S
  s = 0;
  i_a = i*7;
  goto Loop;
Loop:
  // a_S = 7 ∧ i_T = i_S
  // ∧ i_a_T = i_S * 7
  // ∧ n_T = n_S ∧ s_T = s_S
  if (i > n)
    return s;
  s += i_a;
  i += 3;
  i_a += 21;
  goto Loop;
```

Fig. 2. Simulation modulo Invariants

The simulation modulo invariants is validated by symbolic execution. For each block, we use the invariant at the entry to build an initial symbolic value for each register in the source and target programs. We then perform the symbolic execution of both the source and target blocks, and we compare their respective results through the invariant associated to every block exit. Let us detail this process for the `Loop` block of Fig. 2:

• For the source block, the symbolic execution starts from symbolic state "a := 7" itself deduced

[10] Our representation also slightly generalizes the "*tree-based representation*" of [Tristan and Leroy 2008, §5.1], because in ours, "sequences of trees" are blocks.

[11] Implicitly, all arithmetic computations operate on `unsigned long`.

from the invariant at the block entry. It thus computes the following symbolic assignments at the "goto Loop" exit: "a := 7 || i := i_S + 3 || s := s_S + i_S * 7".
• For the target block, the symbolic execution deduces from the invariant at block entry the following initial symbolic state : "i := i_S || i_a := i_S * 7 || n := n_S || s := s_S". Then, it obtains for this same exit: "i := i_S + 3 || i_a := i_S * 7 + 21 || n := n_S || s := s_S + i_S * 7".
• We then compare those symbolic values according to the invariant at the exit point (here again, the invariant at the Loop entry). After substituting the symbolic expressions of the source and target registers within this invariant, we check the following syntactic equalities: (1) 7 = 7, for equation a_S = 7; (2) i_S + 3 = i_S + 3 for i_T = i_S; (3) i_S * 7 + 21 = (i_S + 3) * 7 for i_a_T = i_S * 7; (4) n_S = n_S for n_T = n_S; (5) s_S + i_S * 7 = s_S + i_S * 7 for s_T = s_S.

In order to prove equality (3), the symbolic execution is extended with rewriting rules normalizing affine expressions (see §6.3). Expressions on both sides are normalized into "21 + 7 · i_S". The rewriting engine within symbolic execution is similar to the one of [Six et al. 2022]: it is simply defined more modularly. Now, each transformation may independently define its own rewriting rules, over a common underlying architecture.

In contrast to [Six et al. 2022], we do not use an ad-hoc validation of register liveness. While our oracles generate only invariants for live registers of the *target* program, the validation of this liveness analysis implicitly results from the preservation of equational invariants between source and target registers. An incorrect liveness analysis will result in an invariant that is invalid after substitution of the target registers, because it will still involve a target register not itself bound to a symbolic expression of source registers.

This implicit liveness validation is strictly more expressive than the explicit liveness validation of [Six et al. 2022]: we express each register "r" considered live by Six-et-al as an equation "r_T = r_S". This enabled us to port their superblock scheduling to BTL (and even improve it) as detailed in §4. Moreover, because they only validate a weak liveness analysis of the source program, their framework is unable to validate any kind of DCE. In contrast, as shown on Fig. 2, ours validates some DCE "for free" (i.e. just from liveness equations as above).

3 BTL AND ITS SYMBOLIC SIMULATION

The abstract syntax of BTL functions—given Fig. 3—is largely inspired by the one of RTL [Leroy 2009b]. Each (internal) function f is associated to a CFG *cfg* and an entry-point pc_{entry} in this CFG. Such a CFG is a map from labels *pc* to instruction blocks *ib*. Syntactically, a block *ib* may be composed of two sub-blocks, either sequentially with Bseq or through a kind of "if-then-else" written Bcond. A block may be reduced to a *basic instruction*: either Bnop that skips, or Bop that assigns the result of "$op(\overrightarrow{r_{arg}})$" to register r_{dst}, or a few others. Last, it may also be a *final instruction* named *fi* that stops the current block execution, such as Bgoto that enters another block of label *pc* and Breturn that returns from the current function. In order to cope with CompCert simulation proofs, we impose that function call—written Bcall—is a final instruction, branching to pc_{ret} after the call. See Fig. 4 for an example in BTL syntax.

Similarly to RTL, the formal semantics of BTL programs and functions is a small-step operational semantics, where each step emits at most one single event. In contrast to RTL where a step runs at most one single instruction, each BTL block is run in one big-step. By construction, such a block big-step necessarily ends just after the RTL-style small-step of a final instruction.[12]

[12]If no final instruction is present, the block cannot step. See our source code online for the Coq definitions.

$$
\begin{array}{ll}
f ::= (sig, \overrightarrow{r_{arg}}, cfg, pc_{entry}, \cdots) & ib ::= \texttt{Bnop} \\
cfg ::= (pc \mapsto ib) & \quad | \, \texttt{Bop}(op, \overrightarrow{r_{arg}}, r_{dst}) \\
fi ::= \texttt{Bgoto}(pc) & \quad \cdots \\
\quad | \, \texttt{Breturn}(\epsilon | r) & \quad | \, \texttt{Bseq}(ib_1, ib_2) \\
\quad | \, \texttt{Bcall}(sig, (r|id), \overrightarrow{r_{arg}}, r_{dst}, pc_{ret}) & \quad | \, \texttt{Bcond}(cond, \overrightarrow{r_{arg}}, ib_{then}, ib_{else}) \\
\quad \cdots & \quad | \, \texttt{BF}(fi)
\end{array}
$$

Fig. 3. Syntax of BTL internal functions

```
if (x >= y) goto L;
x = z << 2;
return x;
```

```
Bseq(Bcond(_>=_, [x;y], BF(Bgoto(L)), Bnop),
     Bseq(Bop(_<<2, [z], x),
          BF(Breturn(x))))
```

Fig. 4. A superblock in C syntax and its BTL representation

$$
\begin{array}{ll}
sv ::= \texttt{Sinput}((\texttt{Src}|\texttt{Trg}), r) & sfv ::= \texttt{Sgoto}(pc) \\
\quad | \, \texttt{Sop}(op, \overrightarrow{sv_{arg}}) & \quad | \, \texttt{Sreturn}(\epsilon | sv) \\
\quad \cdots & \quad | \, \texttt{Scall}(sig, (sv|id), \overrightarrow{sv_{arg}}, r_{dst}, pc_{ret}) \\
sm ::= \texttt{Sinit} & \quad \cdots \\
\quad | \, \texttt{Sstore}(sm, chunk, addr, \overrightarrow{sv_{arg}}, sv_{src}) & ss ::= \texttt{Scond}(cond, \overrightarrow{sv_{arg}}, ss_{then}, ss_{else}) \\
sis ::= (pre, (r \mapsto sv), sm) & \quad | \, \texttt{Sfinal}(sis, sfv)
\end{array}
$$

Fig. 5. Syntax of the symbolic representations

3.1 Symbolic representations

The symbolic execution operate on symbolic representations of the dynamic values involved in the semantics of BTL. Fig. 5 presents the syntax of the main symbolic representations. They express the dynamic values as functions of the context, which fixes the values of the source and target registers at the entry of the block, represented by `Sinput`, and the common memory at the entry, represented by `Sinit`. An internal symbolic state *sis* represents the pair of a register state and a memory, using a symbolic memory *sm* and by associating a symbolic value *sv* to each register. It also contains a symbolic proposition *pre* (such as the `OK` precondition of Ex. 2.1), so that the symbolic state does not represent any concrete state if this proposition is not satisfied. Our paper often write internal symbolic states as parallel assignments such as the one of Ex. 2.1.

3.2 Compact Invariants

A symbolic value represents the value of a register r after a sequence of concrete BTL assignments. As introduced in §2.2.2, our invariants themselves contains symbolic values. However, Ex. 2.1 illustrates on block (B_2) that the symbolic values may contain (exponential) term duplications w.r.t. a sequential representation. Because invariants are syntactically provided by the oracle under validation, we design a *compact* syntax which represents them as sequences of assignments from the source state. A first sequence, the *history invariant* $\mathcal{H}$, provides symbolic expressions for the source registers. A second sequence, the *gluing invariant* $\mathcal{G}$, provides symbolic expressions for the target registers.[13] The preconditions of the operations of the sequences are used to define the precondition of the invariant.

$$
\begin{array}{lll}
ci & ::= & (\overrightarrow{r_{dst} := iv}, \overrightarrow{r_{output}}) \\
iv & ::= & \texttt{input?}\, r \\
 & | & \texttt{Iop}(op, \overrightarrow{\texttt{input?}\, r}) \\
 & | & \texttt{Iload}(chunk, addr, \overrightarrow{\texttt{input?}\, r})
\end{array}
$$

Fig. 6. Syntax of Compact Invariants

[13] The term "gluing invariant" is inspired by Abrial [1996]. Our "gluing invariants" have also similarities with "simulation invariants" of Rinard and Marino [1999], and our "history invariants" with their "standard invariants".

Fig. 6 provides our abstract syntax for compact invariants. A compact invariant *ci* is a sequence of assignments of invariant values *iv* to registers r_{dst} together with a set of output registers. An invariant value *iv* may read a register *r*, optionally with a prefix "`input`" indicating that the input value is considered instead of the current one (within the assignment sequence). It can also represent an operation involving current or input registers, or a load from the *current* memory. Initially, current and input registers are the registers of the source. In the rest of this document, we will note sequential invariants as "`([a1:=v1;a2:=v2;...;aK:=vK], {o1,o2,...,oN})`", where the "`ai:=vi`" are the assignments and where the set of output registers is between braces.

Example 3.1. Interpreted as gluing invariants, the two following compact invariants
"`([a:=y[5]; z:=a+input x; x:=a+input z], {x,z,t})`"
and "`([z:=y[5]; x:=z+input z; z:=z+input x], {x,z,t})`"
both represent the equations "$x_T = y_S[5] + z_S \wedge z_T = y_S[5] + x_S \wedge t_T = t_S$" with precondition "`OK(Read, y[5])`".

Given a symbolic internal state representing the source state, the simulation checker performs a hash-consed symbolic execution of a compact invariant to obtain symbolic representations of the right-hand side expressions of the invariant and its precondition. At the entry of the block, we can complete the result of this execution using trivial equalities $r_S = r_S$ or $r_T = r_T$ to obtain a symbolic value for each register.

The symbolic validation exploits two properties of those invariants:
• Given a pair of source and target symbolic internal states, we can check that an invariant holds by executing $\mathcal{H}$ and $\mathcal{G}$ from the source state and comparing the symbolic values obtained for the right-hand sides with the symbolic value associated to the registers by the symbolic states. We also check that the precondition of the invariant is implied by the precondition of the symbolic states.
• Given some invariants $\mathcal{H}$ and $\mathcal{G}$, we can build symbolic states representing any pair of concrete states related by those invariants and reflecting the properties they enforce. We start with an empty symbolic state representing trivially the source state (using $r \mapsto r_S$ for the registers). We execute $\mathcal{H}$ to obtain a new representation of the source state that enforce the equalities and the precondition of $\mathcal{H}$. Then we execute $\mathcal{G}$ from this new symbolic state to obtain a state representing the target state. Finally, we refine the source symbolic state by adding the precondition of $\mathcal{G}$.

3.3 Symbolic Execution with Trace-Partitioning By Continuation-Passing

Our theory of symbolic execution improves upon Six et al. [2022]'s with mainly two features. First, as explained in §2.2.2, the oracle provides invariants that allow the validation of inter-block transformations. Second, BTL symbolic execution supports arbitrary nested sequences of "if-then-else" instead of superblocks only. This required both a more general representation of the *symbolic states* generated by symbolic execution and a kind of trace-partitioning within it.

A symbolic state represents all possible block steps of given BTL block. Fig. 5 defines a *symbolic state ss* as a binary decision tree, where each branch represents one possible execution path of the block. A "`Scond`" node represents the choice between two branches. Leaves are "`Sfinal`(*sis*, *sfv*)" where *sis*—*a symbolic internal state*—represents the state of the program when it reaches the exit of the block, and *sfv*—*a symbolic final-value*—represents the final instruction of the exit (where registers have been substituted by their final symbolic values).

Fig. 7 sketches the Coq code of the symbolic execution. Here, `sstate` and `sistate` are respectively the types of symbolic state *ss* and symbolic internal state *sis*. Function `set_sreg` updates the current *sis* by assigning a new symbolic value to register `dst`. The symbolic execution of the register list `args` into a symbolic value list is delegated to function `lmap_sv`.

```
Fixpoint sexec_rec ib sis (k: sistate → sstate): sstate :=
  match ib with
  (** basic instructions *)
  | Bnop ⇒ k sis
  | Bop op args dst ⇒ k (set_sreg dst (Sop op (lmap_sv sis args)) sis)
  | ...
  (** others *)
  | Bseq ib1 ib2 ⇒ sexec_rec ib1 sis (fun sis2 ⇒ sexec_rec ib2 sis2 k)
  | Bcond cond args ifso ifnot ⇒
      Scond cond (lmap_sv sis args) (sexec_rec ifso sis k) (sexec_rec ifnot sis k)
  | BF fi ⇒ Sfinal sis (sexec_final_sfv fi sis)
  end
```

Fig. 7. Symbolic Execution in CPS

The symbolic execution—named `sexec_rec`—computes over block `ib` recursively. The initial (internal) state is `sis`. The trace partitioning is here realized in Continuation Passing Style (CPS). Continuation `k` represents how symbolic execution should "normally" continue on updates of internal state. It is initialized as "`error_sstate`", which returns a state with a false precondition, reflecting the fact that each BTL block step must reach a final instruction.

3.4 Symbolic Simulation Modulo Invariants

The blue subdiagram in Fig 8 represents the computations performed by our symbolic simulation validator. The surrounding diagram (in black color) represents its correctness proof wrt concrete executions. Hence, we deduce the lock-step forward simulation[14] of the source BTL code by the target one from their simulation block-by-block, and check the latter by comparing the symbolic states produced by the symbolic execution of each block. Our simulation relation $\mathcal{R}$ relates pairs of states at the same program location, using the invariants provided by the oracle for this location. In the formal proof, we also need to relate the call stacks of the two programs (see Appendix A.1).

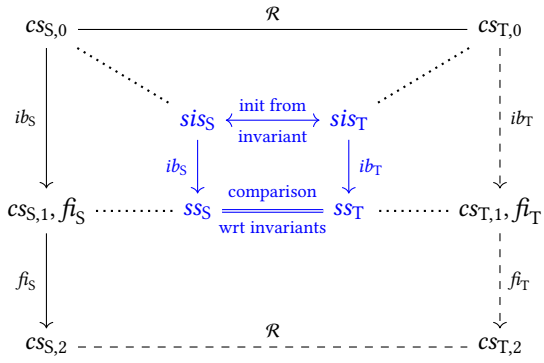

Fig. 8. Simulation Step using Symbolic Simulation

Since we prove a forward simulation, we consider a pair of source and target states $cs_{S,0}$ and $cs_{T,0}$ at the entry of the block and related by the invariant defined by the oracle for the block. We assume that the source program take a step to $cs_{S,2}$, and we have to prove that the target program can take a corresponding step to some state $cs_{T,2}$ related to $cs_{S,2}$.

Because $cs_{S,0}$ and $cs_{T,0}$ are related by the invariant of the block, we can build from it two symbolic internal states sis_S and sis_T representing the source and target initial states. The validator perform the symbolic execution of the blocks to obtain two symbolic states ss_S and ss_T representing the outcomes of the executions of the blocks. It then performs a comparison of those two decision trees. It checks that they have the same structure with the same conditions. For each corresponding leaves, it checks that (1) the symbolic final values and symbolic memories are identical; (2) the symbolic precondition of the source outcome implies the one of the target; (3) the simulation relation is

[14]Thanks to deterministic target languages, the correctness of CompCert's backend passes reduces to a *forward simulation* [Leroy 2009b].

satisfied after an execution of the common final instruction. For this last condition, it checks the invariants associated to all the blocks the final instruction can jump to. In the case of a final value "$\texttt{Scall}(sig, (sv|id), \overrightarrow{sv_{arg}}, o, pc_{ret})$", it also checks that those invariants do not constrain the memory and the output register o, possibly modified by the call: it only allows "liveness" equation $\mathsf{o}_\mathrm{T} = \mathsf{o}_\mathrm{S}$.

The step of the source block can be decomposed into an execution up to a final instruction fi_S followed by an execution of this final instruction. By the correctness of `sexec`, ss_S represents the state of the source program before the execution of fi_S. In particular, its precondition is satisfied. The comparison ensures that the precondition of ss_T is also satisfied, hence by the correctness of `sexec`, the target program can take a step-up to a final instruction fi_T. Since fi_S and fi_T are represented by the same symbolic final value, we can simulate the execution of fi_S by an execution of fi_T to a state $cs_{\mathrm{T},2}$. Because of the comparison performed between the symbolic states, the simulation relation holds between $cs_{\mathrm{S},2}$ and $cs_{\mathrm{T},2}$.

Besides this block-to-block simulation, we also need to ensure that invariants at the CFG entry do not constrain any register, except by liveness equations. Indeed, in our intra-procedural verification, the value of parameters must be considered as unknown.

3.5 Refinement with Hash-Consing and Normalized Rewriting

In our Coq development, the code in Fig. 7 specifies the symbolic state that the symbolic execution should produce modulo semantic equivalence. This specification is refined with normalized rewriting and hash-consing in the impure monad of Boulmé [2021]. Rewriting reduces comparisons (of symbolic values) modulo a set of equations to structural equalities [Kirchner and Kirchner 2014]. Hash-consing reduces these structural equalities to pointer equalities. Hence, §3.4 gives itself a specification of our validator which compares symbolic values using pointer equality of OCaml. Here, we adapt the technique of Six et al. [2022] with much more complex rewriting rules. See §6.3.

BTL is annotatable with results of verified static analyses. See Appendix A.4. These annotations are taken into account in the semantics and used to justify some rewritings. In particular, we rely on a non-alias analysis to overcome our symbolic representation of the memory as a single variable (see Appendix B), and an interval analysis enables an integer promotion pass (see §6.4).

3.6 Limitations of our Simulation Checker

Our simulation checker has two kinds of limitations: performance ones (impacting CompCert running times) and expressivity ones (restricting the class of simulations that can be validated).

Performance. While in theory, any piece of code without loops may be represented as a BTL block, in practice our symbolic execution performs a naive trace partitioning: it is thus exponential over the number of *internal joins* of the input block (i.e. such that two branches of a given block join at a point which is not a block entry). Because we currently only apply our checker to extended blocks, which, by definition, do not have such joins, this is not an issue. Furthermore, for blocks with a bounded number of internal joins, and without rewriting rules, our symbolic execution is linear in the size of invariants and blocks (see §8.1). In the general case, its cost depends on the normalization system. For example, for the normalization of affine forms (for SR), it is expected to be quadratic in the worst case. Lastly, the comparison of symbolic states costs $O((l + t) \times e)$ where l (resp., t) is the maximal number, by execution path, of liveout registers (resp., trapping instructions) and e is the number of execution paths (coinciding with the number of exits for blocks without internal joins). Block selection is a way to finely control e, and thus checker performance.

Expressivity. The relative simplicity and efficiency of our checker comes at a price: its expressive power is limited. (1) Our invariants only support equations of the form "$r = sv$" but not the more general "$sv_1 = sv_2$": this limitation avoids the need of costly saturation techniques. (2) Our

simulation checker performs no reasoning on *conditions*. It simply checks that the two symbolic states under comparison have the same binary decision tree structure, with syntactically equals conditions on nodes. Future works include supporting conditions within invariants with a more expressive comparison of decision trees and preconditioned rewriting rules. (3) Our invariants implicitly express that their trapping expressions are actually safe in the execution context. This forbids the target to *anticipate* traps with respect to the source. Avoiding this restriction would require *prophecies* [Abadi and Lamport 1991] ensuring that these traps will *eventually be observed on the source before any subsequent observable event*. Besides generalizing the semantics of our invariants, this would need introducing a notion of "*decreasing variant*" forbidding never-realized prophecies. Currently, we partly overcome this restriction with the help of CFG morphisms. See §5.1. (4) The invariants presented here require the source and target memories to be equal at BTL block boundaries. We can however generalize the invariants while keeping the formalism of §3.4. This enables for example validating the loop invariant code motion of a memory update after a loop.

4 APPLICATION TO SUPERBLOCK SCHEDULING

In-order processor cores execute assembly instructions in their syntactic order. If one instruction computes a register used by the next one, then the core stalls until the value computed becomes available, which may take several clock cycles. Optimizing compilers thus reorder instructions to minimize stalling: e.g. prioritizing instructions with high latencies and impacting many other instructions. For example, the superblock scheduler of Six et al. [2022, §6] attempts to minimize the running times of the execution path covering the whole superblock, even if it may increase running times of early exiting paths, but without increasing register usage beyond the point where the subsequent register allocation would be forced to spill registers to memory. This scheduler is based on a solver that reorders the superblock without code duplication.

We ported the Six et al.'s superblock scheduler from RTLpath to BTL. We now explain, on a variant of their example [Six et al. 2022, Fig. 10], how the design of BTL makes the implementation much easier. This example applies a *weak* form of *software pipelining*, that they call *if-lifting*, and targets ARM Cortex-A53 (AArch64), a dual-issue in-order processor with two ALUs. In Fig. 9, we optimize the top source C code by parallelizing computations between two successive iterations of the loop body. The left-hand side represents the BTL code of the loop body, after a loop-rotate and unroll-body (see Fig. 1). Hence, the "Loop" superblock (which is a loop body containing two iterations of the original loop) is scheduled as the BTL block represented on the right-hand side: the two floating-point computations of the first iteration (in violet color) have been moved below the intermediate exit in order to be interleaved with those of the second iteration. The

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

```
Loop:
  x7 = float64[x2+x3<<3]
  x6 = x7 *f x7
  x4 = x4 +f x6
  x3 = x3 +l 1
  if (x3 >=lu x1)
    goto Exit
  // start second iteration
  x7 = float64[x2+x3<<3]
  x6 = x7 *f x7
  x4 = x4 +f x6
  x3 = x3 +l 1
  if (x3 >=lu x1)
    goto Exit
  goto Loop

Exit:
  return x4
```

```
Loop: // live: x1, x2, x3, x4
  x11 = float64[x2+x3<<3]
  x8  = x3 +l 1
  if (x8 >=lu x1) {
    x10= x11 *f x11
    x9 = x4 +f x10
    x4 = x9
    goto Exit // live: x4
  }
  x3 = x8 +l 1
  x7 = float64[x2+x8<<3]
  x10 = x11 *f x11
  x6 = x7 *f x7
  x9 = x4 +f x10
  x4 = x9 +f x6
  if (x3 >=lu x1)
    goto Exit // live: x4
  goto Loop
```

Fig. 9. Interleaving of unrolled loop-bodies on AArch64.

scheduler predicts[15] that the target loop body will run in at most 22 cycles instead of 32 cycles for the original one (e.g. more than 30% gain of running time). However, preserving the semantics requires *register renamings* with fresh registers (in red color) and insertion of *compensation code* (in blue color) at the intermediate exit. Because of this insertion, the target block is an extended block.

This whole transformation is directly validated by our simulation checker, with gluing invariants reduced to a conjunction of liveness equations "$\mathtt{r}_T = \mathtt{r}_S$", over all registers `r` of the live sets given on the right-hand side (in yellow color). On the contrary, because RTLpath can only represent superblocks, validating this transformation with Six's design required an "*intricate combination of passes*" [Six et al. 2022, Footnote 10].

Our implementation of the oracle is also quite simple. First, register renamings are performed backward in a preliminary pass (we avoid forward renamings of [Six et al. 2022, §5.3] because they tend to pollute the superblock under scheduling with compensation renamings). Then, we invoke the scheduling solver on a *fake superblock*, with an empty live set on intermediate side exits. Last, we insert the necessary compensation code in the side exits, following Justus Fasse [2021]'s heuristic. We then compare the makespan (total estimated time) of this scheduling to the standard one (computed on the *original superblock* with correct liveness and without compensation code). If the ratio of the size of the compensation code over the makespan gain is greater than a given threshold, we keep the standard one instead. Hence, compiler users may control this scheduling heuristic by tuning the threshold on the command line.

We have also refined this formally verified scheduler with reordering of non-interferring load or store w.r.t store for some non-alias analyses (see Appendix B). Yet, we measured a gain only on a few benchmarks.

5 APPLICATION TO LAZY CODE MOTION

Code motion consists in anticipating some instructions in order to remove redundant computations. For example, by data-flow analysis, we may detect expressions remaining constant within a loop and anticipate their computation before the loop: this is *LICM* (*Loop Invariant Code Motion*). However, if done carelessly, this transformation may anticipate a loop-invariant expression that traps (e.g. a memory load from a potentially invalid pointer, or a division operation on some architectures), whereas this computation is unreachable in the original loop. Safe elimination of such computations—that are redundant on some but not all program paths—is called *PRE* (*Partial Redundancy Elimination*). According to Bodík et al. [1998], "*to achieve a complete PRE, control flow restructuring must be applied. However, the resulting code duplication may cause code size explosion.*" They propose to guide these CFG restructuring with path-profiling and data-flow frequency analysis.

Lazy Code Motion [Knoop et al. 1995, 1992] performs safe and optimal PRE without CFG unrolling, while limiting the register pressure induced by code motion. Instructions are safely anticipated but not earlier than the minimum necessary to reach *computational optimality* (i.e. with a minimal average running time for PRE without CFG unrolling). In other words, among computationally optimal code motions, LCM selects those that minimize *register pressure*. Below, we present our formally verified PRE within the CompCert compiler: §5.1 combines LCM with simple CFG unrollings on the example in Fig. 10; §5.2 details the validation of our LCM oracle on this example.

5.1 Performing LICM by PRE on an Example

Figure 11 presents an extract of the RISC-V code produced by CompCert with CSE3 of [Monniaux and Six 2021, 2022] activated for the source C code in Fig. 10. The computation of `a[0]` has been

[15]This estimation occurs at an abstract level and thus cannot be precise. First, the subsequent register allocation could introduce unexpected spills. Second, this estimation assumes that there is no cache miss. Third, the pipeline model is inexact.

factorized in f3 register over the whole program. But, computations (in violet color) of a[1] in f0 and a[2] in f2, and loading of floating-point 7 in f1 is performed at each iteration of the loop of label .L102.

In contrast, in Fig. 12 (the four first lines are omitted because identical), after unrolling the first iteration (see Fig. 1) our LCM moves all these computations before the loop, starting now at label .L106. Remark that if the condition of the loop is initially false, a[2] is not computed by the original loop, but may trap if the address is invalid. Thus, simply anticipating the computation of a[2] before the loop is incorrect: unrolling the first iteration gives a simple workaround.

However, it may not suffice. For example, in Fig. 10, if the test on line 3 was omitted, then simply unrolling the first iteration would not suffice to allow a[0] to be moved before the loop. Indeed, if r < a[2] at the first iteration, then a[0] is not computed and may still trap afterward. Actually, following [Bodík et al. 1998], we may find an unrolling (validated by Six's CFG morphism checker) that enables it. But this would cost even more code duplications than those of Fig. 12.

Let us now explain why LCM is more powerful than CSE. Applying CSE3 after unrolling produces almost the same code as the one of Fig. 12 except that the load of floating-point 7 is not factorized.[16] This is due to the fact that some execution path of the first iteration does not load floating-point 7 into f1. Indeed, CSE3 can only eliminate computations that are available on all incoming paths. Thus, CSE3 only performs some *FRE* (*Full Redundancy Elimination*): it misses FRE if the same value is available on different incoming paths, but in different registers. In contrast, LCM is able to perform any FRE and even non-trapping PRE *without unrolling*. On Fig. 10 example, the load of floating-point 7 is anticipated even without any loop unrolling. In the original version of [Knoop et al. 1995], LCM also safely moves a[1] out of the loop without any loop transformation: this is a FRE, since a[1] is present in the condition of the loop, which is at least run once. Nevertheless, due to the current design of our simulation checker (see §3.6), anticipating trapping code w.r.t. the original program does not pass validation: our LCM can only eliminate a[1] within the loop, after at least a loop-rotation (see Fig. 1). This is not an issue on this very simple example: after applying our CFG minimization, we still finally achieve the FRE of a[1] without any code duplication.

```
double approx(double *a) {                    1
  double r = 2;                               2
  if (a[0] < 2) return 2;                     3
  while (r < a[1])                            4
    if (r >= a[2]) r -= a[0];                 5
    else r *= 7;                              6
  return r;                                   7
}                                             8
```

Fig. 10. Four Candidates for LICM

```
  fld     f3,0(x10)
  fld     f10,.L100,x31
  flt.d   x31,f3,f10
  bne     x31,x0,.L101
.L102: # Loop Entry
  fld     f0,8(x10)
  flt.d   x31,f10,f0
  beq     x31,x0,.L101
  fld     f2,16(x10)
  fle.d   x31,f2,f10
  bne     x31,x0,.L103
  fld     f1,.L104,x31
  fmul.d  f10,f10,f1
  j       .L102
.L103:
  fsub.d  f10,f10,f3
  j       .L102
.L104: ...# 7.0 in hexa
.L100: ...# 2.0 in hexa
```

Fig. 11. CSE3 alone

```
  ... # Same prolog
  fld     f0,8(x10)
  flt.d   x31,f10,f0
  beq     x31,x0,.L101
  fld     f2,16(x10)
  fle.d   x31,f2,f10
  bne     x31,x0,.L103
  fld     f10,.L105,x31
  j       .L102
.L103:
  fsub.d f10,f10,f3
.L102:
  fld     f1,.L104,x31
.L106: # Loop Entry
  flt.d   x31,f10,f0
  beq     x31,x0,.L101
  fle.d   x31,f2,f10
  bne     x31,x0,.L107
  fmul.d f10,f10,f1
  j       .L106
.L107:
  fsub.d f10,f10,f3
  j       .L106
.L105: ...# 14.0 in hexa
.L104: ...# 7.0 in hexa
```

Fig. 12. Unroll+LCM

[16]The original CSE of CompCert does not even eliminate the redundant "a[0]". This contrast with gcc -O1 (version 9.4.0) which performs a PRE with slightly less code duplications than ours on this example. However, the original CSE of CompCert factorizes the load of floating-point 2 into register "f10". Such a "trivial" CSE is required by the LCM algorithm: block-based LCM cannot exempt from, at least, an intra-block CSE. See [Knoop et al. 1995].

5.2 Formally Validated LCM

Code motion moves computations from one block to another, and is thus global over the CFG. Figure 2 illustrates how invariants propagate information between blocks. Our simulation modulo invariants also validates our LCM, as now detailed on the previous example.

Our LCM oracle optimizes a BTL CFG of basic blocks. The CFG of the example is represented in Fig. 13, where each node is identified by a unique number (in yellow), and where synthetic nodes (explained in Footnote 8) are denoted with a violet border. The four candidates detected by the oracle have been inserted at their optimal points, by assigning them to a fresh variable (in red).

For instance, the load of floating-point 7 illustrates this "lazy" behavior of LCM as it is inserted in two different blocks (14 and 16) to minimize the live range. The calculation is therefore duplicated on two branches, and both chosen blocks are the last possible ones before the loop. However, this code duplication does not appear in the final assembly code of Fig. 12, because it is factorized by our subsequent CFG minimization pass. In Fig. 13, it also appears that some fresh variables, such as `x10` and `x11`, are duplicated through a compensation `move` (in blue). This behavior will be explained §6. Note that these pseudo-register duplications do not increase the actual live range since they will be removed by the subsequent register allocator.

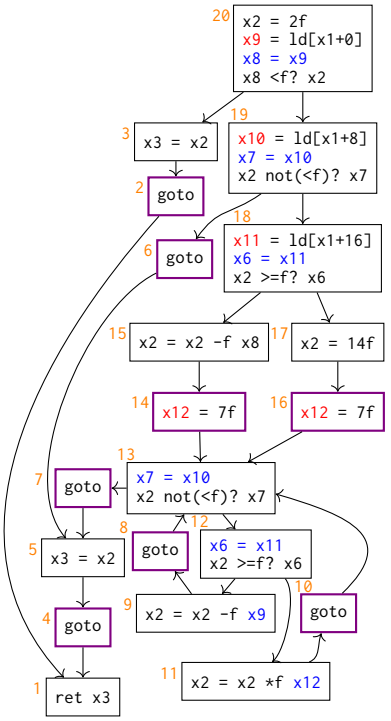

Fig. 13. Full BTL CFG of Fig. 12

Now let us detail how the invariants are generated by our LCM oracle before sending the whole result to the verifier. Our LCM only requires gluing invariants: thus, history invariants remain trivially empty here. Invariants are generated in the sequential representation of §3.2. As explained at the end of §3.4, the invariant of the entry block (here block 20) is always reduced to a liveness set. Besides liveness sets, invariants are updated for each candidate just after they appear in code: at the entry of block 19, we have "`([x9:=ld[x1+0]], {x1,x2,x8,x9})`" to remember the load, and because these four variables are live. The second (resp. third) load is added to the gluing invariant at block 18 (resp. blocks 15 and 17). Thus, for all blocks with a label in 14 . . . 17, the invariant contains the same sequence of assignments (but the sets of live variables are different): "`x9:=ld[x1+0]; x10:=ld[x1+8]; x11:=ld[x1+16]`". From block 13 and up to block 8 (included), we append to this list the assignment "`x12:=7f`". Finally, blocks 1 to 7 only contain pure-liveness invariants, as the verification need not remember the values of candidates anymore.

Example 5.1. Validating symbolic simulation for block 14 (with the validator described in §3.4).

$\mathcal{G}(14) =$ `([x9:=ld[x1+0]; x10:=ld[x1+8]; x11:=ld[x1+16]], {x2,x9,x10,x11})`

This leads to the following initial states:

$$sis_S = (pre, \epsilon)$$

$$sis_T = (pre, \texttt{x2} := \texttt{x2}_S \| \texttt{x9} := \texttt{ld}_S[\texttt{x1}_S + \texttt{0}] \| \texttt{x10} := \texttt{ld}_S[\texttt{x1}_S + \texttt{8}] \| \texttt{x11} := \texttt{ld}_S[\texttt{x1}_S + \texttt{16}])$$

$$\text{where } pre = \texttt{OK}(\texttt{Read}, \texttt{ld}_S[\texttt{x1}_S + \texttt{0}]) \wedge \texttt{OK}(\texttt{Read}, \texttt{ld}_S[\texttt{x1}_S + \texttt{8}]) \wedge \texttt{OK}(\texttt{Read}, \texttt{ld}_S[\texttt{x1}_S + \texttt{16}])$$

After the symbolic execution of the blocks:

$$ib_s = \texttt{BF}(\texttt{Bgoto}(13)) \qquad \text{and} \qquad ib_t = \texttt{Bseq}(\texttt{Bop}(\texttt{7f}, [], \texttt{x12}), \texttt{BF}(\texttt{Bgoto}(13)))$$

we obtain:

$$ss_S = \texttt{Sfinal}(sis_s, \texttt{Sgoto(13)})$$
$$ss_T = \texttt{Sfinal}((pre, \texttt{x2} \coloneqq \texttt{x2}_S \| \texttt{x9} \coloneqq \texttt{ld}_S[\texttt{x1}_S + \texttt{0}] \| \ldots \| \texttt{x12} \coloneqq \texttt{7f}), \texttt{Sgoto(13)})$$

The two decision trees have the same structure and their unique leaves match. In particular, the gluing invariant of the successor block 13 is satisfied:

$\mathcal{G}(13) =$ `([x9:=ld[x1+0]; x10:=ld[x1+8]; x11:=ld[x1+16]; x12:=7f], {x2,x9,x10,x11,x12})`

For instance, the right-hand side expression for `x9` evaluates to $\texttt{ld}_S[\texttt{x1}_S + \texttt{0}]$ in ss_S, which matches the value of `x9` in ss_T.

We further detail in §9 the comparison of our formally verified LCM w.r.t. [Monniaux and Six 2021, 2022; Tristan and Leroy 2009].

6 APPLICATION TO LAZY STRENGTH REDUCTION

Strength-reduction (SR) consists in replacing (sequences of) computations by semantically equivalent—but more efficient—ones. Simple forms of strength reduction, for instance replacing a multiplication by a power of two with a left shift, are already implemented in CompCert. Here we tackle a much more advanced approach: *Lazy Strength Reduction* [Knoop et al. 1993], a generalization of LCM that reduces computations *while* moving them. For instance, an expensive multiplication $C \times i$ within a loop with loop index i may be replaced by accumulation ($C \times (i + 1) = C \times i + C$).

6.1 Motivations for RISC-V

Some architectures provide instructions or addressing modes for commonly found patterns, such as array addressing. In truly reduced instructions sets, such as RISC-V, these patterns instead result in a multi-cycle sequence of instructions, amenable to SR. Figure 14 shows the single AArch64 load generated for array access "`x = a[i]`" (with an addressing mode that shifts an index by three bits and adds it to a base address) compared to the succession of RISC-V instructions that shift, add, then load. The lack of SR for such sequences may explain why CompCert performs more poorly compared to GCC on RISC-V than on other architectures.

```
ldr   x0,[x0,w1,sxtw#3]
```

```
slli  x6,x11,3
add   x6,x10,x6
ld    x6,0(x6)
```

Fig. 14. AArch64 (top) vs. RISC-V (bottom) addressing

6.2 SR on a realistic example

We specialized and extended our LCM algorithm in order to strength-reduce multiplicative and additive computations on the RISC-V (64 bits) backend. Our contribution generalizes the LSR algorithm of [Knoop et al. 1993] to operate over basic blocks, and is also validated via our general-purpose framework. Note that for LSR, both types of invariants are exploited, because the simulation have to remember the value of constants when verifying the correctness of newly inserted instructions. Since the feasibility of SR for a given candidate is conditioned by the existing dependencies on its variables, we apply a pass of *move forwarding* in the first place. The latter removes read-after-write dependencies coming from `move` instructions, that might be obstacles to the LSR.

The C code of Figure 15 multiplies a slice `[i, n)` of a vector `x` by a scalar `l`, and contains two candidates to be reduced. Indeed, in addition to the product itself, the addressing computation to access `x[i]` can be rewritten as well. The original and optimized BTL codes are set side-by-side in Figure 16. The yellow comment on the left gives the correspondence between registers and variables from the source C program. The multiplication "`l * i`" corresponds to "`x1 *l x4`" in both codes, and, on the left code, the sequence "`x8 = x1 <<l 3; x6 = x3 +l x8`" calculates into `x6` the address of `x[i]`. Synthetic nodes are still denoted in violet, fresh variables in red, and compensation code in blue. We omitted blocks 14 and 1 in the optimized BTL code, as they are identical.

```
void init_slice(long *x, long n, long i) {
  long l = 10;
  for(; i < n; i++) x[i] = l * i;
}
```

Fig. 15. Two Candidates for LSR

```
Variables:
x1 = i; x2 = n;
x3 = x; x4 = l

14: x4 = 10L
    goto 12

12: goto 11

11: if (x1 >=ls x2)
       goto 1
    goto 9

9:  x8 = x1 <<l 3
    x6 = x3 +l x8
    x7 = x1 *l x4
    int64[x6 + 0] = x7
    x1 = x1 +l 1
    goto 3

3:  goto 11

1:  return
```

```
ℋ: ([x4:=10], {x4})
𝒢: ([], {x1,x2,x3,x4})
12: x9 = x1 <<l 3
    x10 = x3 +l x9
    x11 = x1 *l x4
    goto 11

ℋ: ([x4:=10], {x4})
𝒢: ([x9:=x1<<3; x10:=x3+x9;
     x11:=x1*x4],
    {x1,x2,x3,x10,x11})
11: if (x1 >=ls x2) goto 1
    goto 9

ℋ&𝒢: see block 11
9:  int64[x10 + 0] = x11
    x10 = x10 +l 8
    x11 = x11 +l 10
    x1 = x1 +l 1
    goto 3

ℋ&𝒢: see block 11
3:  goto 11
```

Fig. 16. Original (left) and Reduced (right) BTL

Our oracle significantly improves the original LSR algorithm of Knoop et al. [1993] by (1) *sorting* (and processing) candidates according to their topological order of appearance[17]; (2) *substituting* freshly introduced variables; (3) *inferring* the history ($\mathcal{H}$) and gluing ($\mathcal{G}$) invariants required for validation (in the Figure, those for block 9 and 3 have been omitted, because they are identical to those of block 11). Applied to our example after moving the left shift instruction from block 9 to block 12, the old destination (here `x8`) is replaced in the instructions following the original position of the candidate (in block 9) with the newly allocated variable (`x9`)[18]. This enables then to also strength-reduce the addition originally assigning to `x6` in block 9 (it is moved to block 12 as the assignment to `x10`). Note that the substitution of `x8` by `x9` is fundamental here: if we had simply inserted a move directly in place of the shift instruction, the data-flow analysis over the addition would have been blocked because of the write to one of the arguments within the block. The multiplication originally in block 9 is moved out as in the standard way of Knoop et al. [1993].

Of course, it is necessary to update the registers of all these anticipated computations as the `x1` argument is incremented inside the loop. To handle this, we keep a map from registers to affine forms which is updated during the candidates' detection phase. For example, the left shift operation associates `x9` (formerly `x8`) to affine form "$8 \cdot$ `x1`" (knowing that $x \ll n = 2^n \cdot x$). When the subsequent addition is selected as a candidate, a new affine form for `x10` (formerly `x6`) is created, and by substitution of existing affine forms, its value is "`x3` $+ 8 \cdot$ `x1`". The normalization of affine forms in the oracle follows the theory given §6.3.

Finally, every affine form "injured" within the loop needs to be incremented (resp. decremented) by the product of the constant factor—within the form—of the concerned variable by its increment (resp. decrement) step in the "injuring" operation (e.g. the loop induction variable). In this specific example, incrementing `x1` by one corresponds to increment the affine forms of `x9` and `x10` by 8. Thus, the oracle inserts assignments "`x9 = x9 +l 8`" and "`x10 = x10 +l 8`" in the loop, before the injuring operation. The exact same method applies to the affine form "$10 \cdot$ `x1`" associated to `x11` (formerly `x7`).

For the sake of simplicity, this updating phase of our LSR oracle does not track whether the `x9` variable is read afterward (either in the current block or in a successor). At the end, after having updated the liveness information to complete the invariant mapping with a set of "output" variables,

[17] Sorting is made possible thanks to a prior post-order CFG renumbering, which also accelerates fixed point calculations.
[18] When the original variable is live in successors or read in the final instruction, the oracle inserts a `move` before the last read (or the end of the block) to compensate.

$$c \cdot (v_1 + v_2) = (c \cdot v_1) + (c \cdot v_2) \qquad c_1 \cdot (c_2 \cdot v) = (c_1 c_2) \cdot v \qquad c_1 \cdot c_2 = c_1 c_2$$

$$v_1 + v_2 = v_2 + v_1 \qquad (v_1 + v_2) + v_3 = v_1 + (v_2 + v_3) \qquad 0 + (v_1 + v_2) = v_1 + v_2 \qquad 0 + (c \cdot v) = c \cdot v$$

$$(c_1 \cdot v) + (c_2 \cdot v) = (c_1 + c_2) \cdot v \qquad (c \cdot v) + v = (c + 1) \cdot v \qquad v + v = 2 \cdot v$$

Fig. 17. Affine Arithmetic of CompCert 64-bits Integer Operators on Values

$$0 + v = v \qquad 0 \cdot v = 0 \qquad 1 \cdot v = v \qquad (v + v) - v = v + (v - v)$$

Fig. 18. Examples of Invalid Equalities for CompCert 64-bits Integer Operators

$$\pi ::= v \mid c \cdot v \qquad \phi ::= v \mid c + \Sigma_{i=1}^{n} \pi_i \quad \text{where } n \geq 0 \text{ and } (\pi_i)_{i \geq 1} \text{ is strictly increasing}$$

Fig. 19. Representation of our Affine Forms

the LSR oracle eliminates the dead assignments previously generated through an untrusted DCE pass, local to the block. In this example, the update "`x9 = x9 +1 8`" is safely removed.

6.3 Formally-Verified Normalized Rewriting on Affine Forms

Equational Theory. Our SR is validated in the variant of affine arithmetic given in Fig. 17, where c represents a 64-bits integer constant and v is a CompCert value. Actually, we consider this theory extended with specific operators such as $v \ll c = 2^c \cdot v$. However, note that some usual equations—such as those given in Fig. 18—do not hold. For example, if one of their argument is not a long integer or not a pointer—e.g. a float—64-bits integer operations return the absorbing `Vundef` value.[19] Moreover, operation "+" also performs pointer arithmetic in the abstract CompCert model of pointers (and our LSR leverages this opportunity). In this model, on a 64-bits architecture, if v is a pointer, then $v + v' \neq$ `Vundef` if and only if v' is a 64-bits integer (in this case, v' is seen as a relative offset w.r.t v). And $c \cdot v =$ `Vundef` if v is a pointer. This explains why we never identify v and $1 \cdot v$. But, if v is a pointer, then $v + v =$ `Vundef` and we still have $v + v = 2 \cdot v$.[20]

Normal Forms. Since our symbolic values are evaluated to CompCert values (for a given block execution context), Fig. 17 also induces semantic equalities about symbolic values. As noticed in §3.5, normalized rewriting (when applicable) reduces such semantic equalities to structural equalities. Our representation of normal (i.e. canonical) forms is given in Fig. 19, where v represents now a *variable* (representing itself a symbolic value). Due to the commutativity of "+", normal forms depend on a total order over variables. Let us assume such an order. Because $1 \cdot v$ may not be v, we introduce a notion of *pseudo-product*, written π (see Fig. 19). We then lift the *total order* over variables to a *total preorder* over pseudo-products. We then define a normal form, written ϕ, as either a variable v or as the sum of a scalar c (possibly null) with a strictly increasing sequence (possibly empty) of pseudo-products (see Fig. 19). Last, the affine normalization is mainly reduced to two operations "$c \cdot \phi$" and "$\phi_1 + \phi_2$" (detailed in Appendix A.2) preserving normal forms by applying Fig. 17 equations.

Implementation. Following the principles described in Six et al. [2022, §7.6], rewritings are handled *during both* symbolic executions of source and target blocks: we apply a normalization

[19] `Vundef` represents an undefined value (e.g. of an uninitialized register) that is not a failure unless it is observable.

[20] Last, if v and v' are two pointers in the same block, then $v - v'$ computes their relative offset. Thus, if v is a pointer, then $v + (v - v) = v$ but $(v + v) - v =$ `Vundef`. This lack of associativity of operator "$-$" within sums explains why it is not yet supported in our LSR.

procedure to every right hand-side of an assignment. Its formal correctness expresses that given a symbolic value sv_1, it returns a symbolic value sv_2 such that for all register and memory states, if evaluation of sv_1 does not fail, then sv_2 evaluates to the same value as sv_1. Its implementation depends on the architecture. Currently, our affine normalization procedure is only implemented for RISC-V. Moreover, our framework has several—potentially overlapping—sets of rewriting rules, so each instantiation of the symbolic execution is parametrized with one of these sets: the scheduling pass does not need any rule, while the SR is configured with our affine normalization procedure. Selecting rules according to oracles avoids slowing down the verifier with useless rewritings.

Our affine normalization needs to integrate affine forms with symbolic values that do not represent pure affine computations. In particular, affine variables in Fig. 19 actually reify symbolic values whose root is not an affine computation (i.e. neither a 64-bits immediate, nor a "$\cdot$", nor a "+"). In practice, we do not introduce explicit affine variables, but rather define a total order on the reified symbolic values. Thanks to hash-consing, each symbolic value stores in its root an integer that is expected to uniquely identify it [Boulmé 2021, §3.3.2]. This identifier gives a simple and efficient total order over symbolic values that we do not even need to prove correct. Similarly to [Boulmé 2021; Six et al. 2022], the correctness of our normalization modulo hash-consing is only derived from a sound Coq model of OCaml pointer equality.[21]

To facilitate the proof of recursive operations over affine forms and future extensions, we extend the representation sv of symbolic values (sketched Fig. 5) with a dedicated constructor, written "$\texttt{Sfoldr}(op, \overrightarrow{sv_i}, sv_0)$", and semantically equivalent to "$\texttt{Sop}(op, [sv_0, \texttt{Sop}(op, [sv_1, \ldots])])$". This enables us to represent affine forms written "$c + \Sigma\vec{\pi}$" in Fig. 19 as "$\texttt{Sfoldr}(+, \vec{\pi}, c)$" in our implementation. See Appendix A.2 for more details.

6.4 Strengthening our Strength-Reduction with a Prior Integer Promotion

Currently, our formally-verified strength-reduction is still embryonic. Its scope is limited to affine arithmetic on `long` (signed and unsigned ones) for RISC-V 64 bits architectures. We now sketch how we mitigate this limitation (Appendix A.3 describes other obstacles to generalizing our SR).

On 64-bit architectures, a 32-bit index (`int`) must be cast to 64 bits before being scaled and added to the base address. Due to overflows, it can be wrong to strength-reduce the scaling. We overcame this limitation of our SR by combining a preliminary pass that combines two techniques: (1) a formally verified interval analysis able to justify the absence of overflow—for instance, under a loop condition of the form "$i < n$" for some n, then we know that the increment of i cannot overflow[22] (2) the use of the intervals found by this static analysis within an oracle in order to validate the "promotion" of `int` variables as `long`: if there are no overflows, sign-extension (or zero-extension) and addition commute. The transformation performed by this oracle is itself validated by our symbolic simulation test modulo appropriate invariants. The intervals found by the static analysis are themselves integrated within the symbolic simulation, as explained in §3.5.

7 ADAPTING THE LAZY TRANSFORMATION ALGORITHMS OF KNOOP-ET-AL

The previous sections present LCM and LSR as two distinct optimizations, but in our implementation, they are achieved within a single pass, from a single oracle. This oracle actually combines the LCM & LSR algorithms of Knoop et al. [1995, 1992, 1993] and adapts them for our validator by symbolic simulation. Our main contribution consists in a generalization of their LSR: (1) that operates over basic blocks by adapting the analysis of Knoop et al. [1993], in the same fashion as it was done

[21] A bug in the hash-consing mechanism makes—at worse—the verifier fail to prove some expected equalities.

[22] In C, overflow has undefined behavior in signed arithmetic, so if the loop index is signed (`int`), as it often is, we could simply assume overflow does not occur. Signedness information is however not available at that stage in CompCert.

in [Knoop et al. 1995] for LCM; (2) which integrates a rewriting procedure to widen the scope of SR over sequences of operations, rather than on each instruction independently; (3) capable of inferring invariants from data-flow equations (including liveness analysis) for the translation validation. We also improve LCM to infer invariants, although we had to restrict the original algorithm because our validator forbids anticipating potentially trapping operations (e.g. loads).[23]

The framework introduced by [Knoop et al. 1995, 1993] combines several advanced data-flow analyses, and its full description requires dozens of pages. The precise description of our algorithms is thus provided in a dedicated "companion" paper [Gourdin 2023]. Below, we simply give a high-level overview of our adaptations to the original LCM & LSR.

7.1 A Brief Overview of the LCM & LSR Analyses

Roughly speaking, the LCM algorithm of Knoop et al. [1995] consists in analyzing which "candidate instructions" can be "moved", and then, where it is "the best place" to move them. This analysis results from the computation of a dozen of predicates by candidates. For example, a predicate called *transparency* is satisfied by blocks that do not erase the candidate's dependencies. Computing this simple predicate only requires a local analysis of each block. In more complex cases, predicates are computed as greatest fixed points of data-flow equations. Operating over basic blocks makes these computations more efficient, but also more complex.[24]

Extending the LCM process sketched above to LSR is mainly about merging the candidate detection phases of both algorithms, and interleaving the LSR specific predicates with those of LCM. A naive generalization of the LCM into LSR would only require a new local predicate (that we call *pseudo-transparency*, as it is a weaker version of it), and a new data-flow one for computing points where an *update* compensation addition must be inserted. Nonetheless, this naive approach would still generate some redundant computations. In contrast, our implementation reproduces the three refinements steps described in [Knoop et al. 1993], and thus adds three data-flow predicates and one global, non data-flow intermediate predicate.

After having detected candidates, our combination of LCM & LSR (that we named "LCT", for Lazy Code Transformations) sort candidates topologically, and iterates over them. Each time, it starts by calculating local and global (including data-flow) predicates, and it rewrites the CFG for the current candidate, while substituting the freshly allocated variable locally in basic-blocks. Thanks to this enhancement, our LCT is capable of reducing instructions sequences, as illustrated in §6.2.

7.2 Co-design for Validation and Invariant Inference

A strong advantage of using such data-flow based algorithms is that we can reuse the computed predicates in order to *infer the hints* needed by our validation mechanism. By combining a liveness analysis (which is in any case required for the symbolic simulation validator) with the data-flow information from LCM, we designed a generic method inferring the exact set of nodes where (for each moved or reduced candidate) a gluing or history invariant must be inserted. Our method to determine the invariants' insertion points is executed after the main loop that iterates over candidates of §7.1. Internally, the LCT oracle represents predicates as bit vectors: each bit is the value of the predicate (true or false) for a given basic block. Hence, the invariant generation efficiently computes a vector V_p of *preservation points* that contains "true" when a gluing invariant must be annotated.

[23]We partially mitigate this limitation thanks to control-flow graph restructurings (duplications and factorization) in pre-/post-processing of our oracle, as previously described.

[24]For each candidate, basic blocks are split in two parts: an *entry* part containing all the instructions until (and including) the first modification of the candidate's dependencies, and an *exit* part with all the remaining statements. All predicates, except *transparency*, apply to block parts.

For a non-trapping candidate, V_p is the disjunction of two situations. First, if the candidate was *replaced* and not *inserted* in the block (we compute this for all blocks at once using a bitwise "and" on the *insertion* and *replacement* predicates); and second, if it is neither marked as *isolated* nor *delayed* (those two information are also encoded in specific predicates, and correspond, resp., to cases where an insertion would be either useless because *isolated*, or *delayable* later). When the candidate may trap, we reuse V_p in conjunction with an *anticipability* constraint, so that we only insert gluing invariants from the first apparition point of the candidate in the source. This constraint enforces the anticipability limitation of trapping instructions previously mentioned.

Finally, and in contrast with gluing invariants' preservation points which only rely on bitwise operations, the places where a history invariant must be inserted are slightly more complex to infer. Roughly, a history invariant is mandatory in a block if the candidate depends on a constant defined before in the CFG, stored in a fresh variable, and when either the block is a gluing preservation point (in V_p), or the constant's fresh variable is live in the block.

8 EXPERIMENTAL EVALUATION

CompCert's formal proof ensures partial correctness: if compilation succeeds, then it is correct, but there is no formal proof that it succeeds. In particular, there is no formal guarantee that our checkers will succeed in validating our untrusted oracles. We checked that CompCert successfully compiled a test suite of thousands of C programs, some from non-regression tests, some from real applications (including large programs such as GCC and OCaml themselves), and others generated from fuzzers (CSmith [Yang et al. 2011], YarpGen [Livinskii et al. 2020], and CCG).[25] When the oracle terminated with an internal error, or when the checker refused the transformation, we reduced the test case using C-Reduce[26] [Regehr et al. 2012] and fixed the oracle and/or the checker. See details on our test methodology in [Monniaux et al. 2023].

8.1 Compilation Time Measurements

Translation validators of complex optimizations may not scale up on large programs. To ensure that our formally verified LCT was efficient even on heavy applications, we instrumented the code to time both the LCT oracle and the validator by Symbolic Executions (SE). The correlation between their execution time is near 99%, considering the four benchmark suites of §8.2. Fig. 20 graphically represents those timings measures w.r.t. the total number of BTL instructions for every benchmark listed in §8.2. Each point in the figure correspond to a single benchmark whose timings and number of instructions were summed for all its BTL functions. Due to a fixed maximum of the candidate number (currently 64), the oracle seems linear in the number of instructions. And the validator seems a bit faster than the oracle for a given benchmark size.

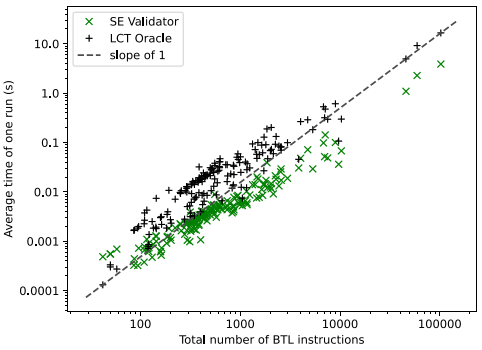

Fig. 20. Oracle and Validator times w.r.t. the number of instructions (logarithmic scale)

8.2 Performance Benchmarks

We evaluated the performance improvements of our optimizations over two architectures: for AArch64, a Cortex-A53 in-order dual-issue core (Raspberry Pi 3 Model B+ Rev 1.3); for RISC-V,

[25] https://embed.cs.utah.edu/csmith/ https://github.com/intel/yarpgen https://github.com/Mrktn/ccg
[26] https://embed.cs.utah.edu/creduce/

a SiFive in-order dual-issue U740 core (HiFive Unmatched). We measured the performance over four suites of benchmarks (distinct from our suite testing the "functional correctness"): (1) a subset of the LLVM test suite[27]; (2) a subset of MiBench [Guthaus et al. 2001]; (3) PolyBench [Pouchet 2012]; (4) TACLeBench [Falk et al. 2016]. To obtain reliable results and avoid common measurement bias [Curtsinger and Berger 2013; Mytkowicz et al. 2009], we always perform multiple runs (between five and twenty) and filter them by setting a threshold on the relative standard derivation (here 2%).

All our results are the average execution time (i.e., number of cycles) gain in percentage w.r.t. the base version. A lower number indicates slower execution, and vice versa. The execution time gain for a configuration C w.r.t. a reference version R is computed using the formula: $gain(C) = ((R - C)/C) \times 100$. The latter gives the evolution rate *relatively* to R. For instance, if $R = 1000$ and $C = 500$ cycles—so C is twice *faster*—then $gain(C) = 100\%$; in contrast, if $R = 500$ and $C = 1000$—so C is twice *slower*—we have $gain(C) = -50\%$. Our benchmark evaluation toolkit will be available online, with our exhaustive list of tests.

We measure the performances of our optimizations within our CompCert fork: our port of the superblock scheduling and CSE3 [Monniaux and Six 2021, 2022] are always enabled; and the comparison is always done w.r.t. the mainline version of CompCert (3.12). Ideally, it would be interesting to lead an experimental comparison with the LCM implementation of Tristan and Leroy [2009]; this would however be very difficult, since their framework is unmaintained, undocumented and designed for a much earlier version of CompCert (e.g. before the integration of 64-bit architectures that we target).

First, Table 1 shows a sample of results on various benchmarks, and focuses on evaluating the performance of the whole LCT algorithm (LCM + LSR) on the U740 core. We propose this individual view to highlight the variability of results according to the input benchmark: the sample includes benchmarks from all suites, and the whole PolyBench suite (at line "PolyBench/*"). The table compares GCC -01 (11.3.0), a "Base1" version of our fork (with scheduling, CSE3, and loop peeling to facilitate LICM), and "Base1"+LCT versus the mainline CompCert. On average, we are still slower than GCC -O1; nevertheless, the LCT implementation closes the gap and even exceeds the GCC's performance in some cases. On the other hand, there are still benchmarks where our optimizations are not perfectly tuned: for instance, on "MiBench/stringsearch" in Table 1, enabling the LCT dramatically decreases timings. This comes from a register pressure issue: without loop peeling, the LCT improves the time by nearly 40% on the latter benchmark; but with the unrolling, the number of alive registers becomes too high and makes the allocator spilling. Such a situation remains, fortunately, quite rare.

Second, another set of results is provided in Table 2: this time, we compared a "Base2" version of our fork (still with scheduling and CSE3, but with loop rotate and loop body unrollings instead of peeling, since they help in lifting conditionals), and we incrementally added if-lifting (which includes register renaming), LCM, and LSR (for RISC-V only). The table contains results for both the AArch64 and RISC-V backends, and distinguishes the four suites of benchmarks. The mean gain observed on our if-lifting implementation is far more interesting than the one observed by Six et al. [2022, §9].

In conclusion, observing that SR has the ability to significantly reduce execution time, we are encouraged to broaden its application scope in the future.

9 RELATED AND FUTURE WORKS

Besides CompCert, Vellvm [Zhao et al. 2012] and CakeML [Kumar et al. 2014] are two other compilers, formally verified with an interactive proof assistant (resp. Coq and HOL4). To our

[27] Accessible at https://github.com/lac-dcc/Benchmarks.

Table 1. Sample of results for GCC, our fork Base1=(scheduling+CSE3+unroll single), and Base1+LCT vs. mainline CompCert on **RISC-V**

Setup	GCC-O1	Base1	**Base1 + LCT**
LLVMtest/fpconvert	+24.22%	+7.9%	+17.15%
LLVMtest/matmul	+15.9%	+115.05%	+144.11%
LLVMtest/nbench_bf	+74.58%	+11.84%	+24.51%
MiBench/jpeg	+27.75%	+20.62%	+24.75%
MiBench/sha	+92.43%	+45.68%	+51.73%
MiBench/stringsearch	+133.34%	+40.28%	-10.15%
PolyBench/*	+64.05%	+38.06%	+46.23%
TACLeBench/bsort	+49.04%	+9%	+33.16%
TACLeBench/deg2rad	+56.75%	+41.5%	+50.28%
TACLeBench/md5	+42.18%	+18.59%	+47.93%

Table 2. Base2=(scheduling+CSE3+unrollings), If-lifting (§4), LCM (§5) & LSR (§6) vs. mainline CompCert on RISC-V & AArch64

	LLVM tests		MiBench		PolyBench		TACLeBench	
Setup	RV	A64	RV	A64	RV	A64	RV	A64
Base2	+19.9%	+23.6%	+19.0%	+19.2%	+38.2%	+68.9%	+19.4%	+25.7%
+If-lifting	+27.4%	+31.5%	+24.2%	+21.5%	+40.7%	+69.1%	+22.0%	+28.5%
+LCM	+27.8%	+31.0%	+24.7%	+25.6%	+43.8%	+84.2%	+24.1%	+33.4%
+LSR	+29.4%	-	+26.4%	-	+55.1%	-	+26.2%	-

knowledge, none of them attempts to leverage translation validation as we do. They do not integrate the kind of formally verified optimizations that we support. Our translation validation framework—within the CompCert compiler—formally verifies various intra-procedural optimizations: superblock scheduling [Hwu et al. 1993], lazy code motion [Knoop et al. 1992] and strength-reduction [Knoop et al. 1993]. It extends the translation validation techniques of Tristan and Leroy [2008] and Six et al. [2022], which were themselves inspired by King [1976]; Necula [2000]; Samet [1976].

Actually, since Pnueli et al. [1998], *translation validation* has become a quite intensive research area. For example, Churchill et al. [2019] and Kasampalis et al. [2021] propose equivalence checkers for translation validations of compiler optimizations, based on SMT-solving. As explained in introduction, our approach induces very different concerns than such classical approaches of translation validation. In our work, "*synchronization points*" and "*invariants*" between source and target code (aka "*program alignment*") are directly given by the oracles that actually perform the translations. Generating these informations inside the transformation phase is not very difficult: it requires quite simple refinements of translation algorithms; in contrast, reconstructing them from compiler output is hard. We thus do not really experience "*false alarms*", because our translations are designed with the validator limitations in mind. In addition, the design of our validators is very constrained, because we want them to be formally verified, lightweight at compile-time (i.e. quasi-linear in practice), and predictable on "*false alarms*". This prevents us from using SMT-solvers in the current state of the art. Below, we only focus on the most related works in translation validation, while attempting to sketch some future works. See [Clément 2022, §8.1] for a recent and complementary bibliography.

General-Purpose Predictable Validators. Tate et al. [2011] generalized the notion of *symbolic value* with *e-graphs* (or expression graphs): such an e-graph represents the contents of a single variable after any arbitrary computations, even including loops. This enables reasoning on loop transformations only by rewritings these e-graphs (e.g. without explicit invariant inference). Moreover, in order to "*simultaneously explore all possible sequences of optimizations*", they applied a saturation technique over their e-graphs. Noticing that saturation does not scale well on large programs, Tristan et al. [2011] experimented with normalized rewriting instead, arguing it is sufficient for

translation validation. Indeed, they succeeded to validate many existing LLVM optimizations, without instrumentation nor hints from these transformations. However, they acknowledged that the algorithmics of their translation validator is complex (and thus probably difficult to formally verify). Moreover, they did not attempt to be sound w.r.t undefined or diverging behaviors, whereas these cases are often complex to handle in CompCert correctness proofs.

Kang et al. [2018] have proposed a variant of formally verified translation validation, called "*Verified Credible Compilation*" (inspired by Rinard and Marino [1999]). They validate the results of two existing optimizations of LLVM (register promotion and global value numbering) with a dedicated oracle that generates proofs in a *Relational Hoare Logic* (inspired by Benton [2004]), itself formalized in Coq. Their tool helped to find several new miscompilation bugs in these optimizations. However, it remains unclear what guarantees are provided to final users of the whole compiler.

Loop Optimizations. Tristan and Leroy [2009] proposed a Coq-verified translation validator for LCM, based on two formally-verified data-flow analyses, *availability* and *anticipability*. These analyses have quite high algorithmic complexity (cubic for availability). In contrast, our validator does not use them. The availability analysis is replaced by our gluing invariants which are themselves provided "for free" by the oracle: we hence avoid to replay a data-flow analysis already performed by the oracle. Hence, from the analysis of §3.6 over the case of our LCM (with a bounded number of candidates in invariants, without rewriting rules and working on basic blocks—ie, with a bounded number of block exits), our validator is quasi-linear in practice: its worst-case complexity is $O(n \times l)$ where n is the size of the code and l the number of maximal simultaneously live registers.

Moreover, we combine LCM with CFG restructurations, which validates some PRE of trapping instructions (a feature that they did not provide). Our CFG restructurations also partly compensate the lack of anticipability checking that is necessary to validate FRE of trapping instructions. In future works, our symbolic simulations might check the anticipability of trapping instructions, with a dedicated notion of *prophecy* (see item (3) in §3.6).

Monniaux and Six [2021, 2022] proposed three dedicated and formally-verified passes to produce an efficient CSE optimization with LICM integrated into CompCert. After loop unrolling (as we do), they run an untrusted analysis to collect inductive invariants in hash-consed sets, whose inductiveness is checked by a proven verifier, before eliminating redundant computations. This last phase actually consists of three sub-steps: replacing computations by `move` operations; and replacing moves from a variable to itself by "no-op"; then apply an existing DCE pass. On the one hand, their decomposition simplifies the formal proofs of each single pass. On the other hand, it can only validate some PRE. In contrast, our approach aims to validate a wider class of transformations, e.g. including scheduling and strength-reduction. However, their optimization includes an elimination of redundant conditions, a feature we leave to future work.

Our framework validates superblock scheduling which interleaves the computations of successive iterations within a loop. Tristan and Leroy [2010] showed that symbolic simulation is able to validate even more advanced scheduling techniques, such as *software pipelining* [Lam 1988]. It remains however to understand how their technique could be integrated to our framework.

Validating optimizations in the polyhedral model, Clément and Cohen [2022] support much more advanced loop transformations than we do; but we support a much wider class of input programs within a general-purpose compiler. While special-purpose translation validation is in the spirit of CompCert design, it seems very challenging (but very interesting) to integrate such advanced techniques within a formally verified general-purpose compiler.

Formally Verified SSA Optimizations. Demange [2012]; Demange and Fernandez de Retana [2016]; Demange et al. [2015] propose Coq-verified translation validators for SSA optimizations within CompCert. In their approach, validators rely on strong SSA invariants (e.g. dominator sets). In

an alternative design, we could imagine extending BTL with optional parallel moves of register at exit points. This would allow representing (partial) SSA forms within BTL using Appel [1998] representation: without explicit ϕ-nodes, but rather by encoding them with explicit parallel moves on joining edges. The validator would completely ignore SSA-invariants, but would be able to compare SSA forms with non-SSA ones. Moreover, only SSA oracles would have to maintain SSA-invariants, without need of formal proof of this.

REFERENCES

Martín Abadi and Leslie Lamport. 1991. The Existence of Refinement Mappings. *Theor. Comput. Sci.* 82, 2 (1991), 253–284. https://doi.org/10.1016/0304-3975(91)90224-P

Jean-Raymond Abrial. 1996. *The B-book - assigning programs to meanings.* Cambridge University Press. https://doi.org/10.1017/CBO9780511624162

Andrew W. Appel. 1998. SSA is Functional Programming. *SIGPLAN Not.* 33, 4 (apr 1998), 17–20. https://doi.org/10.1145/278283.278285

Nick Benton. 2004. Simple Relational Correctness Proofs for Static Analyses and Program Transformations. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Venice, Italy) *(POPL '04)*. Association for Computing Machinery, New York, NY, USA, 14–25. https://doi.org/10.1145/964001.964003

Rastislav Bodík, Rajiv Gupta, and Mary Lou Soffa. 1998. Complete Removal of Redundant Expressions. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation* (Montreal, Quebec, Canada) *(PLDI '98)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/277650.277653

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

Berkeley Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. 2019. Semantic Program Alignment for Equivalence Checking. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 1027–1040. https://doi.org/10.1145/3314221.3314596

Basile Clément and Albert Cohen. 2022. End-to-End Translation Validation for the Halide Language. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 84 (apr 2022), 30 pages. https://doi.org/10.1145/3527328

Basile Clément. 2022. *Translation Validation of Tensor Compilers.* Ph. D. Dissertation. École Normale Supérieure, Paris, France. https://basile.clement.pm/papers/phd.pdf

Charlie Curtsinger and Emery D Berger. 2013. STABILIZER: Statistically Sound Performance Evaluation. In *ASPLOS'2013*. ACM, 219–228. https://doi.org/10.1145/2451116.2451141

Delphine Demange. 2012. *Semantic Foundations of Intermediate Program Representations.* Ph. D. Dissertation. École Normale Supérieure de Cachan, France. http://people.irisa.fr/Delphine.Demange/papers/DemangePhD.pdf EAPLS Best PhD Dissertation Award 2012. Gilles Kahn PhD Thesis Award 2013.

Delphine Demange and Yon Fernandez de Retana. 2016. Mechanizing conventional SSA for a verified destruction with coalescing. In *25th International Conference on Compiler Construction*. Barcelona, Spain. https://doi.org/10.1145/2892208.2892222

Delphine Demange, David Pichardie, and Léo Stefanesco. 2015. Verifying Fast and Sparse SSA-based Optimizations in Coq. In *24th International Conference on Compiler Construction, CC 2015*. London, United Kingdom. https://doi.org/10.1007/978-3-662-46663-6_12

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

Léo Gourdin. 2023. "Lazy" Code Transformations in a Formally Verified Compiler. In *ICOOOLPS 2023: Proceedings of the 18th ACM International Workshop on Implementation, Compilation, Optimization of OO Languages, Programs and Systems, July 2023 (LNCS)*. ACM. https://doi.org/to_appear

M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown. 2001. 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)*. IEEE, Austin, TX, USA, 3–14. https://doi.org/10.1109/WWC.2001.990739

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

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

Jeehoon Kang, Yoonseung Kim, Youngju Song, Juneyoung Lee, Sanghoon Park, Mark Dongyeon Shin, Yonghyun Kim, Sungkeun Cho, Joonwon Choi, Chung-Kil Hur, and Kwangkeun Yi. 2018. Crellvm: Verified Credible Compilation for LLVM. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 631–645. https://doi.org/10.1145/3192366.3192377

Theodoros Kasampalis, Daejun Park, Zhengyao Lin, Vikram S. Adve, and Grigore Roşu. 2021. Language-Parametric Compiler Validation with Application to LLVM. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 1004–1019. https://doi.org/10.1145/3445814.3446751

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

Claude Kirchner and Hélène Kirchner. 2014. Equational logic and rewriting. In *Handbook of the History of Logic*, Dov M. Gabbay, Jörg H. Siekmann, and John Woods (Eds.). History of Logic and Computation in the 20th Century, Vol. 9. Elsevier. https://hal.inria.fr/hal-01183817

Jens Knoop, Oliver Ruthing, and Bernhard Steffen. 1995. Optimal Code Motion: Theory and Practice. *ACM Transactions on Programming Languages and Systems* 16 (September 1995). https://doi.org/10.1145/183432.183443

Jens Knoop, Oliver Rüthing, and Bernhard Steffen. 1992. Lazy code motion. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation - PLDI '92*. ACM Press, San Francisco, California, United States, 224–234. https://doi.org/10.1145/143095.143136

Jens Knoop, Oliver Rüthing, and Bernhard Steffen. 1993. Lazy Strength Reduction. *Journal of Programming Languages* 1 (1993), 71–91.

Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A Verified Implementation of ML. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. Association for Computing Machinery, New York, NY, USA, 179–191. https://doi.org/10.1145/2535838.2535841

Monica S. Lam. 1988. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *Programming Language Design and Implementation (PLDI)*. ACM Press.

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

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

Xavier Leroy. 2011. Verified squared: does critical software deserve verified tools?. In *POPL'11*. ACM, Austin, TX, USA, 1–2. https://doi.org/10.1145/1926385.1926387

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

David Monniaux and Sylvain Boulmé. 2022. The Trusted Computing Base of the CompCert Verified Compiler. In *Programming Languages and Systems (ESOP 2022) (LNCS, Vol. 13240)*, Ilya Sergey (Ed.). Springer, Munich, Germany, 204–233. https://doi.org/10.1007/978-3-030-99336-8_8

David Monniaux, Léo Gourdin, Sylvain Boulmé, and Olivier Lebeltel. 2023. Testing a Formally Verified Compiler. In *Tests and Proofs - 17th International Conference, TAP 2023, Held as Part of STAF 2023, July, 2023, Proceedings (LNCS, Vol. to appear)*. Springer. https://hal.science/hal-04096390

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

David Monniaux and Cyril Six. 2022. Formally Verified Loop-Invariant Code Motion and Assorted Optimizations. *ACM Trans. Embed. Comput. Syst.* (mar 2022). https://doi.org/10.1145/3529507

Eric Mullen, Daryl Zuniga, Zachary Tatlock, and Dan Grossman. 2016. Verified peephole optimizations for CompCert. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, Chandra Krintz and Emery Berger (Eds.). ACM, 448–461. https://doi.org/10.1145/2908080.2908109

Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F Sweeney. 2009. Producing Wrong Data Without Doing Anything Obviously Wrong!. In *ASPLOS'2009*. ACM, 265–276. https://doi.org/10.1145/1508244.1508275

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

Amir Pnueli, Michael Siegel, and Eli Singerman. 1998. Translation Validation. In *Tools and Algorihtms for the Construction and Analysis of Systems (TACAS) (LNCS, Vol. 1384)*. Springer, 151–166. https://doi.org/10.1007/BFb0054170

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

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

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

Martin C. Rinard and Darko Marino. 1999. Credible Compilation with Pointers. In *Proceedings of the FLoC Workshop on Run-Time Result Verification*. https://people.csail.mit.edu/rinard/paper/credibleCompilation.html

Valentin Robert and Xavier Leroy. 2012. A Formally-Verified Alias Analysis. In *Certified Programs and Proofs - Second International Conference, CPP 2012, Kyoto, Japan, December 13-15, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7679)*, Chris Hawblitzel and Dale Miller (Eds.). Springer, 11–26. https://doi.org/10.1007/978-3-642-35308-6_5

Hanan Samet. 1976. Compiler testing via symbolic interpretation. In *Proceedings of the 1976 Annual Conference, Houston, Texas, USA, October 20-22, 1976*, John A. Gosden and Olin G. Johnson (Eds.). ACM, 492–497. https://doi.org/10.1145/800191.805648

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

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

Cyril Six, Léo Gourdin, Sylvain Boulmé, David Monniaux, Justus Fasse, and Nicolas Nardino. 2022. Formally Verified Superblock Scheduling. In *Certified Programs and Proofs (CPP '22)*. Philadelphia, United States. https://hal.archives-ouvertes.fr/hal-03200774

Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2011. Equality Saturation: A New Approach to Optimization. *Log. Methods Comput. Sci.* 7, 1 (2011). https://doi.org/10.2168/LMCS-7(1:10)2011

Zachary Tatlock and Sorin Lerner. 2010. Bringing Extensibility to Verified Compilers. *SIGPLAN Not.* 45, 6 (jun 2010), 111–121. https://doi.org/10.1145/1809028.1806611

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

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

Jean-Baptiste Tristan and Xavier Leroy. 2009. Verified Validation of Lazy Code Motion. In *Programming Language Design and Implementation (PLDI)*. ACM Press, 316–326. http://gallium.inria.fr/~xleroy/publi/validation-LCM.pdf

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

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

Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. 2012. Formalizing the LLVM Intermediate Representation for Verified Program Transformations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Philadelphia, PA, USA) *(POPL '12)*. Association for Computing Machinery, New York, NY, USA, 427–440. https://doi.org/10.1145/2103656.2103709

Zhide Zhou, Zhilei Ren, Guojun Gao, and He Jiang. 2021. An empirical study of optimization bugs in GCC and LLVM. *Journal of Systems and Software* 174 (2021), 110884. https://doi.org/10.1016/j.jss.2020.110884

These appendices cover implementation details or provide context about possible extensions to our translation validation framework.

A COMPLEMENTARY DETAILS ON BTL

A.1 Overview of the Formal Proof of our Simulation Checker

Our symbolic simulation checker over BTL programs enables formally proving a generic pass parametrized by an oracle. The oracle is declared as an OCaml function expecting as argument a source BTL function f (as defined in Fig. 3), and returning a pair (cfg, gm) where cfg is the target CFG and gm is a map—called the "glue map"—which associates to each block label pc a pair of invariants previously written $(\mathcal{H}(pc), \mathcal{G}(pc))$. When the checker validates the oracle results on all the functions of a source BTL program, the pass returns the target BTL program. Otherwise, the pass fails.

The pass is formally proved to perform a lockstep forward simulation, partly inspired by the proof on RTLpath performed by Six et al. [2022]. It is pictured in Fig. 21: for any blockstep on source concrete states $S_1 \rightarrow^e S_2$ (emitting a possible observational event e), for any target state S'_1 related to S_1 by the glue maps, relation written $S_1 \sim_{\mathcal{R}} S'_1$, there exists a blockstep on target concrete states $S'_1 \rightarrow^e S'_2$ such that $S_2 \sim_{\mathcal{R}} S'_2$.[28]

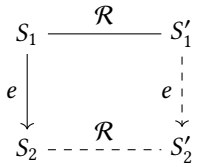

Fig. 21. Lockstep Simu.

Defining the "$\mathcal{R}$" relation is not completely straightforward because we need to express that the source call-stack is simulated by the target call-stack through the glue map of each caller function. This is necessary even if the analysis and the transformation are performed separately for each function: this simulation invariant on stackframes is needed to establish that the invariant is still true when returning from a function call. In the Coq code Fig. 22, this relation is formalized as `match_states`. Let us sketch the idea.

(1) The `match_function` relation between a source BTL function `f` and a target one `f'` expresses that `f'` is a *symbolic* simulation of `f`. In other words, the CFG entry-points are identical (condition `preserv_entrypoint`); the invariants at this entry-point only contain liveness equations of the form "$x_T = x_S$"; and, as expressed by condition `match_sexec_ok`, for any source block `ib` at label `pc`, there is a target block `ib'` at label `pc` such that the symbolic simulation condition depicted by Fig. 8 is satisfied.

(2) The `match_stackframes` predicate relates a source and a target stackframe (under a global environment `ge`). Like in RTL, such a stackframe "`(Stackframe res f sp pc rs)`" saves the execution context of the caller—`f`—in the state of the callee execution: `res` is the register to which the result should be assigned, `pc` is the returning address after the call, `rs` saves the state of the caller registers and `sp` is its stack pointer. Hence, `match_stackframes` describes how the source stackframe is simulated by a target stackframe: the target caller is a *symbolic* simulation of the source one (condition `TRANSF`); and, condition `MATCHI`, for any returned value `v` and memory `m` at the end of the call, the source state and target states after the call are related by the invariants at `pc`.

(3) The `match_states` predicate relates a source and a target (concrete) state. Like in RTL, there are three kinds of states. The kind `State` indicates a normal blockstep, whereas special kind `Callstate` (resp. `Returnstate`) indicates a context switch: entering into (resp. returning from) a function call. In these states, the source (resp. target) stack is written `stk` (resp. `stk'`). Such a stack is list of stackframes. Condition `STACKS` expresses that each stackframe of `stk` is simulated by the corresponding one in `stk'`. The `MATCHI` condition of the `match_states_intro` case expresses that normal states are related by the invariants at their common label `pc`. Note

[28] This diagram follows the usual convention: solid lines are hypotheses of the theorem, dashed lines are conclusions.

```
Record match_function (f f': BTL.function) : Prop := {
  (* f and f' have also the same signature ... *)
  preserv_entrypoint: fn_entrypoint f = fn_entrypoint f';
  trivial_histinv_entrypoint: only_liveness (history (f'.(fn_gm) (fn_entrypoint f)));
  trivial_glueinv_entrypoint: only_liveness (glue (f'.(fn_gm) (fn_entrypoint f)));
  match_sexec_ok: ∀ pc ib, (fn_code f)!pc = Some ib →
                   ∃ ib', (fn_code f')!pc = Some ib'
                   ∧ ∀ ctx, match_sexec_si ctx f'.(fn_gm) (entry ib) (entry ib') pc;
}

Inductive match_stackframes (ge: genv): stackframe → stackframe → Prop :=
  | match_stackframe_intro
      sp res f pc rs rs' f'
      (TRANSF: match_function f f')
      (MATCHI: ∀ v m, match_invs (Bcctx ge sp (rs#res ← v) m) (f'.(fn_gm) pc) (rs'#res ← v))
      : match_stackframes ge (Stackframe res f sp pc rs) (Stackframe res f' sp pc rs')

Inductive match_states (ge: genv): state → state → Prop :=
  | match_states_intro
      stk f pc sp rs rs' m stk' f'
      (TRANSF: match_function f f')
      (MATCHI: match_invs (Bcctx ge sp rs m) (f'.(fn_gm) pc) rs')
      (STACKS: list_forall2 (match_stackframes ge) stk stk')
      : match_states ge (State stk f sp pc rs m) (State stk' f' sp pc rs' m)
  | match_states_call
      stk stk' f f' args m
      (STACKS: list_forall2 (match_stackframes ge) stk stk')
      (TRANSF: match_fundef f f')
      : match_states ge (Callstate stk f args m) (Callstate stk' f' args m)
  | match_states_return
      stk stk' v m
      (STACKS: list_forall2 (match_stackframes ge) stk stk')
      : match_states ge (Returnstate stk v m) (Returnstate stk' v m)
```

Fig. 22. Simulation of Concrete BTL States - Induced by Symbolic Simulation

that because our invariants require the equality of memories at BTL block boundaries, the target and the source memories are identical (both written `m`).

Fig 8 proves the simulation pictured by Fig 21 when the source state S_1 is a normal state. Note that in BTL, the observational events e can only be emitted by final instructions. Thus, we only have to make sure that the final step of the target (the execution of fi_T) has the same observational behavior as the final step of the source (the execution of fi_S). In the case of a function call, we use the `match_states_call` case for the states after the final instruction. We use the assumption that the symbolic checker accepted all functions to prove the `TRANSF` condition for the callee. The simulation of the two other kinds of steps (entering into or returning from a call) follows from the definition of `match_states`.

A.2 Formally Verified Normalization over our (Pseudo) Affine Forms

In order to normalize an affine operation "$c \cdot sv$" or "$sv_1 + sv_2$", we first define a function $\mathcal{A}$, which maps any symbolic value sv to an affine form and satisfying the equations below (where "$[\![.]\!]$" is the evaluation function from symbolic to concrete values).

$$[\![c \cdot sv]\!] = [\![c \cdot \mathcal{A}(sv)]\!] \qquad\qquad [\![sv_1 + sv_2]\!] = [\![\mathcal{A}(sv_1) + \mathcal{A}(sv_2)]\!]$$

In other words, within the context of an affine operation, the normal forms returned by $\mathcal{A}$ preserve the semantics.[29] In practice, since the normalization is applied after each assignment (see §6.3), $\mathcal{A}$

[29] Because of the invalid equations in Fig. 18, it would be too strong to simply require "$[\![\mathcal{A}(sv)]\!] = [\![sv]\!]$".

only needs to perform a simple case analysis on the root of its argument. It is simply defined by:

$$\mathcal{A}(sv) = \begin{cases} sv & \text{if } sv \text{ matches } \texttt{Sfoldr}(+, \_, \_) \\ \texttt{Sfoldr}(+, [], c) & \text{if } sv \text{ matches } \texttt{Sop}(c, []) \text{ where c is a 64-bits integer} \\ \texttt{Sfoldr}(+, [sv], 0) & \text{otherwise} \end{cases}$$

Then, the normalization of "$c \cdot \mathcal{A}(sv)$" (resp. "$\mathcal{A}(sv_1) + \mathcal{A}(sv_2)$") reduces to a computation of the form "$c \cdot \texttt{Sfoldr}(+, \overrightarrow{\pi}, c_0)$" (resp. "$\texttt{Sfoldr}(+, \overrightarrow{\pi_1}, c_1) + \texttt{Sfoldr}(+, \overrightarrow{\pi_2}, c_2)$").

The computation of "$c \cdot \texttt{Sfoldr}(+, \overrightarrow{\pi}, c_0)$" returns "$\texttt{Sfoldr}(+, c \cdot \overrightarrow{\pi}, c\ c_0)$" where "$c \cdot \overrightarrow{\pi}$" is an instance of a "list-map" operation over pseudo-product list $\overrightarrow{\pi}$ (and is verified by applying the three equalities at the top line in Fig. 17).

The computation of "$\texttt{Sfoldr}(+, \overrightarrow{\pi_1}, c_1) + \texttt{Sfoldr}(+, \overrightarrow{\pi_2}, c_2)$" returns "$\texttt{Sfoldr}(+, \overrightarrow{\pi_1} + \overrightarrow{\pi_2}, c_1 + c_2)$", where "$\overrightarrow{\pi_1} + \overrightarrow{\pi_2}$" is very similar to the merge of sorted lists $\overrightarrow{\pi_1}$ and $\overrightarrow{\pi_2}$ for the pseudo-product preorder, except that when two compared pseudo-products are equivalent for the preorder, they are themselves merged by an operation described just below. The equivalence test on pseudo-products, described in Figure 23, uses pointer equality "==" to validate that two pseudo-products can be merged. In Figure 17, the three equations on the bottom line (from left to right) correspond to the three cases of *equiv* (from top to bottom). Each of this case in *equiv* is thus associated to a rewrite rule that merge the pseudo-products by applying the corresponding equation from left to right. For instance, supposing we have $equiv(\pi_1, \pi_2) = \texttt{true}$ for the third case, then the merge of this two pseudo-products is $2 \cdot \pi_1$, because then we know that $\pi_1 = \pi_2$ and that π_1 is a "reified" symbolic value (i.e. not having an affine computation at its root).

$$equiv(\pi_1, \pi_2) \stackrel{\text{def}}{=} \begin{cases} sv'_1 == sv'_2 & \text{if } (\pi_1, \pi_2) \text{ matches } (c_1 \cdot sv'_1, c_2 \cdot sv'_2) \\ sv == sv' & \text{else if } (\pi_1, \pi_2) \text{ matches } (sv, c \cdot sv') \text{ or } (c \cdot sv, sv') \\ \pi_1 == \pi_2 & \text{otherwise} \end{cases}$$

Fig. 23. Testing the Equivalence Induced by the Preorder over Pseudo-Products

A.3 Limitations of our Formally Verified Strength-Reduction

As detailed in §6.3, we do not fully support the standard affine arithmetic. It seems that we could recover more powerful equations by considering a *multi-sorted* equational theory. But, BTL, inspired by RTL, is an untyped language which makes this way difficult. Let us now discuss other limitations of our SR.

Targeting 32-bits architecture. §6.4 explained that our SR is currently limited to 64 bits architecture. Porting it to a 32-bits architecture seems rather straightforward. Actually, combining 32-bits and 64-bits arithmetic on a 32-bits architecture seems easier than the opposite (described in §6.4) because truncation commutes with most `long` operations (in modular arithmetic). It would only require a little generalization of the syntax of our history invariants for allowing the source registers to be defined as symbolic expressions of target registers. But, this generalization does not seem difficult because our semantics of invariants already enables it.

Eliminating/reducing loop counters. Let us consider the code generated by "`gcc -O1`" for RISC-V 64 bits on the source in Fig. 15. It is quite similar to the reduced code generated by our CompCert version, represented at Fig. 16 except that the loop is rotated (see Fig. 1; Six et al.'s version of CompCert also optionally rotates loops) and the increment "`x1=x1+1`" is eliminated from the loop. GCC compensates for this elimination by replacing the loop condition "`(x1 >=ls x2)`" by condition

"`(x10 =l x12)`" where `x12` is a fresh variable initialized by "`x12 = x2 «l 3; x12 = x3 +l x12`", before the loop. In other words, GCC replaces the source condition "`i < n`" by "`x+i != x+n`".

We cannot prove such a transformation with our validator. Indeed, such a transformation seems difficult to verify in CompCert. First, note that the replacement of condition "`i < n`" by "`x+i < x+n`" would be incorrect because of possible overflows (but "`!=`" is preserved in modular arithmetic). Second, justifying the replacement of condition "`i < n`" by "`i != n`" requires inferring the loop invariant "`i <= n`": proving such an invariant, and allowing to rewrite—under this invariant—the condition "`i < n`" into "`i != n`" require non-trivial extensions of our validator. Last, in the CompCert memory model (as in the C standard), the comparison "`x+i != x+n`" is only well-defined if "`x+i`" and "`x+n`" are valid and point within the *same* allocated block (or just after the end of the block). Hence, it is highly non-trivial to prove that if the source program has no undefined behavior then "`x+i != x+n`" is also well-defined. See related discussion at item (3) in §3.6.

This example illustrates that some seemingly simple optimizations of "`gcc -O1`" are still difficult to formally justify within CompCert.[30]

A.4 Porting Static Analyses from RTL to BTL

Official releases of CompCert run multiple analyses over the RTL intermediate representation. Since BTL has almost the same vision of the execution state (pseudo-registers and memory) as RTL, the same abstract transfer functions can be used. We defined an interface (as a Coq module type) that provides the abstract states and transfer functions of an analysis (as well as proofs of their correctness). Given an implementation of this interface, one obtain a proven abstract analysis that can be run both on RTL and BTL.

We ported into this framework the existing *value analysis*[31] of Robert and Leroy [2012], which includes an alias analysis. This analysis abstracts values within an (infinite) lattice of finite height, and also abstracts the contents of memory blocks. One important difference is that the original abstract execution of a single instruction in RTL produces one single abstract state (made of an abstract register state and an abstract memory state), which is propagated to all successors of the instruction (only branching instructions have multiple successors), whereas our framework allows the branching instructions to provide a different abstract state for each successor. We took this opportunity to slightly improve on the approach implemented in the original analysis, which did not implement transfer functions for conditions (thus, for instance, it did not track that $i = 0$ after a branch with condition $i = 0$ is taken). We added transfer functions to conditions (only, so far, for equality tests), and we also added propagation of the value of the branching variable through jump table instructions.

We used the same framework to implement the interval analysis needed for the integer promotion pass (see §6.4).

After a run of an analysis, we use some annotations to add assertions in the semantics of BTL in order to propagate its results. Those annotations are used both by the oracles and by the symbolic validation (see §3.5).

B EXTENDING SUPERBLOCK SCHEDULING WITH NON-ALIAS ANALYSES

We investigated the benefits of alias analyses for our superblock scheduler. Such analyses allow swapping stores with other non overlapping memory accesses. For this, we implemented another version of the system of constraints for the scheduling problem. We run some alias analyses on

[30] This limitation of CompCert's memory model may seem overly stringent, but it is difficult to relax while preserving the many necessary properties of the memory model.

[31] https://compcert.org/doc/html/compcert.backend.ValueAnalysis.html

```
extern void foo(int *u);

void bar(int *t) {
  int u[3];
  u[0] = t[0]+1;
  u[1] = t[1]+1;
  u[2] = t[2]+1;
  foo(u);
}
```

```
ldr  w2, [x0, #0]
add  w2, w2, #1
str  w2, [sp, #16]
ldr  w4, [x0, #4]
add  w6, w4, #1
str  w6, [sp, #20]
ldr  w3, [x0, #8]
add  w5, w3, #1
str  w5, [sp, #24]
```

```
ldr  w2, [x0, #0]
ldp  w3, w4, [x0, #4]
add  w6, w2, #1
add  w5, w3, #1
add  w4, w4, #1
stp  w6, w5, [sp, #16]
str  w4, [sp, #24]
```

Fig. 24. AArch64 Scheduling with Robert and Leroy [2012] Analysis (Right Frame)

```
void incr3(int *x) {
  x[0] ++;
  x[1] ++;
  x[2] ++;
}
```

```
ldr  w1, [x0, #0]
add  w5, w1, #1
str  w5, [x0, #0]
ldr  w4, [x0, #4]
add  w1, w4, #1
str  w1, [x0, #4]
ldr  w3, [x0, #8]
add  w2, w3, #1
str  w2, [x0, #8]
```

```
ldp  w1, w5, [x0, #0]
ldr  w3, [x0, #8]
add  w4, w1, #1
add  w1, w5, #1
stp  w4, w1, [x0, #0]
add  w2, w3, #1
str  w2, [x0, #8]
```

Fig. 25. AArch64 Scheduling with Relative Addressing Analysis (Right Frame)

memory accesses. A dependency is inserted between a read and a subsequent write (write-after-read dependency), a write and a subsequent write (write-after-write), a write and a subsequent read (read-after-write) only if according to our analyses, they *may* interfere. It is sufficient that one analysis proves noninterference for the dependency not to be inserted. We used two alias analyses.

The first alias analysis runs the per-function value analysis discussed in Sec. A.4, and uses the noninterference predicate provided by the value domain in official CompCert releases. For instance, if a pointer is proved to always point inside some global variable, and another pointer to always point inside some other global variable, then they cannot interfere—recall that attempting to move, through pointer arithmetic, between different variables has undefined behavior in C, and this is reflected by CompCert's memory model that each variable lives in a distinct memory block. This is a direct port of an existing analysis in CompCert [Robert and Leroy 2012].

In practice, the most useful noninterference case seems to be between contents of the current stack frame (`Stack` in the value domain), and anything outside the current stack frame (`Nonstack` in the value domain), such as anything pointed to by function parameters—indeed, a parameter pointer cannot point into the current stack frame, because the block of the current stack frame does not exist yet when the pointer is created. Consider the source code in Fig. 24 left frame. Without alias analysis, the AArch64 code produced appears in the Fig. 24 middle frame. The three memory assignments are not reordered by the scheduler because of a potential interference. They are carefully preserved in sequence, each as load (`ldr`), addition (`add`), store(`str`). This sequence will result in pipeline stalls, since every load takes multiple cycles even if available in the first level cache. With alias analysis, it is known that `t[.]` and `u[.]` cannot alias, because the former is outside the current stack frame and the latter is inside. Thus, the scheduler can first perform the three loads, then the three additions, then the three stores. Furthermore, noninterfering load and store to consecutive addresses are fused (into `ldp` and `stp`) by the postpass instruction rewriter of Six et al. [2022, §2]. This gives the assembly code in Fig. 24 right frame.

The second alias analysis addresses the cases where noninterference can be established because two pointers point to non-overlapping data chunks within the same object, for instance different fields inside the same structure. It performs a local abstract interpretation within the superblock. Abstract values for pointers are of the form $v_i + o$, where i is an integer index, v_i designates a

"symbolic value", and o is a constant integer offset; we also have abstract values consisting only of a constant integer. When a value comes from the starting point of the superblock, or is computed by an operation inside the superblock that is not handled by the abstraction (e.g., multiplication), that value is abstracted by $v_i + 0$ where v_i is fresh (the index i has never been used so far). When a value is computed by adding a pointer abstracted by $v_i + o$ to an integer constant c, the result is abstracted by $v_i + (o + c)$. Chunks of size s_1 and s_2 pointed to by pointers abstracted by $v_i + o_1$ and $v_i + o_2$ (note the same base pointer v_i) respectively are deemed not to interfere if the intervals $[o_1, o_1 + s_1)$ and $[o_2, o_2 + s_2)$ do not overlap.

Consider the source program of Fig. 25 left frame. Without alias analysis—see Fig. 25 middle frame—the scheduler is faced with the same issue as in the previous example: the three memory increments are kept in sequence, and the pipeline stalls. With alias analysis, like in the previous example, the scheduler can swap and group loads and stores. See Fig. 25 right frame. The only difference comes from the criteria to ensure nonaliasing. In Fig. 25, we consider offset relatively to the same base, whereas in Fig. 24, we consider the allocation class of the pointers.

While both analyses are appealing, and indeed improve code on examples such as the above, experiments showed that, often, the improvement is not noticeable even on examples where the schedule is altered by activating these analyses. Performance is improved markedly only in specific benchmarks. Our symbolic validator fully support the first analysis but only a restricted version of the second analysis. Since the latter is only used by the scheduler and for marginal gains, we postponed its complete integration in the checker.