”Lazy” Code Transformations in a Formally Verified Compiler

Léo Gourdin

"Lazy" Code Transformations in a Formally Verified Compiler

Léo Gourdin*
Université Grenoble Alpes, Verimag
Grenoble, France
Leo.Gourdin@univ-grenoble-alpes.fr

ABSTRACT

Translation validation verifies the results of an untrusted translator—called an *oracle*—at the compiler's runtime using a validator. This approach enables validating intricate optimizations without directly having to prove them correct. Parametrizing such a validator with *hints* provided by oracles greatly simplifies its integration within a formally verified compiler—such as CompCert. However, generating those hints requires adapting state-of-the-art optimizations.

The *co-design* of a validation framework supporting a class of optimizations led us to improve the *Lazy Code Motion* (LCM) and *Lazy Strength Reduction* (LSR) data-flow algorithms of Knoop, Rüthing, and Steffen. We propose an efficient implementation in OCaml combining both LCM and LSR, operating over basic-blocks, and whose result is checked by a Coq-verified validator. We show how to generate invariant annotations from the data-flow equations as hints for the defensive validation, and we introduce several algorithmic refinements w.r.t. the original papers.

Our solution is fully integrated within CompCert, and is, as far as we know, the first formally verified strength-reduction of loop-induction variables.

1 INTRODUCTION

To avoid optimization bugs [33, 37], safety standards often require relying on a certified compiler such as CompCert [20, 21], the first verified compiler used in safety-critical embedded systems [15]. However, compared to GCC and LLVM, it is only moderately optimizing. On RISC-V, which is a promising candidate for embedded, critical systems [1, 10, 22], CompCert is *much less* efficient than GCC. Indeed, the instruction set architecture (ISA) is truly reduced [35], so the compiler must be clever to generate efficient assembly code.

For instance, RISC-V addressing modes are less sophisticated than those of the AArch64 ISA, as depicted in Figure 1. When such a triplet of instructions appears in a loop, strength-reduction (SR), which *replaces costly sequences* of operations with simpler and more efficient alternatives, becomes particularly beneficial in order to minimize the number of cycles per iteration.

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

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

Figure 1: AArch64 (top) vs. RISC-V (bottom) addressing

Moreover, in embedded systems, it is common to use in-order cores (i.e. that do not dynamically reorder instructions) with a simple micro-architecture. Compiler optimizations thus have more impact on those processors than on complex ones that reorder instructions and perform speculative execution.

Typically, partial redundancy elimination (PRE) is a code motion-based optimization that can suppress duplicated computations even when they are not present on all paths (in contrast to full redundancy elimination, FRE). *Lazy Code Motion* (LCM) [18] is an improved PRE algorithm limiting the increase of *register pressure*. In fact, lifting operations in the control-flow graph (CFG) of the program certainly allows removing common subexpressions, but it can also increase the *liverange* (i.e., the interval during which a variable is live[1]). LCM solves two conflicting objectives: reaching *computational optimality* thanks to a clever placement of instructions, and *liveness optimality* by its "lazy" behavior. It moves operations high enough in the CFG to satisfy the first goal, but not more than necessary to limit register pressure (i.e. as late as possible without loosing computational optimality). Nonetheless, Bodík et al. [3] show that restructuring the CFG allows more removals than only moving instructions, as LCM does. Extending their work on code motion, Knoop et al. [19] proposed a *Lazy Strength-Reduction* (LSR) algorithm and an implementation-oriented paper about LCM [17].

Directly proving such optimizations would surely be very difficult. The *translation validation* approach [27] circumvents this issue by noticing that it is much simpler to verify the result of a transformation rather than the whole algorithm. Rideau and Leroy [29] leveraged this method for the CompCert register allocation, and Six et al. [32] for superblock scheduling and instruction rewriting. Precisely, we have recently implemented a similar, albeit more general, validator [13] supporting a large class of optimizations by *symbolic execution* [16, 26, 30]. Generalizing the approach of the previously mentioned successes, our method relies on formally verified defensive programming [4], as we have a *co-designed* integration of oracles and validators. In particular, oracles provide *hints* to validators, allowing them to avoid replaying some complex analyses. These hints guide symbolic execution and effectively reduce its complexity. Any incorrect hint would result in an error in the simulation test, preventing the production of erroneous code.

Limitations of LSR. The original LSR algorithm is only operating on a single instruction CFG, while the LCM proposed in [17] was simplified and optimized to work on basic-blocks: linear sequences of instructions with a single entry-point and a single exit-point. Notably, the basic-block structure reduces the number of nodes in the CFG (i.e. nodes become blocks), and consequently the number of predicates' values to store. In other words, the results of data-flow

*This work is supported by the French National Research Agency in the framework of the "France 2030" program (ANR-15-IDEX-0002) and by the LabEx PERSYVAL-Lab (ANR-11-LABX-0025-01).

[1] After dead-code elimination, a variable is live at program point p if it is read on a path starting from p.

equations are retained at each block rather than at each instruction, which allows the analyses to converge faster. Basic-blocks are the standard representation used in most mainstream compilers. Because of this discrepancy on the supported representation, one would be forced to redo some computations already performed by LCM (on basic-blocks) to implement LSR; the latter being relying on the same base of logical predicates.

On another note, the original LSR does not treat instructions in their topological order, and inserts a move in place of each replaced instruction. The problem with this behavior is that it prevents LSR from handling *sequences* of reducible operations. In real compilers, these sequences are often generated by instruction selection. For example, on RISC-V, a multiplication $c = i \times 10$ can be replaced with the less costly sequence "$a = i << 1; b = i << 3; c = a + b$". Usually, in a compiler like GCC or LLVM, this would not really be an issue since one can still apply LSR *before* the selection. However, to facilitate its formal proof of correctness, the instruction selection pass of CompCert operates on a structured intermediate representation, placed upstream of most other optimizations. Our LSR, which works on a basic-blocks CFG, must therefore be located downstream (to take advantage of other optimizations, and of the existing CFG-based intermediate language of CompCert which comes after selection). Let us imagine a loop containing the above sequence: an addition of the results of two shifts[2], where i is an induction variable incremented by 3 at each iteration. One would like to take out of the loop the whole sequence, and to insert an addition $c = c + 30$ instead (i.e. $30 = 3 \times 2^1 + 3 \times 2^3$). The increment on i would not be modified, and the resulting code would be much more efficient. But, the insertion of moves constrains the analyses by creating new dependencies and makes the original LSR unable to reduce such sequences.

In practice, instruction selection is not the only source of reducible sequences: they may also appear directly in the source code, or be produced for calculations of memory addresses during translations.

Problematic. As of today, CompCert is not proposing any form of loop-induction variable strength-reduction. With the validation framework we developed, implementing this kind of global optimization becomes possible as long as we manage to provide it the right invariants (the so-called *hints*). We found that LCM & LSR algorithms are well-suited for this objective by means of their data-flow results. Nevertheless, the original algorithms are still limited on some aspects, as highlighted above. Hence, our goal in this work is to co-design an enhanced version of LSR, integrating LCM, overcoming those limitations, and capable of feeding the validator with the expected, correct invariants. We have measured that adding these optimizations *significantly improves* the performance of the code generated by CompCert on 64-bit RISC-V, without degrading compilation times (including formally verified defensive checks).

Contributions & Outline. This paper focuses on improvements made to the LCM and LSR algorithms of Knoop et al. [18, 19] in the context of those works. The main contributions suggest a generalization of LSR: (i) that operates over basic-blocks by adapting the analysis of Knoop et al. [19], in the same fashion as it was done in [17] for LCM; (ii) performed together with LCM in a single transformation; (iii) which integrates a rewriting procedure to widen the scope of SR over sequences of operations, rather than on each instruction independently; (iv) inferring the invariants needed for the translation validation from data-flow equations (including liveness analysis). We provide all the *essential* information concerning our certified validator [13], although this document is not intended at describing it in details. Indeed, being provided with the necessary hints, our certified validator is able to validate a large class of inter-procedural optimizations, beyond LCM & LSR. Therefore, its formalization in the Coq proof assistant is *out of the scope* of this article (it is further detailed in [13]).

The paper is structured as follows: first, Section 2 describes the validation process of LCM & LSR, and recalls their functioning. Then, since LSR is an extension of LCM, we start by presenting our adaptation of the latter in Section 3, along with parts common to both algorithms. Our refinements to LSR are explained in Section 4, and Section 5 details a method to infer the invariants needed by the validator. Two experimental evaluations on performance and compilation (validation) time are given in Section 6. Finally, Section 7 discusses related work and concludes.

2 GENERAL OVERVIEW

The principle of LCM & LSR algorithms is to infer information from data-flow analyses on a CFG. LCM was originally designed for a single-instruction CFG, and later generalized for basic-blocks. In addition to the limitations listed in §1, our adaptation of LCM & LSR is motivated by mainly 3 reasons. (i) Basic-blocks lower the amount of hints (i.e. invariants) needed for the validation, so that the communication between the oracles and the validator is more efficient, as well as the validator itself. (ii) The data-flow approach helps in generating those invariants (see §5). (iii) They are among the most efficient algorithms of this kind not based on SSA[3]. From here, we note "LCM" the basic-block version of it, and "LCT" (for "Lazy Code Transformations") the whole oracle combining both LCM and LSR.

2.1 Quick Background on LCM & LSR

These algorithms assess the validity of certain predicates for each potential *candidate* (to move or strength-reduce). A predicate is classified as *global* if its value at each CFG block relies on a forward or backward data-flow analysis. Conversely, a *local* predicate only depends on the current block and does not necessitate any fixed point computation. LCM & LSR are built on the seminal work of Morel and Renvoise [25], in which the authors propose two global properties, namely availability and anticipability (of a given candidate)[4], and a local transparency property. A block is transparent for a candidate if it does not modify its dependencies. LCM is based on four purely unidirectional data-flow analyses, by *conceptually* splitting basic-blocks in two parts (for each candidate): an *entry* part containing every instruction up to (and including) the last modification of the candidate's dependencies, and an *exit* part,

[2]This pattern is actually implemented in the official CompCert for RISC-V.

[3]Static Single Assignment algorithms prove to be difficult to formalize in a verified compiler, as concluded by Demange and Fernandez de Retana [8]. Currently, our translation validation framework does not support SSA forms.

[4]For LCM & LSR, we reuse the vocabulary of Knoop et al. [17] where availability is called *up-safety* and anticipability *down-safety*.

consisting of all remaining statements [17, §2.3]. Consequently, all predicates (except transparency, which concerns an entire block) are duplicated for entry/exit parts of blocks. The preliminary step is to detect candidates: for LCM, any arithmetic operation or load is subject to be factorized, while LSR only targets multiplicative operations where one operand is a constant. Then, the goal is to find for each candidate (whether they are LCM or LSR ones), the optimal insertion points to pre-compute it in a fresh variable, and all the replacement points where a redundant occurrence can be replaced by a move from the fresh variable. Those locations are deduced from the data-flow predicates, as detailed in §3.1.

The analyses require a CFG devoid of critical edges: edges going from nodes with multiple successors to nodes with multiple predecessors. The usual approach is to split them by inserting an *empty* block, called *synthetic node*. Actually, we cut every edge leading to a join point in the CFG (so we insert more synthetic nodes than just by splitting critical edges). Empty nodes before joins are necessary due to the constraints imposed by our intermadiate representation. It also facilitates the LCT algorithms. Those nodes are easily removed by the "tunneling" passes of CompCert [21, §9] afterwards.

2.2 Architecture of our Solution

We implemented an intermediate representation, the Block Transfer Language (BTL), similar to the Register Transfer Language (RTL) intermediate representation of CompCert [21, §6]. The BTL CFG features a syntactical structure of loop-free blocks (in this case, basic-blocks). Since BTL is before register allocation, there is a virtually unbounded number of available registers.

Globally, our solution works as follows: first, given an RTL program P_{rtl}, an oracle translates it to a BTL program P_{btl}, while selecting basic-blocks with synthetic nodes and eventually performing some *structural* duplications/factorizations (e.g. loop-un/rerolling). This translation is validated in both directions with a specific checker. Second, the LCT oracle is called on P_{btl} and yields an optimized program P'_{btl} along with a map of *invariants* containing, for each CFG block, two sets of invariants (i.e. hints) that are needed to help the verification (as explained in §2.3). Third, the certified validator simulates both programs, block-by-block, using the provided hints: if they are equivalent, the compilation continues (coming back to RTL); otherwise, it fails and the compilation is aborted.

2.3 Symbolic Simulation Modulo Invariants

Our verified checker is based on symbolic execution (SE), a versatile technique. The principle is to *simulate* the execution of blocks of code with a symbolic representation of registers and memory (together forming a symbolic *state*).

The SE validator we implemented simulates, for both P_{btl} and P'_{btl}, the source and the optimized BTL blocks (two-by-two) until reaching a final state; then, final (symbolic) states are compared to ensure semantics preservation. When the transformation is *local* to a block, this is sufficient to prove it correct; otherwise, if the modifications are *global* over the function, we must also propagate information between block executions. To handle this, our formally verified defensive framework asks oracles for two types of invariant annotations: (i) *gluing invariants* (GI) to anticipate non-trapping operations, to remember already computed trapping operations, and to eliminate dead-code; and (ii) *history invariants* (HI) to share a common execution past (i.e. remember the values of some variables in the source). Together, those represent a semantic relation of the form "$\mathrm{HI}(r_s, m) \wedge r_t \equiv_t \mathrm{GI}(r_s, m)$" from a source state (r_s, m) to a target state (r_t, m). The memory state m is unchanged, but the register state r_s becomes r_t. Here, "$\equiv_t$" is equality of register states only for target live registers (which are syntactically provided by gluing invariants). In our implementation, we impose that invariants never fail: this restriction greatly facilitates the formal proof of simulation, and is implicitly part of the history invariant; hence, we *cannot anticipate potentially trapping instructions* (e.g. loads) w.r.t. the source.

We provide this brief overview of our simulation as a basis for the next sections. Let $\mathcal{S}$ and $\mathcal{T}$ be the source and target blocks, respectively; we note $\mathcal{G}_I$ and $\mathcal{G}_J$ (resp. $\mathcal{H}_I$ and $\mathcal{H}_J$) the in (at the entry of the current block) and out (at the entry of the next blocks) gluing (resp. history) invariants.

Figure 2 resumes how the validation is performed. Both blocks start with the same initial symbolic states δ_{s0} and δ_{t0}, obtained by applying $\mathcal{H}_I$ on the empty state ϵ. On the source side, we symbolically execute block $\mathcal{S}$ on δ_{s0} (leading to δ_{s1}), before applying the gluing invariants of the *successors'* blocks $\mathcal{G}_J$. Symmetrically, on the target side, we apply the gluing invariant of the current block $\mathcal{G}_I$ (leading to state δ_{t1}), and then execute block $\mathcal{T}$ itself. These two steps lead, respectively, to states δ_{s2} and δ_{t2}.

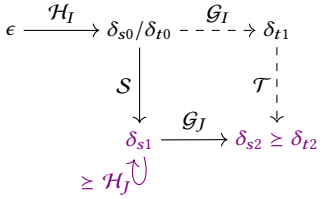

Figure 2: Modulo Invariants Simulation Diagram

The overall simulation theorem (proved in Coq)[5] states that the semantics is *preserved* under the validity of two predicates (in the sense of CompCert's forward simulations). First, the application of the output history invariant on δ_{s1} must produce a symbolic state that simulates δ_{s1} on the source's variables. In other words, the application of $\mathcal{H}_J$ on δ_{s1} should be the identity. This enforces the correctness of history invariants: as they *replay* a past execution, they must *already hold* on the source's side. Second, δ_{s2} must simulate δ_{t2} on the target's variables (so that we implicitly encode a liveness validation) for the gluing invariants to be correct. Those two properties are colored in violet in Figure 2, where "$\geq$" means *simulates*.

2.4 LCT of a Running Example

Let us explain both LCM and LSR on a simple example illustrating the anticipability constraint of §2.3. The simple C code of Figure 3 features two loop-invariant computations, the square of "y" and the load of "*p", denoted in blue. In addition, the multiplication of the loop-induction variable "i" by the constant 5 (in teal color) is prone to be reduced.

Ideally, we would like these two operations to be taken out of the loop, as it would be done by the original LCM. In fact, one may notice that the load (which may trap) will always be executed, so it

[5]The whole code is available online at: https://gricad-gitlab.univ-grenoble-alpes.fr/certicompil/Chamois-CompCert.

can be safely moved out of the loop without adding any potential failure. However, we explained that our validation mechanism is unable to anticipate a trapping operation w.r.t. the original program.

We tackle this problem by unrolling the first iteration of the loop *before* applying LCM. The unrolled code in the left-side of Figure 4 contains two times the load, and the first occurrence is now in the loop header. The violet comment of Figure 4 indicates correspondence between variables. Remark that the multiplication of "i" (by 5) was decomposed by the CompCert instruction selection as a *sequence* of a shift and an addition (still in teal color).

```
long main(long x, long y,
          long *p) {
  long i = 0, b = 0;
  while (i < 100) {
    x *= *p;
    if (i > 35)
      b += y * y;
    x += i * 5;
    i += 3;
  }
  return x-b;
}
```

Figure 3: Running example

Our improved LCT produces the code in the right-side of Figure 4, including four fresh variables. The load is now in an auxiliary variable x12, which is substituted in place of x11 in the first instruction of the loop. The square, a loop-invariant computation, is lifted in the loop header with variable x13, and the multiplication pattern has been reduced and lifted as well. To compensate the increment[6] (of 3) over x5, the oracle inserts a new increment just before the old one. It recognizes the pattern as a product by 5, and deduces the final amount to add by multiplying it with the old increment (i.e., 5×3).

```
// x3 = x ; x2 = y ; x1 = p
Head:
  x4 = 0
  x11 = int64[x1 + 0]
  x3 = x3 * x11
  x5 = 3; goto Loop
Loop:
  x11 = int64[x1 + 0]
  x3 = x3 * x11
  if (x5 > 35) {
    x10 = x2 * x2
    x4 = x4 + x10 }
  x9 = x5 << 2
  x7 = x5 + x9
  x3 = x3 + x7
  x5 = x5 + 3
  if (x5 >= 100) goto Exit
  goto Loop
Exit:
  x6 = x3 - x4
  return x6
```

```
Head:
  x4 = 0
  x12 = int64[x1 + 0]
  x3 = x3 * x12
  x5 = 3
  x13 = x2 * x2
  x14 = x5 << 2
  x15 = x5 + x14; goto Loop
Loop:
  x3 = x3 * x12
  if (x5 > 35) x4 = x4 + x13
  x3 = x3 + x15
  x15 = x15 + 15
  x5 = x5 + 3
  if (x5 >= 100) goto Exit
  goto Loop
Exit:
  x6 = x3 - x4
  return x6
```

Figure 4: Unrolled (left) and optimized (right) RTL code

On this example, the "CSE3" of Monniaux and Six [24] would also not be able to take out the load without a previous unrolling, for the same reasons of anticipation. To the same extent, the transformation becomes possible for the CSE3 when the first iteration is duplicated (because it corresponds to a FRE); on the other hand, even if the square was duplicated before the loop, the CSE3 would be unable to eliminate it (because it corresponds to a PRE). Furthermore, in a variant of this example where "b" would be initialized at line 3 by "b = *p" instead of "b = 0", CSE3 would not factorize the "*p" redundancy, contrary to our LCM, even without unrolling. Indeed, because "b" is modified in the loop, factorizing the computation of "*p" requires introducing a fresh register that stores this computation. But, in contrast with LCM, CSE3 does not introduce any fresh register.

Finally, the two original assignments to x9 and x7 have been rewritten into a single one to x15. This illustrates how we enhanced the SR algorithm of Knoop et al. [19] by enabling the reduction of sequences. The reduction of such sequences thus improves CompCert's instruction selection.

2.5 Symbolic Simulation of our Example

We now explain how we validate by SE the transformation of Figure 4. The BTL CFG of the optimized code is provided in Figure 5, where synthetic nodes are framed in violet, and blocks are numbered in post-order (for the fixed point to converge faster). Notice that block 8 was originally a synthetic node, having served as an insertion point.

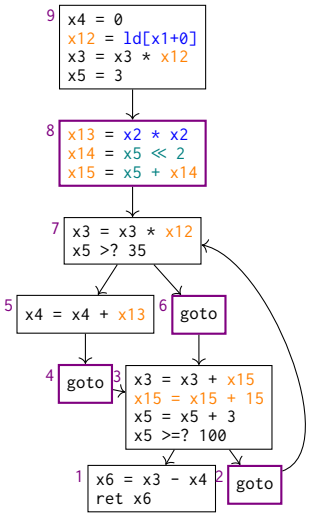

Figure 5: Optimized BTL

In our example, the simulation only requires gluing invariants, because the source's multiplication "x9=x5≪2" directly contains the immediate constant. Nevertheless, if the constant was loaded in a register in an earlier block, then we would have to maintain an history invariant between its definition and its uses in loop blocks' gluing invariants. In particular, it would have been live in the loop, and so the history invariant too. We represent invariants as sequences of register assignments, followed by a set of live registers at the block's entry. The CFG entry-point is always annotated by a pure-liveness invariant[7]; the gluing invariant for block 9 of Figure 5 is thus "([], alive = {x1,x2,x3})" ("[]" represents the empty sequence). Block 9 (on the target side) pre-computes the load in x12. To remember its value, the oracle must add an assignment in the successor's invariant (i.e. block 8), in addition to the liveness information. Therefore, block 8 is annotated with "([x12:=ld[x1+0]], alive = {x2,x3,x4,x5,x12})". Following this principle, the gluing invariants for blocks belonging to the loop propagate definitions from block 8 across the body. Hence, the same gluing invariant annotates each loop-body block (i.e. blocks 2 to 7): "([x12:=ld[x1+0]; x13:=x2*x2; x14:=x5<<2; x15:=x5+x14], alive = {x3,x4,x5,x12,x13,x15})". The exit, in block 1, only defines registers x3 and x4 as live.

Since we only rely on gluing invariants, the simulation check of §2.3 is reduced to the comparison between δ_{s2} and δ_{t2}. Example 2.1 details a part of the simulation for block 3, where symbolic states are written as parallel assignments (e.g."$x1 := x1 + x2 \| x2 := x1$"). They represent a relation from an initial concrete state to a final one in which $x1$ is incremented by the initial value of $x2$,

```
x9 = x5 << 2
x7 = x5 + x9
x3 = x3 + x7
x5 = x5 + 3
x5 >=? 100
```

Figure 6: Block 3 source

[6] Knoop et al. [19] name such increments "injuring" operations.

[7] The validator considers that functions' parameters may have any value.

and $x2$ has been assigned to the initial value of $x1$, other variables being dead or unchanged.

Example 2.1. Each exit branch of block 3 must pass the simulation test pictured in Figure 2. Let us consider the "else" branch, leading to block 2. The source block $\mathcal{S}$ is shown in Figure 6, and its symbolic execution will result in the state δ_{s1} on left column below. On the target side, we obtain δ_{t1} from the execution of $\mathcal{G}_I$, in the middle column below. As visible with variable x3 in δ_{s1}, we apply an *on-the-fly affine normalization* (see §4.3) over values.

$\delta_{s1} = x3 := x3 + 5 \cdot x5$	$\delta_{t1} = x3 := x3$	$\delta_{s2} = \delta_{t2} =$
$\quad \| \; x5 := 3 + x5$	$\quad \| \; x4 := x4$	$\quad x3 := x3 + 5 \cdot x5$
$\quad \| \; x7 := 5 \cdot x5$	$\quad \| \; x5 := x5$	$\quad \| \; x4 := x4$
$\quad \| \; x9 := 4 \cdot x5$	$\quad \| \; x12 := ld[x1 + 0]$	$\quad \| \; x5 := 3 + x5$
	$\quad \| \; x13 := x2 \cdot x2$	$\quad \| \; x12 := ld[x1 + 0]$
	$\quad \| \; x15 := 5 \cdot x5$	$\quad \| \; x13 := x2 \cdot x2$
		$\quad \| \; x15 := 15 + 5 \cdot x5$

Indeed, a naive execution would have assigned to $x3$ the value $x3 + x5 + x5 \cdot 4$ rather than the simplified form $x3 + 5 \cdot x5$. To finish, we execute $\mathcal{G}_J$ and $\mathcal{T}$ from δ_{s1} and δ_{t1}, respectively, and get the final states δ_{s2} and δ_{t2} on the right side above. As expected, considering the target's live variables[8], both symbolic states are equal, so *the optimization is validated*. Implicitly, the affine normalization of this example demonstrated that $x5 + 3 + 4 \times (x5 + 3) = x5 + 4 \times x5 + 15$.

3 OUR ANTICIPATION-RESTRAINED LCT

In this section, we implement the base of our LCT algorithm in two steps: a first phase identical to the LCM of Knoop et al. [17], that will serve as a base to include LSR in §4; and a second phase to restrict the LCM's sets of insertion and replacement points, in order to respect the anticipation constraint of §2.3.

The type of BTL instructions is parametrized by the target ISA. It groups together every basic operations under the same constructor, and features a dedicated constructor for loads. In the specific case of RISC-V, none of the BTL operations can fail, and loads are always trapping. Nevertheless, LCM must be architecture-independent; the fact is some operations can be trapping on certain backends (e.g. divisions by zero), and there also exists backends where loads can be non-trapping (e.g. on KVX [31]).

$$\begin{array}{lcl} \textit{type}\ \texttt{cm\_ckey\_t} & ::= & \texttt{CMop}(\textit{trap\_mode}, \textit{op}, \overrightarrow{r_{arg}}) \\ & \mid & \texttt{CMload}(\textit{trap\_mode}, \textit{chunk}, \textit{addr}, \overrightarrow{r_{arg}}) \end{array}$$

Figure 7: LCM candidates' key type

type cand_t ::= {		
lhs	: $blk \mapsto S_{ofs}$;	Left hand-sides positions
state	: *predicates*;	Record of bit vectors
vaux	: *r option*;	Candidate's fresh variable
memdep	: *bool*;	Memory read dependency
was_reduced	: *bool*;	SR confirmation
updated_args	: $\overrightarrow{r_{arg}}$ *option*;	Substituted arguments
orig_args	: $\overrightarrow{r_{arg}}$; }	Original arguments

Figure 8: LCT candidates' value type

For our oracle to work with both operations and loads, we define a type representing right hand-sides (RHS) of register assignments in Figure 7, that serves as a key to a hash table whose values in Figure 8 record candidate information (bold fields are mutable). In §4, the type of LCM candidates' keys of Figure 7 will be encapsulated to also support LSR ones, using the same hash table and values' type of Figure 8. Thanks to this structure, we rebuild BTL instructions from candidates' keys, and match RHS of instructions to existing candidates efficiently using the hashed key. Note that both constructors of Figure 7 include a boolean to indicate if the candidate may trap or not. Furthermore, loads are subject to a memory-read dependency that is not syntactically modeled by our intermediate language (encoded by the memdep boolean of Fig. 8). The LCT data-flow predicates (see below) are stored in the state field of Figure 8. They are encoded as bit vectors, whose indices are in *PC* (the set of block IDs). Except for transparency (which concerns a whole block), a predicate P concerns either the entry (noted $\lceil P \rceil$) or the exit (noted $\lfloor P \rfloor$) part of blocks, w.r.t. the considered candidate (cf. §2.1). Other fields of Figure 8 will be introduced as they are explained. Below, §3.1 gives a decomposition of the common steps for all types of candidates (i.e. code motion ones, trapping or not; and SR ones), and our solution to the anticipation problem is presented in §3.2.

3.1 Steps Common to All Candidates

Most implementation parts are common to every kind of candidates. As indicated in §2.1, the first step is to detect them. We traverse each block and insert a new mapping for every operation or load. The lhs field of Figure 8 is a map from block IDs to sets of offsets (i.e. positions in a given block), recording the points where the candidate was seen (assigned to a left hand-side) in the CFG. So if a candidate is detected in multiple places, we simply update the lhs map with the new position. Once the detection is done, candidates are sorted by first their appearance block (thanks to the prior post-order renumbering), and second their offset in the latter. They are then handled one by one in this topological order by the oracle.

Secondly, we compute local predicates: the *transparency* (TRANSP), being true when the candidate is untouched in a block, and the $\lceil$COMP$\rceil$/$\lfloor$COMP$\rfloor$ variants of anticipability/availability, that hold when the candidate is *computed* in the block's entry/exit part. Obtaining this information is easily done by running through each block, for each candidate. Thirdly, we compute data-flow and data-flow based predicates. We prefix them with "↑", "↓", or nothing according to whether they require a backward, forward, or no analysis, respectively. *Up-safety* (↓U-SAFE) and *down-safety* (↑D-SAFE) indicate if a computation at node n does not introduce a new value for every path leading and starting at n, respectively; *earliestness* (EARL) is true if the candidate cannot be safely placed earlier, without data-flow from the two latter; *delayability* (↓DELAY) encodes the possibility to safely move the inserted value from its earliest down-safe point; *latestness* (LATEST) represents the optimality of delayability, the maximum delay, without data-flow from the latter; and *isolation* (↑ISOL) detects the case where a computation inserted at a node would only be used (i.e. isolated) in this node.

More formally, let us assume a graph $G = (B, E, s)$ with B the set of nodes (basic-blocks), E the set of edges, and s the unique entry-point of the code. Functions $succ(n)$ and $pred(n)$ return, respectively, the set of successors and predecessors of node n; with $\prod$ and $\sum$ the finite conjunction and disjunction over those sets, respectively. The negation of a predicate P is noted $\overline{P}$. Comparing to

[8] In the general case, δ_{t2} may affect dead registers on the δ_{s2} side.

the original LCM, we do not impose G to have a unique exit-point (to be more general); but our predicates' bit vectors are initialized to *false*. After having detected candidates, and computed local predicates, the Knoop et al. [17]'s algorithm solves all of the systems of equations below:

$$\downarrow \begin{cases} \lceil \texttt{U-SAFE} \rceil(n) \triangleq \begin{cases} \textit{false} \text{ if } n = s \\ \prod_{m \in pred(n)} (\lfloor \texttt{COMP} \rfloor(m) \vee \lfloor \texttt{U-SAFE} \rfloor(m)) \end{cases} \\ \lfloor \texttt{U-SAFE} \rfloor(n) \triangleq \texttt{TRANSP}(n) \wedge (\lceil \texttt{COMP} \rceil(n) \vee \lceil \texttt{U-SAFE} \rceil(n)) \end{cases}$$

$$\uparrow \begin{cases} \lceil \texttt{D-SAFE} \rceil(n) \triangleq \lceil \texttt{COMP} \rceil(n) \vee \texttt{TRANSP}(n) \wedge \lfloor \texttt{D-SAFE} \rfloor(n) \\ \lfloor \texttt{D-SAFE} \rfloor(n) \triangleq \lfloor \texttt{COMP} \rfloor(n) \vee \prod_{m \in succ(n)} (\lceil \texttt{D-SAFE} \rceil(m)) \end{cases}$$

$$\begin{cases} \lceil \texttt{EARL} \rceil(n) \triangleq \lceil \texttt{D-SAFE} \rceil(n) \wedge \\ \qquad \prod_{m \in pred(n)} (\lfloor \texttt{U-SAFE} \rfloor(m) \vee \lfloor \texttt{D-SAFE} \rfloor(m)) \\ \lfloor \texttt{EARL} \rfloor(n) \triangleq \lfloor \texttt{D-SAFE} \rfloor(n) \wedge \overline{\texttt{TRANSP}(n)} \end{cases}$$

$$\downarrow \begin{cases} \lceil \texttt{DELAY} \rceil(n) \triangleq \lceil \texttt{EARL} \rceil(n) \vee \begin{cases} \textit{false} \text{ if } n = s \\ \prod_{m \in pred(n)} \left(\begin{array}{l} \overline{\lfloor \texttt{COMP} \rfloor(m)} \wedge \\ \lfloor \texttt{DELAY} \rfloor(m) \end{array} \right) \end{cases} \\ \lfloor \texttt{DELAY} \rfloor(n) \triangleq \lfloor \texttt{EARL} \rfloor(n) \vee \lceil \texttt{DELAY} \rceil(n) \wedge \overline{\lceil \texttt{COMP} \rceil(n)} \end{cases}$$

$$\begin{cases} \lceil \texttt{LATEST} \rceil(n) \triangleq \lceil \texttt{DELAY} \rceil(n) \wedge \lceil \texttt{COMP} \rceil(n) \\ \lfloor \texttt{LATEST} \rfloor(n) \triangleq \lfloor \texttt{DELAY} \rfloor(n) \wedge \left(\begin{array}{l} \lfloor \texttt{COMP} \rfloor(n) \vee \\ \sum_{m \in succ(n)} (\overline{\lceil \texttt{DELAY} \rceil(m)}) \end{array} \right) \end{cases}$$

$$\uparrow \begin{cases} \lceil \texttt{ISOL} \rceil(n) \triangleq \lfloor \texttt{EARL} \rfloor(n) \vee \overline{\lfloor \texttt{COMP} \rfloor(n)} \wedge \lfloor \texttt{ISOL} \rfloor(n) \\ \lfloor \texttt{ISOL} \rfloor(n) \triangleq \prod_{m \in succ(n)} \left(\begin{array}{l} \lceil \texttt{EARL} \rceil(m) \vee \\ \overline{\lceil \texttt{COMP} \rceil(m)} \wedge \lceil \texttt{ISOL} \rceil(m) \end{array} \right) \end{cases}$$

Finally, the LCM's insertion and replacement points (IR-points) are deduced using the formulas below[9]:

- $\texttt{INSERT}(pc) \triangleq \texttt{LATEST}(pc) \wedge \overline{\texttt{ISOL}(pc)}$
- $\texttt{REPLACE}(pc) \triangleq \texttt{COMP}(pc) \wedge (\overline{\texttt{LATEST}(pc) \wedge \texttt{ISOL}(pc)})$

When INSERT is true, we store the candidate in its allocated, unique auxiliary variable (in the vaux field of Fig. 8). In every node marked as REPLACE, the candidate is replaced by a move from vaux.

3.2 IR-points for Trapping Instructions

However, the above IR-points let LCM anticipates trapping instructions, while our validator only allows one to move them if they were already computed before in the source. We sketch a restrictive algorithm to calculate IR-points for trapping instructions. The idea is to start by computing the set of block IDs where we may *replace* a trapping candidate[10]. We traverse the CFG from the entry-point, and remember each block ID satisfying two *necessary* conditions: (i) the candidate appears in the entry-part; and (ii) the entry-part is "up-safe". Indeed, as stated by (i), we cannot eliminate a trapping instruction if its dependencies are modified: this means that replaceable trapping candidates are at block entry. Point (ii) reflects the availability condition (we cannot eliminate an unavailable computation). Recall the notion of block's entry part for candidates of §2.1. The result is returned by the COMPUTE_POT_REP(entry) function as a set $P = \{pc_p \mid \lceil \texttt{COMP} \rceil(pc_p) \wedge \lceil \texttt{U-SAFE} \rceil(pc_p)\}$.

From there, we need to ensure that these points are actually reachable from a previous calculation of the candidate. For a given $pc_p \in P$, FILTER_COMP_BLOCKS(pc_p) finds the set I of available previous calculations (e.g. usable to factorize the candidate). It is defined as the set of pc_i such that $pc_i \neq pc_p \wedge (\lceil \texttt{COMP} \rceil(pc_i) \vee \lfloor \texttt{COMP} \rfloor(pc_i))$, and such that *there exists a path* from pc_i to pc_p preserving the transparency property of the candidate. Thus, I groups nodes where we should insert *and* replace the candidate.

Algorithm 1 IR-points for trapping instructions

```
1:  procedure COMPUTE_LCM_TARGETS_TRAP(entry, cand)
2:      st ← cand.state
3:      P ← COMPUTE_POT_REP(entry)
4:      for pc_p ∈ P do
5:          I ← FILTER_COMP_BLOCKS(pc_p)
6:          if |I| > 0 then
7:              st.⌈REPLACE⌉(pc_p) ← true
8:              if st.⌊COMP⌋(pc_p) then
9:                  st.⌊INSERT⌋(pc_p) ← true
10:                 st.⌊REPLACE⌋(pc_p) ← true
11:         for pc_i ∈ I do
12:             if st.⌈COMP⌉(pc_i) then
13:                 st.⌈INSERT⌉(pc_i) ← true
14:                 st.⌈REPLACE⌉(pc_i) ← true
15:             else if st.⌊COMP⌋(pc_i) then
16:                 st.⌊INSERT⌋(pc_i) ← true
17:                 st.⌊REPLACE⌋(pc_i) ← true
```

Using those two functions, we define in Algorithm 1 the main procedure used to fill the candidate's IR-points. When $I = \emptyset$, we abandon the potential replacement in block at pc_p (equivalently to the isolation predicate of §3.1). Otherwise, $\lceil \texttt{REPLACE} \rceil(pc_p)$ is set to true. Moreover, if the block also contains an exit computation of the candidate, then the latter must be saved into its auxiliary variable[11] (lines 8-10 in Alg. 1). Finally (lines 11-17), both INSERT and REPLACE predicates are set to true for all $pc_I \in I$ (we set their entry variant if the node has an entry computation, and their exit one otherwise).

4 ITERATIVE LSR WITH SUBSTITUTIONS

Akin to Knoop et al. [19], we refined our LCM into a SR algorithm. LSR candidates are multiplications of the form "$v \times c$", between a variable v and a constant c. The LSR principle is to weaken the LCM's notion of transparency by considering that *additions* $v = v + c'$ with a constant c' (named *injuring* operations), do not break the transparency. Thus, multiplications are moved as if they were LCM candidates. To compensate the effect of additions on v, the algorithm inserts *update* assignments: for a candidate relocated in auxiliary variable v', an addition "$v' = v' + (c \times c')$" may be inserted in each block containing an injuring addition.

The original—non basic-block based—LSR was refined in three stages to overcome some of its limitations:

(1) **R1**: avoids inserting an update addition (i.e. an increment of the SR variable) if a multiplication (i.e. the candidate itself) must be inserted on the same path;
(2) **R2**: finds the "best" insertion point (for the multiplication), considering lifetime using the delay, latest, and isolation analyses;

[9] Either for entry or exit parts by substituting predicates accordingly.
[10] We only *factorize* trapping candidates with a previous computation of the instruction, so insertion points are also replacement points.
[11] The candidate being present in both block parts, it probably needs to be recalculated: otherwise it would have been removed by the trivial CSE.

(3) **R3**: avoids having multiple update additions on the same path for the same variable.

R1 and R2 concern IR-points: the former finds substitutes for the original insertion points (without changing replacement points); and the three additional analyses of the latter minimize the live-range induced by code motion. R3 does not change IR-points, but tries to accumulate update assignments. Knoop et al. [19, §3.1.3] first compute a naive code motion (where INSERT = D-SAFE ∧ EARL), and then apply R1, R2, and R3 (in that order). Our basic-block implementation of §3.1 (inspired from Knoop et al. [17]), includes R2 *without* R1 (noted R2$^\flat$) "for free" by unifying the code motion part of predicate inference[12]. Rather than using R3 directly, we suggest an alternative, generic representation. Last, we describe a new technique to propagate results locally from previous iterations, as a fourth refinement R4 (coming after R3).

Unlike the original LSR, we refine LCM as follows:
R2$^\flat$ (§4.1) → R1 (§4.2.1) → R2 (§4.2.2) →
alternative R3(§4.3) → *new* R4 (§4.4).

4.1 Instantiating Common LCT Steps for LSR

We extend the common part of §3.1 for LSR. The `state` field (Fig. 8) contains either the "real" transparency (TRANSP) for code motion candidates, or the weak transparency (noted SR-TRANSP) for SR ones. Specific SR predicates are also stored in `state`. As LSR targets multiplications with a constant, we perform a simple *constant detection* before the candidate detection of §3.1, that builds a hash map from registers to immediate load instructions[13]. In the local analysis, we add an "injuring" predicate (noted INJURED) to the `state` being true when an argument is only "injured" by an additive operation (preserving SR-TRANSP). The TRANSP predicate (still needed in R1) is rebuilt trivially knowing that TRANSP = SR-TRANSP ∧ $\overline{\text{INJURED}}$. Executing steps from §3.1 with this notion of weak transparency gives us the R2$^\flat$ IR-points.

Example 4.1 highlights the difference between multiplicative and additive SR candidates, and the separation with LCM-only candidates. In practice, we encapsulate the type of Figure 7 with the new candidate key defined in Figure 9.

type `sr_t`	::=	`SRmul | SRadd`
type `ckey_t`	::=	`CCM(cm_ckey_t)`
	|	`CSR(sr_t, op, `$\overrightarrow{r_{arg}}$`)`

Figure 9: LCT candidate's key type

Example 4.1. CompCert's instruction selection tries to *decompose* multiplications into a sequence of one or two left shifts (powers of two). When there are two shifts, an addition of their results is appended to the sequence. Consider the decomposition "$x1 = 5 \cdot x2$" into "$x3 = x2 \ll 2$; $x1 = x2 + x3$", and assume it is inside a loop with an injuring increment over $x2$ (i.e., $x2 = x2 + 1$). The reduction starts by lifting the shift out of the loop in an auxiliary variable xA, and inserts an update assignment $xA = xA + 4$ just before the increment.

[12] In other words, our algorithm naturally includes (and infers) R2$^\flat$, while R1 is calculated only when necessary, and after R2$^\flat$.

[13] Such instructions are considered constant as long as their destination register is never rewritten.

Then, we improve this first transformation by noticing that in most cases, the shift's intermediate result is only used to compute the addition. If applicable, we thus lift the addition too using an auxiliary variable xA', and apply the update (i.e. adding 5, as "$4 \cdot 1 \cdot x2 + x2 = 5 \cdot x2$") on xA'.

Compensatory updates assignments (i.e. additions) are always inserted in blocks containing an injuring operation on the candidate, but not necessarily in all of them. More specifically, an injured node must receive an update assignment either if it contains an occurrence of the candidate (whether in its entry or exit part), or if it has at least one successor not marked as an insertion point (for both the entry and exit parts) but identified as an update point. These blocks are characterized by the least solution of the below equation, which covers a whole basic-block:

$$\uparrow \texttt{UPDATE}(n) \triangleq \begin{array}{l} \lceil \texttt{COMP} \rceil(n) \vee \lfloor \texttt{COMP} \rfloor(n) \vee \\ \sum_{m \in succ(n)} \left(\begin{array}{l} \overline{\lceil \texttt{INSERT} \rceil(m)} \wedge \overline{\lfloor \texttt{INSERT} \rfloor(m)} \\ \wedge \texttt{UPDATE}(m) \end{array} \right) \end{array}$$

After having inserted and replaced candidates, LSR inserts update additions in every node satisfying both INJURED *and* UPDATE.

4.2 Generalizing LSR on Basic-Blocks

On the example of Figure 4, a multiplication (in fact, a shift) inside the loop is replaced by an addition. To keep the code correct, the multiplication is also inserted before the loop. In some complex cases (e.g. the nested loops of Knoop et al. [19, Fig.3]), such an insertion of the multiplication may itself need to be compensated by an addition. This is precisely what R1 seeks to avoid: not placing a multiplication too early, so that a supplementary addition is unnecessary. The applicability of R1 thus depends on the candidate kind: additive SR candidates are always preceded by a multiplication (otherwise they are selected as code motion candidates), and do not require R1. The latter is therefore only computed for multiplicative SR candidates.

Technically, the first refinement (R1) of Knoop et al. [19] computes a set of *critical* points from which there exists a path with no other occurrence of the candidate before the injuring operation. Then, *critical-insertion* points are both critical and marked as insertion (in the sense of R2$^\flat$), and represent places where the "naive" (without R1) LSR would place both a multiplication *and* an update assignment on the same path. To optimize this inefficiency, the authors define a predicate *substitution-critical* that encodes the set of *substitutes* (i.e. alternatives) of critical-insertion points. Intuitively, R1 simply delays each critical-insertion point until the first reachable, non-critical point.

4.2.1 New Data-Flow Equations for **R1**. SR additive candidates keep R2$^\flat$ IR-points from the first step; but the `state` of SRmul candidates is extended with results of R1. Our method to compute R1 on top of R2$^\flat$ leads to insertion points equivalent to the original R2. We adapted the original (backward) "critical" predicate below:
$\uparrow \texttt{CRIT}(n) \triangleq \overline{\texttt{COMP}(n)} \wedge (\overline{\texttt{TRANSP}(n)} \vee \sum_{m \in succ(n)} (\texttt{CRIT}(m)))$
to a basic-blocks based analysis by splitting it into:

$$\uparrow \begin{cases} \lceil \texttt{CRIT} \rceil(n) \triangleq \overline{\lceil \texttt{COMP} \rceil(n)} \wedge (\overline{\texttt{TRANSP}(n)} \vee \lfloor \texttt{CRIT} \rfloor(n)) \\ \lfloor \texttt{CRIT} \rfloor(n) \triangleq \overline{\lfloor \texttt{COMP} \rfloor(n)} \wedge \sum_{m \in succ(n)} (\lceil \texttt{CRIT} \rceil(m)) \end{cases}$$

Deducing the above equations is straightforward; the first step is to duplicate the original predicate in two variants with $\lceil$COMP$\rceil$

and ⌊COMP⌋. Since it must be solved backward (i.e. it depends on the successor relationship), the existential $\sum$ in the first equation is replaced with ⌊CRIT⌋. For the exit equation, we remove the transparency term (as it does not depend on basic-blocks parts, and is already present in the entry equation); finally, noticing that the successor of an exit part is obviously an entry part, the disjunction over successors is updated with the entry equation.

The bitwise "and" between the entry/exit variants of R2^b INSERT and CRIT gives us the entry/exit "critical-insertion" points noted ⌈CRITINS⌉/⌊CRITINS⌋ (i.e. CRITINS = INSERT ∧ CRIT). Those are needed to adapt the original "substitution-critical" forward equation below, in the same fashion as before.

$$\downarrow \texttt{SUBSTCRIT}(n) \triangleq \begin{array}{l} \texttt{CRITINS}(n) \vee \\ (\sum_{m \in pred(n)} \overline{\texttt{COMP}(m)} \wedge \texttt{SUBSTCRIT}(m)) \end{array}$$

which can be decomposed, from a reasoning symmetrical to that of the CRIT predicate, into:

$$\downarrow \begin{cases} \lceil \texttt{SUBSTCRIT} \rceil(n) \triangleq & \lceil \texttt{CRITINS} \rceil(n) \vee \\ & \sum_{m \in pred(n)} \left(\begin{array}{l} \overline{\lfloor \texttt{COMP} \rfloor(m)} \wedge \\ \lfloor \texttt{SUBSTCRIT} \rfloor(m)) \end{array} \right) \\ \lfloor \texttt{SUBSTCRIT} \rfloor(n) \triangleq & \lfloor \texttt{CRITINS} \rfloor(n) \vee \\ & (\overline{\lceil \texttt{COMP} \rceil(n)} \wedge \lceil \texttt{SUBSTCRIT} \rceil(n)) \end{cases}$$

4.2.2 Pushing Critical Insertion Points Forward. We now update R2^b insertion points based on R1, in order to obtain an INSERT predicate equivalent to the one of R2. Recall that replacement points are not impacted by R1.

Algorithm 2 R2 insertion points from R1 & R2^b

```
1:  procedure FIND_CRIT_TARGETS_REC(cand, pc)
2:      st ← cand.state
3:      VISIT(pc)
4:      for s ∈ succ(pc) do
5:          if ¬st.⌈CRIT⌉(s) ∧ st.⌈SUBSTCRIT⌉(s) then
6:              st.⌈INSERT⌉(s) ← true
7:          else if ¬st.⌊CRIT⌋(s) ∧ st.⌊SUBSTCRIT⌋(s) then
8:              st.⌊INSERT⌋(s) ← true
9:          else if ¬VISITED(s) then
10:             FIND_CRIT_TARGETS_REC(s)
11: procedure FIND_CRIT_TARGETS_GEN(cand, p_ins, p_ins_crit)
12:     RESET_VISITED_BLKS(void)
13:     for pc ∈ {pc_i | p_ins_crit(pc_i) = true} do
14:         p_ins(pc) ← false
15:         FIND_CRIT_TARGETS_REC(cand, pc)
```

The FIND_CRIT_TARGETS_GEN procedure of Algorithm 2 pushes forward (in the direction of the control-flow) insertion points for SR candidates. For each of them, we call the procedure if R1's ⌈CRITINS⌉ (resp. ⌊CRITINS⌋) is *not* full of zeros (i.e. not always false); with *p_ins* = ⌈INSERT⌉ (resp. ⌊INSERT⌋) and *p_ins_crit* = ⌈CRITINS⌉ (resp. ⌊CRITINS⌋). First, we set the insert predicate (which can be either the entry or exit one) to false for every block satisfying the given (entry or exit) critical-insertion predicate. Second, the FIND_CRIT_TARGETS_REC procedure replaces insertion points: it recurses over successors from the critical-insertion block, and stops when encountering an already visited block. For entry and exit parts, if a successor is not critical but substitution critical (lines 5 and 7 of Alg. 2), then its entry/exit INSERT predicate is set to true.

$$\begin{array}{lcl} \textit{type } C \texttt{ affine\_form} & ::= & \texttt{Aff\_term}(C, r, C \texttt{ affine\_form}) \\ & | & \texttt{Aff\_const}(C) \end{array}$$

Figure 10: Polymorphic affine forms

Our insertion points are now equivalent to the second refinement of Knoop et al. [19, §4.1].

4.3 Affine Forms Strength-Reduction

The third refinement of Knoop et al. [19] *accumulates* update assignments when the source includes multiple injuring operations (as illustrated in Ex. 4.1). Their solution is to first record program points where an accumulated update should be inserted, and second to define a function that calculates the accumulation effect. Nonetheless, this idea involves a prior detection of *extended basic-blocks* [19, footnote 15]. Mimicking this technique would be possible with our block-based LSR, even if it seems a bit heavy in our formally verified defensive framework. Moreover, this mechanism is subsumed by noticing that candidates can either multiply or add values, which amounts to manipulate affine forms[14]. We simply define addition and scalar multiplication of affine terms (forming a semimodule [12]), to accumulate "injuries" over induction variables, to reduce products between constants, and to factorize additions on the same variable (cf. the sequence in Ex. 4.1).

Hence, we improve R3 (but only for basic-blocks) with the affine forms of Figure 10, where C is the type for constants (affine forms are polymorphic on C, e.g. C might be `int64`), and r for registers. The oracle maintains a hash-table $(pc, r) \mapsto (C \texttt{ affine\_form})$, so we map (block ID, register) pairs to affine values. The detection phase applies operations over these forms as they occur, and the substitutions of §4.4 keep the table up-to-date with auxiliary variables. When inserting the update assignment, we invoke a function that takes a list of block IDs and the candidate's auxiliary destination register (fresh register) to retrieve the compensation amount that needs to be added.

4.4 Iterative Substitution of Auxiliary Variables

We mentioned in §3.1 that we process candidates in their topological order. The original LCM of Knoop et al. [18] does not specify any order and neither does the original LSR of Knoop et al. [19]. However, as illustrated in the example of §2.4 (and in Ex. 4.1), there are sometimes—mainly because of the instruction selection, but not always—sequences of instructions prone to be reduced. In that situation, the order of treatment is essential: the first instruction of the sequence must be moved (or reduced) first to detect a new opportunity when initiating the analysis on the following ones. Technically, after having detected and sorted candidates from the hash table as a list of (*ckey*, *cand*) pairs, our LCT repeats the four steps below for each pair. (i) *Update* the current candidate: if its original arguments (stored in the `orig_args` field of Fig. 8) were modified by previous substitutions (through the `updated_args` field), then its key of Figure 9 is modified with the substituted arguments[15].

[14] Our validator uses a similar representation, as visible in example 2.1.

[15] We replace the candidate in the hash table (and take the updated *ckey* in the sorted list). If a candidate with the new arguments already exists, the new one is merged with the old one (by unifying their `lhs` fields).

Furthermore, if the updated arguments do not contain any SR auxiliary variable, and if the candidate is of the SRadd type (Fig. 9), then we "downgrade" it to an LCM candidate (i.e. SR additive candidates must be preceded by a multiplication, see §4.2). (ii) *Initialize* predicates (with their default value) and *execute the local analysis*: the calculation of TRANSP/SR-TRANSP, ⌈COMP⌉/⌊COMP⌋, and INJURED (if applicable). (iii) *Data-flow analysis.* (iv) Eventually *rewrite the CFG* if the candidate must be inserted, replaced, or updated.

The original LCM replaces an assignment from a redundant expression by a "move" from its auxiliary variable. Simply applying this technique would prevent our propagation of its affine form, as expressed in (i) above. We thus propose a new refinement that replaces the early move instruction by direct substitutions with the auxiliary variable, so that the move is pushed forward as late as possible in the basic block, enabling the propagation of the affine form above this move.

Let pc_t and off_t be the target block ID and an offset inside this block (resp.) where we are going to replace the candidate. The function traverses the basic-block starting from the entry, and, depending on the current offset off_c:

- If $off_c < off_t$, simply continue and increment off_c;
- If $off_c = off_t$, ensure a match between the current instruction and the candidate to replace, replace it by a no-op (no-operation) instruction (rather than directly by a move), and continue[16]. The algorithm saves the original destination of the replaced instruction;
- If $off_c > off_t$ (meaning the candidate was already replaced by a no-op), there are two possible subcases:
  - A final case when either (i) the auxiliary variable or the original destination of the candidate is rewritten; (ii) the current instruction is another occurrence of the candidate or an injuring operation; (iii) we are reaching the end of the block.
    If so, we insert the move from the auxiliary variable and stop the substitution algorithm;
  - A recursive case otherwise, where we substitute the previously saved original destination in the updated_args field of the current candidate by its auxiliary variable.

5 INFERRING INVARIANTS FROM ANALYSES

Once the main loop of LCT—consisting of the four steps described in §4.4—terminates, each possible LCT optimization was applied on the CFG. Before generating invariants annotations, we perform a liveness analysis and a dead code elimination (DCE). This DCE is validated "for free", together with liveness information, and allows removing "dead" moves inserted by the forward substitution of §4.4.

Invariants annotations are inferred from both the liveness and the LCT analyses. This process is done for each pair $(ckey, cand)$ in the list L of candidates (the same list as in §4.4) with a *defined* auxiliary variable (i.e. not None) in $cand$.vaux (of Fig. 8). In other words, we iterate over the set C of candidates defined on line 2 of Algorithm 3. In fact, a defined auxiliary variable means that the candidate was moved or strength-reduced and so the validator will need invariants to ensure the transformation's correctness. Variable gm (stands for "gluemap") in Algorithm 3 is a structure containing both invariants mappings, gluing and history. The gluing invariants mapping in gm is already initialized with the above-mentioned liveness analysis results (so the "alive" sets at each node are filled).

Algorithm 3 Generation of invariants annotations

```
1:  function BUILD_INVARIANTS(entry, L, gm, constants)
2:      C ← {(ckey, cand) ∈ L | cand.vaux ≠ None}
3:      for (ckey, cand) ∈ C do
4:          st ← cand.state
5:          G ← ((st.⌈REPLACE⌉ ∧ ¬st.⌈INSERT⌉)∨
                 (st.⌊REPLACE⌋ ∧ ¬st.⌊INSERT⌋))
                 ∨(st.ISOL ∧ st.DELAY)
6:          if IS_TRAPPING(ckey) then
7:              G ← G ∧ st.⌈U-SAFE⌉
8:          const_prod ← IS_CONSTANT_PRODUCT(ckey)
9:          ok_reduced ← cand.was_reduced
10:         for pc ∈ CFG do
11:             if pc ≠ entry ∧ const_prod ∧ ok_reduced then
12:                 r_c ← CONSTANT_REG(ckey)
13:                 c_op, c_pc ← constants[r_c]
14:                 ok_fresh ← ¬IS_FRESH_VAR(r_c)
15:                 A ← LIVE_INPUTS(gm, pc)
16:                 if pc < c_pc ∧ ok_fresh ∧ (G(pc) ∨ r_c ∈ A) then
17:                     gm ← ADD_HI(gm, pc, r_c, c_op)
18:             if G(pc) then
19:                 op ← GET_CKEY_OP(ckey)
20:                 gm ← ADD_GI(gm, pc, cand.vaux, op)
21:     return gm
```

5.1 Preservation Points for Gluing Invariants

Given a pair $(ckey, cand)$, the vector of block IDs where a gluing invariant about the candidate must be preserved is named G (line 5 in Alg. 3). Preservation points depend on four predicates for non-trapping candidates, and five otherwise. G is efficiently calculated with bitwise operations on predicates.

There are two types of nodes in which we must insert a gluing invariant: one for each alternative of line 5. The *first* (in teal color) groups blocks where *cand* was replaced, but not inserted. In this case, the target simulation must retrieve the candidate's value from the input gluing invariant (recall the target gluing invariants' simulation: $\mathcal{G}_I$ is applied *before* executing $\mathcal{T}$). For instance, in Figure 5, a gluing invariant assignment of the square of x2 in x13 is needed for block 5, where the candidate is replaced (in entry) but not inserted. Conversely, a counter-example (where the alternative is false), arises in block 8: actually, an input gluing invariant would be *wrong* to remember the square, since it is not yet executed on the target side. However, this first alternative is not sufficient because the candidate's value must also be preserved in the gluing invariant if the auxiliary variable is live after (e.g. across loops). Thus, we define a *second* alternative (in orange) to insert an input gluing invariant on every node which is neither isolated nor delayed. Indeed, an isolated candidate is by definition (of INSERT) never used for insertion (ISOL(n) is true if an insertion at n would

[16] At this point, we also update the tables of affine values and constants by copying the previous mapping (if existing) to a new one bound to the auxiliary variable of the candidate.

be only used at n itself). Moreover, it must not be delayed: if *cand* is delayed at node n, we know that its potential insertion can only happen after n (further in the CFG). Still in the example of Figure 5, the loop block 7 satisfies these conditions for the square candidate (neither isolated nor delayed); thus, in the source side simulation of block 8, the square of `x2` will be defined when executing the output gluing invariant, as expected (recall the source gluing invariants' simulation: $\mathcal{G}_I$ is applied *after* executing $\mathcal{S}$).

The disjunction encoded by G suffices to obtain preservation points for non-trapping candidates, but is not strong enough for trapping ones (e.g. loads). Hence, we restrict G (line 7 of Alg. 3) by conjunction with the entry up-safety predicate (as for condition (ii) of potential replacement points in §3.2). In Figure 5, this stronger version of G holds on the "`ld[x1+0]`" candidate for blocks 2 to 8, thus allowing to insert the necessary invariants for `x12`.

A gluing invariant assignment of the candidate's operation (in *ckey*) into the auxiliary register *cand*.`vaux` is therefore inserted for every block $pc \in CFG$ such as $pc \in G$ (lines 18-20 of Alg. 3).

5.2 Saving Constants With History Invariants

We saw in §2.5 that if the multiplication to reduce was directly containing the constant value as an immediate, history invariants were unnecessary. In contrast, when the constant is stored in a register, history invariants are necessary to remember its value.

The IS_CONSTANT_PRODUCT(*ckey*) function (line 8 in Alg. 3) returns true if if the candidate is of type `SRmul`, and if its constant is in a register (by seeking in the constants' table of §4.1). Inserting an history invariant is relevant only if the multiplicative candidate was effectively strength-reduced: this is indicated by the *cand*.`was_reduced` boolean (defined in Fig. 8, and read at line 9 in Alg. 3). Furthermore, as the CFG entry must only include pure-liveness invariants (see §2.5); the condition of line 11 checks that the current block is not the entry-point[17], along with the two conditions defined above. Nevertheless, some additional checks are required before inserting an history invariant: (i) the constant must be defined in a previous block (if it is defined in the current block, no need for an history invariant); (ii) the constant must not be in an auxiliary variable (otherwise it will be handled by gluing invariants); and (iii) either the current block must be in G, or the constant's register live in the block (if these two conditions are false, there is no need to propagate the constant's value). The algorithm first gathers the constant register (line 12), and the constant operation and block of appearance (line 13) from the constant table (§4.1). The comparison $pc < c_{pc}$ then checks condition (i) above; and the negation of function IS_FRESH_VAR(r_c) (line 14) ensures (ii). The set A of live variables in the block (line 15) was already computed before the DCE pass; here, we simply retrieve this information from *gm*. Finally, line 16 (the disjunction corresponds to condition (iii)) verifies that the three requirements are satisfied. If so, the algorithm inserts an history invariant assignment of the constant operation c_{op} into its associated variable r_c, and add r_c in the "alive" set of history invariant at block pc (line 17).

Notice that since the alive set for gluing invariants was already filled by the liveness analysis, we only add information to the alive set for history invariants here. In the end, the oracle returns both the new BTL code and the "gluemap" to our certified validator.

[17] For gluing invariants, this was implicitly ensured by the formula of G.

6 EXPERIMENTAL EVALUATION

We only implemented LSR on RISC-V, and for 64-bits long integers (but LCM is working on all backends). In the future, we may port it to other architectures, and support a wider class of candidates. Nevertheless, our LSR takes advantage of a promotion[18] pass—also validated by our symbolic simulation test—that converts 32-bits instructions into 64-bits ones, hence producing more SR opportunities.

Measurements presented in this section were performed using benchmarks from four suites: (i) the LLVM test suite[19]; (ii) the MiBench [14] and (iii) TACLeBench [11] embedded systems oriented suites, and (iv) the PolyBench [28] computational suite.

A more detailed overview of our testing approach, on both performance and debugging aspects of oracles, is provided in [23].

6.1 Performance benchmarks

We measured experimentally the performance of the generated code on a SiFive U740 core (HiFive Unmatched), a dual-issue, in-order RISC-V processor. Abnormal results whose relative standard derivation was exceeding 2% over five runs were filtered. The performance gain of a compiler C w.r.t. a reference compiler R is computed by $gain(C) = ((time(R) - time(C))/time(C)) \times 100$, where $time(X)$ represents the execution time, measured in number of cycles, of a benchmark compiled with compiler X. In order to achieve the utmost level of optimization, *we integrated the CompCert's fork of Six et al. [32]* into BTL. We refer to this integration as "Base", which combines their superblock scheduler, CSE3 [24], and first iteration loop-unrolling.

We compared GCC-01 (11.3.0), our "Base" fork, and the latter with LCT activated (i.e. "Base"+LCT) to mainline CompCert (3.12). Globally, our "Base"+LCT version is 20.7% faster than mainline CompCert on LLVM tests, 14.7% on MiBench, and 22.83% on TACLeBench; results for the full PolyBench suite are at line "PolyBench/*" of Table 1. The sample results of Table 1 highlights the performance variability according to inputs. The poor result on MiBench/stringsearch is explained by unrolling, that dramatically increases the register pressure and the code size. Without unrolling, the "Base" score on this benchmark drops to 0.43%, whereas "Base"+LCT yields a gain of 39.4%. Both the oracle and the validator scale well on every benchmark, and also on randomly generated programs from Csmith, Yarpgen, or CCG[20]. Finally, our algorithm is able to close the gap with GCC on a non-negligible set of benchmarks.

6.2 Translation Validation Time Measurements

To ensure that our validator was scaling well even on large applications, we instrumented the OCaml code generated from Coq to time each symbolic execution. In practice, after testing it over every benchmark from our four suites, the worst validation time was of approximatly 4 seconds. We observe an almost perfect linear

[18] The underlying oracle is out-of-scope of this paper.

[19] https://github.com/lac-dcc/Benchmarks

[20] See https://github.com/csmith-project/csmith, https://github.com/intel/yarpgen, and https://github.com/Mrktn/ccg.

Table 1: GCC, our fork Base=(scheduling + CSE3 + unroll single), and our fork Base+LCT versus mainline CompCert on RISC-V— Higher is better

Setup	GCC-O1	Base	**Base + 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%

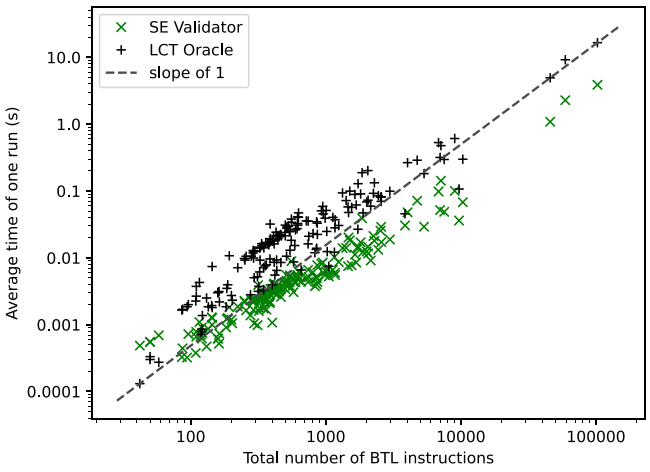

Figure 11: Oracle and Validator times w.r.t. the number of instructions (logarithmic scale)

correlation between the validator's and the LCT oracle's execution times (near 99%). Furthermore, if we bound using a threshold (for our experiment, we set it to 64) the number of LCT's candidates, we see that both the validator and the oracle are linear in the total number of instructions per BTL function. These results are shown in Figure 11. On average, and keeping the 64 candidates threshold, the validator is even slightly faster than the oracle for a given benchmark size.

7 RELATED WORK AND CONCLUSION

Strength-reduction designates various transformations, from replacing single instructions to linear-function test replacement. The only form of SR in the official CompCert is a form of peephole[21], divided among instruction selection and constant propagation. Modern, untrusted compilers rather implement straight-line SR (SLSR), a more powerful transformation targeting code sequences with arithmetic statements[22], that simplifies complex sequences unhandled by loop SR algorithms. We ported the expansion mechanism recently proposed by Six et al. [32] to BTL, and it could be (as a future work) extended to perform SLSR as well.

The loop nests SR algorithms in GCC & LLVM are SSA based[23], and might be very difficult to adapt in a formally verified context. They can reduce induction variables, but also perform linear-function test replacement (i.e. completely eliminate the original induction variable). Supporting similar algorithms would require extending BTL with partial SSA forms (for instance by encoding phi-nodes with explicit parallel moves on joining edges [2]). Alternatively, one might want to extend the SSA validators of Demange [7], Demange and Fernandez de Retana [8], Demange et al. [9], to support finer invariants (in addition to strong SSA invariants). Furthermore, both GCC and LLVM use a scalar evolution (SCEV) analysis, an efficient technique to find induction variables in specific code regions (e.g. loops). Proving correct such an analysis would nonetheless be interesting, knowing that it is subject to implementation bugs [36, §3.7, LLVM Bug #4]. Among state-of-the-art methods for loop SR, there are two main axes (as documented by Cooper et al. [6, §2]): methods working "a single loop at a time", seeking for loop-induction variables [5] (e.g. SSA based techniques of GCC & LLVM); and data-flow approaches [25] (e.g. LSR), which do not require control-flow analyses, and are mostly inspired by code motion & PRE.

Tristan and Leroy [34] proposed the first formally verified LCM optimization, and experimented it on the top of CompCert. They used a translation validator replaying some data-flow analyses: the advantage of their approach is that it does not need hints, because it redoes the necessary calculations from scratch. However, their validator is *a priori* more costly than ours. In contrast with us, their LCM is also able to anticipate trapping instructions thanks to an anticipability (also called inevitability) calculus performed during the validation. On the other hand, they do not have unrolling mechanisms like ours that increase LCM's opportunities.

Unfortunately, their work was never integrated in the official CompCert (for reasons unknown to us). Moreover, the code available online[24] does not include any information about compatible CompCert versions, nor about how to compile, integrate, or even run it. As it dates from 2009, their code is surely based on a CompCert version older than ten years (e.g. before the support of 64-bits architectures). Thus, we have not experimentally compared our LCM with theirs, and the related paper only mentions results about compilation/validation time.

We implemented an enhanced version of the LCM & LSR of Knoop et al. [18, 19] within CompCert, validated by our formally verified general purpose framework. Our adaptation of the original

[21] Replacing an instruction sequence by a more efficient pattern.

[22] In LLVM: https://github.com/llvm/llvm-project/blob/main/llvm/lib/Transforms/Scalar/StraightLineStrengthReduce.cpp;
In GCC: https://github.com/gcc-mirror/gcc/blob/master/gcc/gimple-ssa-strength-reduction.cc.

[23] LoopSR in LLVM: https://github.com/llvm/llvm-project/blob/main/llvm/lib/Transforms/Scalar/LoopStrengthReduce.cpp; IVOPTS in GCC: https://github.com/gcc-mirror/gcc/blob/master/gcc/tree-ssa-loop-ivopts.cc.

[24] https://github.com/jtristan/CompCert-Extensions

data-flow analyses facilitates the defensive validation by symbolic execution of those algorithms.

REFERENCES

[1] Jan Andersson. 2020. Development of a NOEL-V RISC-V SoC Targeting Space Applications. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. 66–67. https://doi.org/10.1109/DSN-W50199.2020.00020

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

[3] 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

[4] 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

[5] John Cocke and Ken Kennedy. 1977. An algorithm for reduction of operator strength. *Commun. ACM* 20, 11 (Nov. 1977), 850–856. https://doi.org/10.1145/359863.359888

[6] Keith D. Cooper, L. Taylor Simpson, and Christopher A. Vick. 2001. Operator strength reduction. *ACM Transactions on Programming Languages and Systems* 23, 5 (Sept. 2001), 603–625. https://doi.org/10.1145/504709.504710

[7] 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.

[8] 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

[9] 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

[10] Stefano Di Mascio, Alessandra Menicucci, Eberhard Gill, Gianluca Furano, and Claudio Monteleone. 2019. Leveraging the Openness and Modularity of RISC-V in Space. 16 (2019), 1–19. https://doi.org/10.2514/1.I010735

[11] 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.

[12] Jonathan S. Golan. 1999. Semimodules over Semirings. In *Semirings and their Applications*. Springer Netherlands, Dordrecht, 149–161. https://doi.org/10.1007/978-94-015-9333-5_14

[13] Léo Gourdin, Benjamin Bonneau, Sylvain Boulmé, David Monniaux, and Alexandre Bérard. 2023. Formally Verifying Optimizations with Block Simulations. (May 2023). https://hal.science/hal-04102940 preprint.

[14] 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

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

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

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

[18] 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

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

[20] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009). HAL:inria-00415861

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

[22] Tao Lu. 2021. A Survey on RISC-V Security: Hardware and Architecture. http://arxiv.org/abs/2107.04175 arXiv:2107.04175 [cs].

[23] 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*, Vol. to appear. Springer. https://hal.science/hal-04096390

[24] 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

[25] E. Morel and C. Renvoise. 1979. Global optimization by suppression of partial redundancies. *Commun. ACM* 22, 2 (Feb. 1979), 96–103. https://doi.org/10.1145/359060.359069

[26] George C. Necula. 2000. Translation validation for an optimizing compiler. 83–94. https://doi.org/10.1145/349299.349314

[27] A. Pnueli, M. Siegel, and E. Singerman. 1998. Translation validation. In *Tools and Algorithms for the Construction and Analysis of Systems*, Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, and Bernhard Steffen (Eds.). Vol. 1384. Springer Berlin Heidelberg, Berlin, Heidelberg, 151–166. https://doi.org/10.1007/BFb0054170 Series Title: Lecture Notes in Computer Science.

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

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

[30] 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

[31] 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

[32] 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

[33] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. 2016. Toward understanding compiler bugs in GCC and LLVM. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, Saarbrücken Germany, 294–305. https://doi.org/10.1145/2931037.2931074

[34] Jean-Baptiste Tristan and Xavier Leroy. 2009. Verified Validation of Lazy Code Motion. 316–326. http://gallium.inria.fr/~xleroy/publi/validation-LCM.pdf

[35] Andrew Waterman, Yunsup Lee, David Patterson, Krste Asanovic, Volume I User level Isa, Andrew Waterman, Yunsup Lee, and David Patterson. 2014. The RISC-V instruction set manual. *Volume I: User-Level ISA', version* 2 (2014).

[36] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. 283–294. https://doi.org/10.1145/1993498.1993532

[37] 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 (April 2021), 110884. https://doi.org/10.1016/j.jss.2020.110884