CRELLVM: Verified Credible Compilation for LLVM

Jeehoon Kang* Yoonseung Kim* Youngju Song*
{jeehoon.kang, yoonseung.kim, youngju.song}@sf.snu.ac.kr
Seoul National University, Korea

Juneyoung Lee Sanghoon Park
Mark Dongyeon Shin Yonghyun Kim
{juneyoung.lee, sanghoon.park}@sf.snu.ac.kr
{dongyeon.shin, yonghyun.kim}@sf.snu.ac.kr
Seoul National University, Korea

Sungkeun Cho
skcho@ropas.snu.ac.kr
Seoul National University, Korea

Joonwon Choi
joonwonc@mit.edu
MIT CSAIL, USA

Chung-Kil Hur†
gil.hur@sf.snu.ac.kr
Kwangkeun Yi
kwang@ropas.snu.ac.kr
Seoul National University, Korea

Abstract

Production compilers such as GCC and LLVM are large complex software systems, for which achieving a high level of reliability is hard. Although testing is an effective method for finding bugs, it alone cannot guarantee a high level of reliability. To provide a higher level of reliability, many approaches that examine compilers' internal logics have been proposed. However, none of them have been successfully applied to major optimizations of production compilers.

This paper presents CRELLVM: a verified credible compilation framework for LLVM, which can be used as a systematic way of providing a high level of reliability for major optimizations in LLVM. Specifically, we augment an LLVM optimizer to generate translation results together with their correctness proofs, which can then be checked by a proof checker formally verified in Coq. As case studies, we applied our approach to two major optimizations of LLVM: register promotion (`mem2reg`) and global value numbering (`gvn`), having found four new miscompilation bugs (two in each).

CCS Concepts • **Theory of computation → Hoare logic**; • **Software and its engineering → Compilers**; **Formal software verification**;

Keywords LLVM, Coq, credible compilation, translation validation, compiler verification, relational Hoare logic

* The first three authors contributed equally to this work and are listed alphabetically.
† Hur is the corresponding author.

PLDI'18, June 18–22, 2018, Philadelphia, PA, USA

ACM ISBN 978-1-4503-5698-5/18/06…$15.00
https://doi.org/10.1145/3192366.3192377

ACM Reference Format:
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 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18).* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3192366.3192377

1 Introduction

Production compilers such as GCC and LLVM are large complex software systems, for which achieving a high level of reliability is hard. Their complexity comes in two fold. First, to generate efficient target code, they perform various complex optimizations. Second, to consume less time and memory during compilation, they are usually written in C/C++ using sophisticated data structures. Due to such complexity, it is hard to make mainstream compilers very reliable.

Although testing is an effective method for finding bugs, that alone hardly guarantees a high level of reliability. Recent random testing tools such as CSmith [53] and EMI [24] have shown their effectiveness by finding hundreds of bugs in GCC and LLVM. However, they missed bugs in the `gvn` and `mem2reg` passes of LLVM, which we discovered later (see §1.2 for details), since they treat compilers as black boxes without examining their internal logics.

In order to provide a higher level of reliability, many approaches that examine compilers' internal logics have been proposed, none of which, however, have been successfully applied to major optimizations of production compilers. For example, while compiler verification techniques have been applied to compilers such as CompCert [26] to guarantee their formal correctness, this approach is not readily applicable to production compilers since it requires compilers to be written in the language of a proof assistant such as Coq. As another example, Alive [30] is a domain-specific language (DSL) in which one can manually write a compiler's optimization logic and automatically verify its correctness or else generate a counterexample. Though this approach has been successfully applied to LLVM, its application is limited to peephole optimizations because it is hard to faithfully translate the implementation of complex optimizations into Alive and, more importantly, Alive does not support cyclic control flows such as loop. As the last example, the credible compilation [16, 33, 34, 44] and verified translation validation [14, 19, 43, 50–52] approaches augment compilers to generate translation results together with their correctness proofs, which can then be checked by a (verified) proof checker. Since a correctness proof is generated and checked at each compilation time, it provides a formal correctness guarantee for the particular translation or else finds a bug (either in the compiler code or in the proof-generation code). However, there has been only a preliminary attempt to apply these approaches to production compilers so far. (See §9 for detailed comparison.)

This paper presents CRELLVM: a verified credible-compilation framework for LLVM, which can be used as a systematic way of providing a high level of reliability for major optimizations in LLVM. Specifically:

1. We design and develop a logic and its proof checker for reasoning about LLVM optimizations, called Extensible Relational Hoare Logic (ERHL), in the proof assistant Coq. This logic's novelty lies in its representation of relational predicates as mostly unary predicates (see §2.2 for details).
2. We fully verify a semantics-preservation result for our proof checker in the style of CompCert using the Coq formalization of LLVM IR (Intermediate Representation) from the VELLVM project [55].
3. As case studies, we wrote proof-generation codes (213 and 440 SLOC[1] in C++) for two major optimizations: register promotion in the `mem2reg` pass and global value numbering (GVN) with partial-redundancy elimination (PRE) in the `gvn` pass. Then we performed validation of the two optimizations for standard benchmarks, five large open-source projects and test files randomly generated by CSmith.
4. As a result, we found four new miscompilation bugs (two in each optimization). It is notable that all the four bugs had been hidden for 7-8 years until we found them.

1.1 Overview of CRELLVM

Framework The framework of CRELLVM works as follows. First, as shown in Fig. 1, we separate the compilation and validation phases. For compilation, as depicted in the left side of Fig. 1, we use the original optimizer to translate the source IR code `src.ll` to the target IR code `tgt.ll`. After the compilation, we can conduct validation, as depicted in the right side of Fig. 1. For this, we first run the optimizer extended with a proof-generation code that produces the target `tgt'.ll` together with the proof `Proof`. Then the proof checker validates `Proof` to see whether `src.ll` is correctly translated to `tgt'.ll`. If the validation fails, we can see a logical reason for the failure, with which we can find a bug either in the compiler or in the proof-generation code. If the validation succeeds, we finally compare `tgt.ll` and `tgt'.ll` using the LLVM IR comparison tool `llvm-diff`.

[1] SLOC stands for significant lines of code *i.e.*, ignoring spaces and comments.

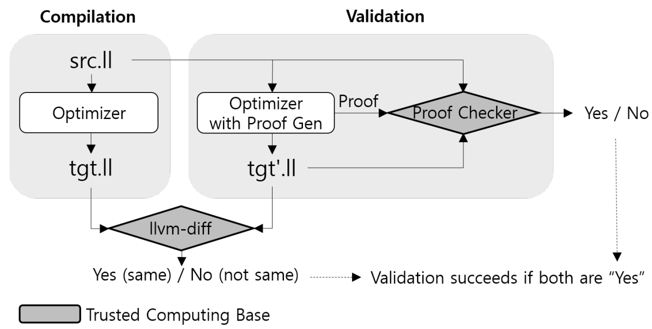


Figure 1. The CRELLVM Framework

There are two points to note about the framework. First, `llvm-diff` essentially performs alpha-equivalence checking, which is necessary because while `tgt.ll` may have unnamed IR registers, `tgt'.ll` has explicit names for all registers for proof-generation purposes. Second, since we just add proof-generation code without modifying existing compiler code except for giving names to unnamed registers, the original and proof-generating compilers are expected to generate alpha-equivalent programs, which is always checked using `llvm-diff` as described above. Therefore, programmers can use the original compiler in regular usage and then run the proof-generating one on occasion to check correctness because the former is much faster than the latter. On the other hand, compiler developers can use the latter for testing on regular basis to find bugs.

ERHL and Proof Checker For validation in CRELLVM, we develop ERHL, which is a variant of relational Hoare logic [16] specialized for LLVM IR. The logic and its proof checker is *extensible* because (*i*) the logic can be extended with any custom inference rules and (*ii*) the proof checker can be extended with any custom automation functions that try to fill in the gaps in incomplete proofs by automatically finding appropriate inference rules, like the `auto` tactic in Coq.

Verification of Proof Checker In the CRELLVM framework, the TCB (Trusted Computing Base) includes only the proof checker, the equality checker (`llvm-diff`) and custom inference rules. In particular, the proof-generation code in the compiler is not a part of the TCB because any incorrect proof would be invalidated by the proof checker.

We further remove the proof checker and inference rules from the TCB by implementing and verifying them in Coq. Though we currently use the (unverified) standard `llvm-diff` tool for comparing IR programs, it would also be possible to implement and verify it in Coq.

Note that verification of the proof checker and inference rules matters in practice. First, we found various corner-case bugs in our proof checker during its verification. Second, we also found one of our two mem2reg bugs [9] during the verification of inference rules. See the example below.

```
p := alloca(); r := *p
foo(r)                          ⇝ foo(1 / ((int)G - (int)G))
*p := 1 / ((int)G - (int)G)
```

Here G is the constant address of a global variable.

To see why this translation is incorrect, suppose that the function `foo(r)` ignores `r` and repeatedly prints out 0 without returning to the caller. Then division-by-zero never happens in the source program, while it does in the target. The problem here is that the mem2reg pass assumes that constant expressions never raise any undefined behavior such as division-by-zero, which is not true since `1 / ((int)G - (int)G)` forms a valid constant expression in LLVM. Following the logic of mem2reg, we also added such a custom inference rule, which we found unsound during the verification of the rule.

It is important to note that all the programs in this paper represent LLVM IR programs and we just use C syntax to help with understanding. For example, the source program in the above transformation is undefined as a C program but well-defined as an IR program. Thus, the transformation is only unsound as an IR-to-IR transformation. The LLVM community considers such an IR-to-IR miscompilation as a definite bug even when it does not cause any C-to-Assembly miscompilation since it can potentially cause an end-to-end miscompilation for other source languages such as Swift and Rust.

Results We wrote proof-generation codes for register promotion in the mem2reg pass and for GVN-PRE in the gvn pass; and also partly for loop-invariant code motion in the licm pass, and 139 micro-optimizations in the instcombine pass in order to demonstrate the generality of ERHL. We then conducted validation of the optimizations for the SPEC CINT2006 C Benchmarks [15], LLVM nightly test suite, and five open-source projects: sendmail, emacs, python, gimp, and ghostscript, in total 5.3 million LOC in C. As a result, we found four new miscompilation bugs.

We present the details of mem2reg validation in §3 and gvn validation in [1, §C].

1.2 Advantages of Crellvm over Testing

Crellvm checks whether optimizations are performed by *correct reasoning*, while testing simply checks *results* of the test programs. This can make a difference as follows.

First, an optimization performed by incorrect reasoning may still be correct for most programs including all the test programs. In this case, testing cannot uncover the bug, while Crellvm can because it checks the underlying reasoning. For example, we found our first mem2reg bug [5] in this situation.

Specifically, the following optimization shows the bug.

```
p := alloca()
loop {                                  loop {
    r := *p; foo(r); *p := 42  ⇝           foo(undef)
}                                       }
```

This translation is incorrect because only in the first iteration of the loop is `r` `undef`[2]; in the remaining iterations `r` is 42 according to the semantics of LLVM. The mem2reg pass performs this due to faulty reasoning.

However, this faulty reasoning is often not visible in the final compiled program. The reason is that, since the input to `foo` is sometimes undefined, for `foo` to be well behaved it often ignores its input `r` (*e.g.,* by using an operation like `r & 0x0`). Thus this transformation is actually correct in such a program since the value of `r` is never used in the program. Indeed, the SPEC benchmark that provoked this faulty reasoning behaved this way, and so the faulty reasoning never led to a faulty program, which is why the bug had been hidden for such a long time.

The fact that the faulty reasoning was inconsequential in this case does not mean the bug is unimportant. As we said before, the LLVM community cares about such an IR-to-IR miscompilation and immediately fixed the bug after we reported it. Moreover, visible miscompilations due to the bug could happen in a realistic situation (see [1, §B] for a concrete example).

Second, a potential flaw introduced by miscompilation may not be exploited by the current compiler and silently disappear during the compilation. Also in this case, Crellvm can detect the bug because it checks the underlying reasoning. For example, we found the two gvn bugs [6, 7] in this situation, which had not been found for 8 years. Note that the two bugs are caused by the same reason but we counted them as two because they appear in two separate places.

Specifically, the following optimization shows the bug.

```
q1 := (p + 10) inbounds       q1 := (p + 10) inbounds
q2 := (p + 10)           ⇝
bar(q1, q2)                   bar(q1, q1)
```

In the source program, `(p + 10) inbounds`[3] is defined to be `undef`[4] when the index 10 is out of the bounds of `p`, while `(p + 10)` is always defined to be the computed address. Thus replacing `q2` with `q1` introduces more undefinedness, which is incorrect because it can be potentially exploited by subsequent optimizations. However, so far the LLVM compiler has not exploited such undefinedness, thereby causing no observable misbehaviors. Indeed this miscompilation happened many times during validation of the standard benchmarks but testing has failed to detect it.

[2]Since `*p` is uninitialized, it contains `undef`, which is a special value representing undefinedness

[3]This denotes the GetElementPtr (GEP) operation.

[4]Technically, it is defined to be `poison` but the difference does not matter here.

```
      { MD(∅) }
10 :  x := add a 1          ⇝   x := add a 1
      {x_src = add a_src 1       x_tgt = add a_tgt 1     MD(∅) }
      { x_src = add a_src 1                              MD(∅) }
            ⋮                         ⋮
      { x_src = add a_src 1                              MD(∅) }
20 :  y := add x 2          ⇝   y := add a 3
      { x_src = add a_src 1
        y_src = add x_src 2      y_tgt = add a_tgt 3     MD({y}) }
                        ⇓ assoc_add(x_src, y_src, a_src, 1, 2)
      { x_src = add a_src 1
        y_src = add x_src 2      y_tgt = add a_tgt 3     MD({y}) }
        y_src = add a_src 3
                        ⇓ reduce_maydiff(y)
      { x_src = add a_src 1
        y_src = add x_src 2      y_tgt = add a_tgt 3     MD(∅) }
        y_src = add a_src 3
      { MD(∅) }
21 :  foo(y)                ⇝   foo(y)
      { MD(∅) }
```

Figure 2. Validation of an assoc-add translation in ERHL

2 Overview

In this section, we give a more detailed overview of how CRELLVM works using the assoc-add optimization of the instcombine pass as a motivating example.

2.1 Translation Example

We first give an example translation of the assoc-add optimization, which is shown in the shaded part of Fig. 2. Here `y := add x 2` is replaced by `y := add a 3` at line 20. This translation can be beneficial because after it, the register x may no longer be used and thus `x := add a 1` at line 10 may be eliminated later. This translation is also sound because (*i*) the *assertion* "`x = add a 1`" holds throughout lines 10-20, since the registers a and x are not redefined between line 10 and 20 thanks to the Static Single Assignment (SSA) property [18][5]; and (*ii*) from this, we can *infer* that `add x 2` = `add (add a 1) 2` = `add a 3` holds at line 20.

2.2 Proof Validation

We now construct a proof for the assoc-add translation example and validate it in ERHL.

ERHL Proof A formal proof of the translation is given in the box of Fig. 2. Specifically, the proof consists of a set of assertions and a list of inference rules at each line. For example, at line 20, the set of assertions is $\boxed{\{\,\mathrm{MD}(\emptyset)\,\}}$ and the list of inference rules is ($\boxed{\texttt{assoc\_add}(\mathtt{x}_{src}, \mathtt{y}_{src}, \mathtt{a}_{src}, 1, 2)}$, $\boxed{\texttt{reduce\_maydiff(y)}}$).

This ERHL proof captures the assertion and the inference step of the intuitive reasoning above. First, the assertion MD(∅) at every line states that every register contains the same value in the source and target program states. Second, the additional assertion $\mathtt{x}_{src} = \mathtt{add}\ \mathtt{a}_{src}\ 1$ between line 10 and line 20 states that in the source state, the value of the register x is equal to the result of `add a 1`. Finally, the inference rules $\texttt{assoc\_add}(\mathtt{x}_{src}, \mathtt{y}_{src}, \mathtt{a}_{src}, 1, 2)$ and `reduce_maydiff(y)` at line 20 are those that need to be applied for validation at line 20. The details of the rules will be given later when we discuss the validation process.

ERHL Assertions Before we proceed to the validation of the proof, we discuss ERHL assertions in more details. An ERHL assertion is a triple (S, T, M), where S is a set of assertions that should hold for the source state; T is for the target state; and M is an assertion relating the source and target states.

First, the source and target assertions, S and T, can contain various forms of predicates. For example, $\mathtt{x}_{src} = \mathtt{add}\ \mathtt{a}_{src}\ 1$ is a source assertion and $\mathtt{x}_{tgt} = \mathtt{add}\ \mathtt{a}_{tgt}\ 3$ is a target assertion. Here and henceforth, x_{src} and x_{tgt} represent the values of the register x in the source and target states, respectively. Though we only use the equality predicate for assoc-add, we will introduce various other predicates later. It is important to note that we do not allow arbitrary assertions relating the source and target states such as $x_{src} = y_{tgt} + 1$.

Second, the relational assertion M is a set of registers, called the *maydiff set*, that may contain different values in the source and target states. In other words, all the registers not in M should have the same value in the source and target states, which we denote by MD(M):

$$\mathrm{MD}(M) \iff \forall x \notin M.\ x_{src} = x_{tgt}\ .$$

Note that the maydiff set is the only form of relational assertion relating the source and target states.

Finally, every ERHL assertion implicitly requires the public parts of the source and target memories to be equivalent. More precisely, we use the CompCert-style memory-injection relation [28]. Later we introduce predicates that allow private memory allocations that do not belong to the public part of memory (see §3.2).

The main novelty of ERHL assertions is that we can use the standard algorithm of (unary) Hoare logic to compute post relational assertions, because ERHL assertions are mainly unary (*i.e.,* only for the source state, or for the target state, not relating them) except for the maydiff set. This unary nature greatly simplifies the ERHL proof checker and its correctness proof. Though mainly unary, ERHL assertions can indirectly encode general forms of relational assertions (see §3.2 for details).

[5] The SSA property says that for every used register x, there is statically (*i.e.,* syntactically) exactly one instruction that defines x (*i.e.,* assigning a value to x), which moreover comes before every use of x.

Proof Validation The gray text in Fig. 2 shows the validation process performed by the ERHL proof checker, which proceeds as follows.

First, the proof checker checks that the initial assertion holds for all possible initial states. It accepts the initial assertion $\{\,\mathrm{MD}(\emptyset)\,\}$ in Fig. 2 since the source and target states are initially equivalent.

Second, the proof checker checks whether the Hoare triple $\{P\}\ I_{src} \sim I_{tgt}\ \{Q\}$ at each line is valid. This means that the assertion Q after the line holds for all program states resulted by executing the source and target instructions I_{src} and I_{tgt} at the line under any program states satisfying the assertion P before the line. In Fig. 2, we only explain validations at lines 10 and 20 in detail because the others are trivial.

At line 10, the proof checker first computes a strong post-assertion, $\{\,\mathsf{x}_{src} = \mathsf{add}\ \mathsf{a}_{src}\ 1, \mathsf{x}_{tgt} = \mathsf{add}\ \mathsf{a}_{tgt}\ 1, \mathrm{MD}(\emptyset)\,\}$, using our post-assertion computation algorithm. Here, the algorithm simply adds the equality predicates corresponding to the executed instructions. Then, the assertion after line 10, $\boxed{\{\,\mathsf{x}_{src} = \mathsf{add}\ \mathsf{a}_{src}\ 1, \mathrm{MD}(\emptyset)\,\}}$, follows from the computed strong post-assertion by a simple inclusion check.

At line 20, the checker also computes a strong post-assertion, $\{\,\mathsf{x}_{src} = \mathsf{add}\ \mathsf{a}_{src}\ 1, \mathsf{y}_{src} = \mathsf{add}\ \mathsf{x}_{src}\ 2, \mathsf{y}_{tgt} = \mathsf{add}\ \mathsf{a}_{tgt}\ 3, \mathrm{MD}(\mathsf{y})\,\}$. Here, the post-assertion computation adds the equality predicates corresponding to the executed instructions and also adds the register y to the maydiff set because the executed source and target instructions are not identical. Then, the proof checker applies the inference rules given by the proof. The rule $\mathsf{assoc\_add}(\mathsf{x}_{src}, \mathsf{y}_{src}, \mathsf{a}_{src}, 1, 2)$ derives $\mathsf{y}_{src} = \mathsf{add}\ \mathsf{a}_{src}\ 3$ from $\mathsf{x}_{src} = \mathsf{add}\ \mathsf{a}_{src}\ 1$ and $\mathsf{y}_{src} = \mathsf{add}\ \mathsf{x}_{src}\ 2$ by associativity:

$$\frac{x = \mathsf{add}\ a\ C_1 \qquad y = \mathsf{add}\ x\ C_2 \qquad C = C_1 + C_2}{\mathsf{add}\ \{\,y = \mathsf{add}\ a\ C\,\}}\ (\mathsf{assoc\_add}(x, y, a, C_1, C_2))$$

The rule reduce_maydiff(y) removes the register y from the maydiff set because $\mathsf{y}_{src} = \mathsf{add}\ \mathsf{a}_{src}\ 3$, $\mathsf{y}_{tgt} = \mathsf{add}\ \mathsf{a}_{tgt}\ 3$ and a is not in the maydiff set:

$$\frac{y_{src} = e_{src} \qquad e_{tgt} = y_{tgt} \qquad \text{no registers in } e \text{ are in the maydiff set}}{\text{remove } y \text{ from the maydiff set}}\ (\mathsf{reduce\_maydiff}(y, e))$$

Then, the assertion after line 20, $\{\,\mathrm{MD}(\emptyset)\,\}$, easily follows by a simple inclusion check.

Finally, the proof checker checks whether the same observable events (*i.e.,* the same sequence of system calls) are produced at each line. It is the case in Fig. 2 because at line 20, no observable events are produced; and at the other lines, the source and target instructions are identical and the maydiff sets are empty implying that the source and target states are equivalent. In particular, at line 21, the proof checker explicitly checks that the same value is passed to the function foo because the function may produce observable events.

Algorithm 1 AssocAdd(F: Function)

A1: **for** l_2: y := add (reg x) (const C_2) **in** F **do**
A2: **if** FindDef(F, x) **is** l_1: x := add (reg a) (const C_1) **then**
A3: C := Simplify(add C_1 C_2)
A4: ReplaceAt(F, l_2, y := add (reg a) (const C))
A5: $\boxed{\mathrm{Assn}(x_{src} = \mathsf{add}\ a_{src}\ C_1, l_1, l_2)}$
A6: $\boxed{\mathrm{Inf}(\mathsf{assoc\_add}(x_{src}, y_{src}, a_{src}, C_1, C_2), l_2)}$
A7: **end if**
A8: **end for**
A9: $\boxed{\mathrm{Auto}(\mathsf{reduce\_maydiff})}$

2.3 Proof Generation

Now we explain how we generate proofs for assoc-add.

Algorithm Algorithm 1 shows the assoc-add optimization algorithm implemented in LLVM's instcombine pass, which is presented in a rather functional style for presentation purposes. Specifically, AssocAdd(F) optimizes each function definition F as follows (ignore the $\boxed{\text{boxes}}$ for now, which are the proof-generation code).

[Line A1] Find an instruction of the form l_2: y := add x C_2 with C_2 constant. In Fig. 2, 20: y := add x 2 can be picked. **[Line A2]** Check if x is defined by an instruction of the form l_1: x := add a C_1 with C_1 constant. Here, FindDef(F, x) finds the instruction that defines the register x.[6] In Fig. 2, 10: x := add a 1 is picked. **[Lines A3-A4]** If it is the case, compute the constant $C = C_1 + C_2$ and replace the instruction at l_2 with y := add a C. In Fig. 2, the instruction at line 20 is replaced by y := add a 3.

Proof Generation Once we understand the assoc-add optimization algorithm, it is quite straightforward to write the proof-generation code given in the $\boxed{\text{boxes}}$ of Algorithm 1.

[Line A5] Add the assertion $x_{src} = \mathsf{add}\ a_{src}\ C_1$ at every line between l_1 and l_2. In Fig. 2, the assertion $\mathsf{x}_{src} = \mathsf{add}\ \mathsf{a}_{src}\ 1$ is added at every line between 10 and 20. **[Line A6]** Add the inference rule $\mathsf{assoc\_add}(x_{src}, y_{src}, a_{src}, C_1, C_2)$ at line l_2. In Fig. 2, the rule $\mathsf{assoc\_add}(\mathsf{x}_{src}, \mathsf{y}_{src}, \mathsf{a}_{src}, 1, 2)$ is added at line 20. **[Line A9]** Enable the custom automation function named reduce_maydiff, which tries to find and insert appropriate reduce_maydiff rules when necessary. In Fig. 2, it figures out that reduce_maydiff(y) is needed at line 20.

Automation An automation function works as follows. When it remains to prove Q implies Q', the designated automation function examines the assertions Q and Q' and tries to find a sequence of inference rules that derives Q' from Q. For example, at line 20 in Fig. 2, after applying the assoc_add rule it remains to prove $Q = \{\,\mathsf{x}_{src} = \mathsf{add}\ \mathsf{a}_{src}\ 1, \mathsf{y}_{src} = \mathsf{add}\ \mathsf{x}_{src}\ 2, \mathsf{y}_{src} = \mathsf{add}\ \mathsf{a}_{src}\ 3, \mathsf{y}_{tgt} = \mathsf{add}\ \mathsf{a}_{tgt}\ 3, \mathrm{MD}(\mathsf{y})\,\}$ implies $Q' = \{\,\mathrm{MD}(\emptyset)\,\}$, from which the automation function finds the inference rule $\{\,\mathsf{reduce\_maydiff}(\mathsf{y})\,\}$.

[6] The instruction that defines x is unique thanks to the SSA property.

Automation functions can greatly simplify proof generation in certain cases. A good example is transitivity reasoning because it is much harder at proof generation time than at validation time. For instance, given a goal $x = y$, to prove it by transitivity, we have to figure out intermediate equations (*e.g.,* $x = a$, $a = b$, $b = y$). For this, at proof generation time, we have to write a code that (sometimes recursively) search through the compiler internal states, which is tightly coupled with the compiler code; while at validation time, since a concrete pre-assertion is given, we just need to search through the equations given in the pre-assertion, which is completely generic and can be easily automated.

It is important to note that automation functions do not need to be verified (*i.e.,* not a part of TCB) because all they do is to insert inference rules, which is a part of proof construction, not that of proof checking.

3 Register Promotion

Register-promotion optimization, the `mem2reg` pass of LLVM, transforms memory accesses to locally allocated memory locations into register accesses, provided that the memory location is only used for loads and stores (*i.e.,* never copied or escaped). This translation is important because register accesses are cheaper than memory accesses, and are subject to further optimizations.

The optimization also performs the SSA transformation so that the target program has the SSA property. This transformation is necessary because there can be statically multiple stores to a single location, and just transforming them to writes to a single register would break the SSA property.

In this section, we show how we generate and validate proofs for the `mem2reg` optimization.

3.1 Translation Example

The shaded part of Fig. 3 shows an example translation of the `mem2reg` optimization, where all memory accesses via `p` is promoted to register accesses to `p1` and uses of 42 and `x`. Note that `c`, `x`, and `q` are the function parameters.

More specifically, the allocation, load and store instructions to `p` are removed (ignore `lnop` for now), and every use of the result of a load from `p` is replaced by the value stored in `*p` at the time of the load. For example, in Fig. 3, the compiler figures out that `*p` contains 42 at line 20 (and so does the register `a`) due to the store of 42 in `*p` at line 11, and thus replaces the use of `a` with 42 at line 21. This translation is sound because (*i*) the assertion $*\mathtt{p}_{src} = 42$ holds from line 11 to 20; and (*ii*) $\mathtt{a}_{src} = 42$ holds from line 20 to 21. Note that we use the blue color for assertions about `*p` and the red color about the registers containing the value loaded from `*p`.

In a case where the value stored in `*p` depends on the control flow, the compiler inserts a ϕ*-node*, which is a unique construct in the SSA form and assigns different values to a register depending on the control flow. For example, at

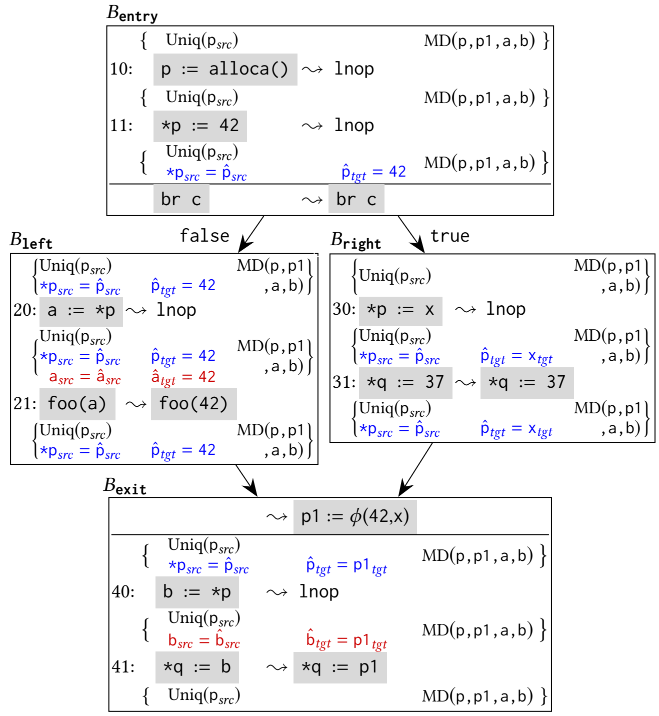


Figure 3. A register-promotion example

line 40, `*p` contains 42 if the control comes from $\mathbf{B}_{\texttt{left}}$, and `x` if it comes from $\mathbf{B}_{\texttt{right}}$. In this case, the compiler inserts a ϕ-node `p1 :=` ϕ`(42,x)` at the beginning of $\mathbf{B}_{\texttt{exit}}$, which defines `p1` to be 42 when coming from $\mathbf{B}_{\texttt{left}}$ and `x` when coming from $\mathbf{B}_{\texttt{right}}$. Then, the use of the register `b` containing the loaded value from `*p` can be replaced by `p1` at line 41.

3.2 ERHL Proof

We show how to turn the intuition for soundness into a formal ERHL proof, which is given in the unshaded part of Fig. 3 including `lnop`. Here we omit the inference rules for simplicity, which will be shown later. We introduce interesting features of ERHL by explaining each part of the proof.

Logical No-Ops for Instruction Alignment Logical no-ops, denoted `lnop`, can be inserted as part of a proof in order to align source and target instructions when their alignment is broken by a translation. For example, in Fig. 3, `lnop` is inserted at lines 10, 11, 20, 30, 40 because the instructions there are removed by `mem2reg`.

Note that `lnop` is logical because it is absent from the real IR code and used only for validation purposes. During validation, it is interpreted as doing nothing (*i.e.,* no-op).

Ghost Registers for Relational Assertions For complex optimizations, we often need to state relational properties (*i.e.,* relating the source and target states) in a proof. For example, in Fig. 3, we need to state $*\mathtt{p}_{src} = \mathtt{p1}_{tgt}$ before line 40, which relates a value in the source ($*\mathtt{p}_{src}$) with that in the target ($\mathtt{p1}_{tgt}$).

Though not directly supported in ERHL, such relational properties can be encoded using *ghost registers*. Specifically, we can encode $e_{src} = e'_{tgt}$ using a fresh ghost register $\hat{g}$:

$$\{\, e_{src} = \hat{g}_{src},\ \hat{g}_{tgt} = e'_{tgt},\ \mathrm{MD}(M) \,\} \text{ with } \hat{g} \notin M$$

Since the ghost register $\hat{g}$ is not in the maydiff set, we have $\hat{g}_{src} = \hat{g}_{tgt}$, which, by transitivity, implies $e_{src} = e'_{tgt}$. For example, in Fig. 3, the assertion $\{*\mathsf{p}_{src} = \hat{\mathsf{p}}_{src}, \hat{\mathsf{p}}_{tgt} = \mathsf{p1}_{tgt}, \mathrm{MD}(\mathsf{p}, \mathsf{p1}, \mathsf{a}, \mathsf{b})\}$ before line 40 effectively states $*\mathsf{p}_{src} = \mathsf{p1}_{tgt}$. Note that the ghost register $\hat{\mathsf{p}}$ has nothing to do with the physical register p and we use $(\hat{-})$ for ghost registers to distinguish them from the physical ones.

Ghost registers are *logical* ones that do not exist in physical program states. Instead, they are existentially quantified in the semantics of ERHL assertions. More specifically, a pair of source and target states $(\sigma_{src}, \sigma_{tgt})$ satisfies an ERHL assertion P, if there *exists* a pair of source and target ghost register files $(\hat{rs}_{src}, \hat{rs}_{tgt})$ such that the pair of σ_{src} extended with $\hat{rs}_{src}$ and σ_{tgt} extended with $\hat{rs}_{tgt}$ satisfies P.

Taking ghost registers into account, the proof in Fig. 3 has five relational assertions: $*\mathsf{p}_{src} = 42_{tgt}$ between line 11 and the end of $\mathbf{B}_{\mathtt{left}}$, $\mathsf{a}_{src} = 42_{tgt}$ between line 20 and line 21, $*\mathsf{p}_{src} = \mathsf{x}_{tgt}$ between line 30 and the end of $\mathbf{B}_{\mathtt{right}}$, $*\mathsf{p}_{src} = \mathsf{p1}_{tgt}$ between the beginning of $\mathbf{B}_{\mathtt{exit}}$ and line 41, and $\mathsf{b}_{src} = \mathsf{p1}_{tgt}$ between line 40 and line 41. It is easy to see that these assertions correctly capture the relational properties caused by executing different instructions in the source and target.

Uniqueness Predicate for Isolation We can use the predicate Uniq in order to state that an address is completely isolated. For example, in Fig. 3, we have $\mathrm{Uniq}(\mathsf{p}_{src})$ at every line. It means that in the source, if p contains an address ℓ, (*i*) ℓ is *not aliased* with any address stored in the other registers or in memory (*i.e.,* they point to disjoint memory blocks); and (*ii*) ℓ is *private* (*i.e.,* it is not in the public part of the memory injection) meaning that it has no corresponding equivalent address in the target. In other words, the address contained in p should point to a completely isolated block.

Note that ERHL also supports memory predicates weaker than $\mathrm{Uniq}(p)$: (*i*) the *privateness* predicate, $\mathrm{Priv}(p)$, which states that the address in p is private; and (*ii*) the *noalias* predicate, $p \perp q$, which states that the addresses in p and q point to disjoint memory blocks.

Maydiff Sets Finally, we have $\mathrm{MD}(\{\,\mathsf{p}, \mathsf{p1}, \mathsf{a}, \mathsf{b}\,\})$ at every line because these registers are removed or introduced so that they have different values in the source and target.

3.3 Proof Validation

We show how our proof checker validates the ERHL proof.

Entry The proof checker checks that the entry assertion, $\{\,\mathrm{Uniq}(\mathsf{p}_{src}), \mathrm{MD}(\{\,\mathsf{p}, \mathsf{p1}, \mathsf{a}, \mathsf{b}\,\})\,\}$, holds for initial states. It accepts the assertion $\mathrm{Uniq}(\mathsf{p}_{src})$ since p is a local register and thus contains the `undef` value initially, which is not an address. It also accepts every maydiff set since the source and target registers initially contain equivalent values.

Allocation of the Promoted Location At line 10, the proof checker allows an allocation, `p := alloca()`, in the source and `lnop` in the target. In this case, it computes a post-assertion from the pre-assertion by (*i*) removing all assertions containing p_{src} because p_{src} is updated, (*ii*) adding $\{\,\mathrm{Uniq}(\mathsf{p}_{src}), *\mathsf{p}_{src} = \mathtt{undef}\,\}$ because p contains a newly allocated address, and then (*iii*) adding p to the maydiff set. Thus, we have $\{\,\mathrm{Uniq}(\mathsf{p}_{src}), *\mathsf{p}_{src} = \mathtt{undef}, \mathrm{MD}(\{\,\mathsf{p}, \mathsf{p1}, \mathsf{a}, \mathsf{b}\,\})\,\}$, from which the assertion after line 10 trivially follows.

Stores to the Promoted Location At line 30 (and similarly at line 11), the proof checker allows a store, `*p := x`, in the source and `lnop` in the target because $*\mathsf{p}_{src}$ is private (*i.e.,* has no corresponding target address) due to $\mathrm{Uniq}(\mathsf{p}_{src})$ in the pre-assertion. In this case, it computes a post-assertion by (*i*) removing all and *only* the assertions containing $*\mathsf{p}_{src}$ because $*\mathsf{p}_{src}$ is updated and p_{src} has no alias with any other address due to $\mathrm{Uniq}(\mathsf{p}_{src})$, and then (*ii*) adding $\{\,*\mathsf{p}_{src} = \mathsf{x}_{src}\,\}$. Thus, we have $\{\,\mathrm{Uniq}(\mathsf{p}_{src}), *\mathsf{p}_{src} = \mathsf{x}_{src}, \mathrm{MD}(\{\,\mathsf{p}, \mathsf{p1}, \mathsf{a}, \mathsf{b}\,\})\,\}$.

At this point, the proof gives the rule $\mathtt{intro\_ghost}(\hat{\mathsf{p}}, \mathsf{x})$, which first makes $\hat{\mathsf{p}}$ fresh by removing all assertions about $\hat{\mathsf{p}}$ and removing $\hat{\mathsf{p}}$ from the maydiff set and then adds $\{\mathsf{x}_{src} = \hat{\mathsf{p}}_{src}, \hat{\mathsf{p}}_{tgt} = \mathsf{x}_{tgt}\}$ when x is not in the maydiff set. Thus, we have $\{\mathrm{Uniq}(\mathsf{p}_{src}), *\mathsf{p}_{src} = \mathsf{x}_{src}, \mathsf{x}_{src} = \hat{\mathsf{p}}_{src}, \hat{\mathsf{p}}_{tgt} = \mathsf{x}_{tgt}, \mathrm{MD}(\{\mathsf{p}, \mathsf{p1}, \mathsf{a}, \mathsf{b}\})\}$. Then, the proof gives the rule $\mathtt{transitivity}(*\mathsf{p}_{src}, \mathsf{x}_{src}, \hat{\mathsf{p}}_{src})$, which derives $*\mathsf{p}_{src} = \hat{\mathsf{p}}_{src}$ from $*\mathsf{p}_{src} = \mathsf{x}_{src}$ and $\mathsf{x}_{src} = \hat{\mathsf{p}}_{src}$. Then the assertion after line 30 trivially follows. (See [1, §I] for the definitions of `intro_ghost` and `transitivity`.)

ϕ-nodes At the ϕ-node of $\mathbf{B}_{\mathtt{exit}}$, the proof checker validates the assertion separately for each incoming block. For the incoming block $\mathbf{B}_{\mathtt{left}}$, the proof checker computes a post-assertion by (*i*) removing all assertions containing $\mathsf{p1}_{tgt}$ because $\mathsf{p1}_{tgt}$ is updated, (*ii*) adding $42 = \mathsf{p1}_{tgt}$ because `p1 := 42` is executed in the target when control comes from $\mathbf{B}_{\mathtt{left}}$, and then (*iii*) adding $\mathsf{p1}$ to the maydiff set. Then the proof gives the inference rule $\mathtt{transitivity}(\hat{\mathsf{p}}_{tgt}, 42, \mathsf{p1}_{tgt})$, which derives $\hat{\mathsf{p}}_{tgt} = \mathsf{p1}_{tgt}$, from which the assertion after the ϕ-node follows trivially. For the incoming block $\mathbf{B}_{\mathtt{right}}$, validation succeeds similarly, where the proof gives the inference rule $\mathtt{transitivity}(\hat{\mathsf{p}}_{tgt}, \mathsf{x}_{tgt}, \mathsf{p1}_{tgt})$.

Note that for presentation purposes here we simplified the post-assertion computation for ϕ-nodes. ERHL performs a more general version to handle cyclic control flows (see §4).

Loads from the Promoted Location At line 40 (and similarly at line 20), the proof checker allows a load, `b := *p`, in the source and `lnop` in the target. In this case, it computes a post-assertion by (*i*) removing all assertions containing b_{src} because b_{src} is updated, (*ii*) adding $\mathsf{b}_{src} = *\mathsf{p}_{src}$ and then (*iii*) adding b to the maydiff set. Thus, we have $\{\mathrm{Uniq}(\mathsf{p}_{src}), *\mathsf{p}_{src} = \hat{\mathsf{p}}_{src}, \hat{\mathsf{p}}_{tgt} = \mathsf{p1}_{tgt}, \mathsf{b}_{src} = *\mathsf{p}_{src}, \mathrm{MD}(\{\mathsf{p}, \mathsf{p1}, \mathsf{a}, \mathsf{b}\})\}$.

At this point, the proof gives the rule intro_ghost($\hat{b}$, $\hat{p}$), which adds $\{\, \hat{p}_{src} = \hat{b}_{src}, \hat{b}_{tgt} = \hat{p}_{tgt} \,\}$ because $\hat{p}$ is not in the maydiff set. Then the proof gives appropriate transitivity rules, which derives $b_{src} = {*}p_{src} = \hat{p}_{src} = \hat{b}_{src}$ and $\hat{b}_{tgt} = \hat{p}_{tgt} = p1_{tgt}$, from which the assertion after line 40 trivially follows.

Equivalence Checking At lines 21, 31 and 41, the proof checker checks that the behaviors of the source and target instructions are equivalent. Specifically, it checks that equivalent values are passed to the same function (at line 21) and stored at equivalent public locations (at lines 31,41) because these can be observed by other functions. These checks succeed thanks to the relational assertions ($\{\, a_{src} = \hat{a}_{src}, \hat{a}_{tgt} = 42 \,\}$ at line 21, $\{\, b_{src} = \hat{b}_{src}, \hat{b}_{tgt} = p1_{tgt} \,\}$ at line 41).

Alias Checking At lines 21, 31, and 41, the proof checker computes post-assertions using memory-alias information. In general, for a function call or store instruction, since it updates the public part of the memory, the proof checker removes all assertions about values stored in memory locations p (*i.e.,* those including $*p$) unless (i) p is in the private part of the memory (*i.e.,* Priv(p) or Uniq(p)), or (ii) p is not aliased with q (*i.e.,* $p \perp q$) in case $*q$ is updated by the store instruction. At lines 21, 31 and 41, thanks to Uniq(p_{src}), the assertions about $*p_{src}$ are preserved.

Note that in the example of Fig. 3, it suffices to use Priv(p_{src}) instead of Uniq(p_{src}). However, in general when more than one location is promoted, we need to know that those promoted locations are not aliased with each other, which follows from Uniq(p_{src}) for each promoted location p. Also for the sake of performance, we use Uniq instead of introducing $\perp$ between each pair of promoted locations.

3.4 Proof Generation

LLVM's mem2reg pass consists of three algorithms: the general register-promotion algorithm and two specialized ones optimized for efficiency: one for the case that the promotable location is stored at most once and the other for the case that the location is used only within a single block. In this section we explain the general algorithm and its proof-generation code. Note that we also validate the two specialized algorithms in the same way since they are just degenerate cases.

Algorithm 2 shows the general algorithm implemented in LLVM's mem2reg pass and the proof-generation code, given in the box, that we inserted. Note that we do not modify the existing compiler code at all and only add the proof-generation code. In detail, the overall algorithm including proof generation works as follows.

Promotable Allocation **[Line A1]** We find a promotable allocation p at line l_a. **[Line A2]** Then we insert empty ϕ-nodes wherever needed[7], and add them to the maydiff set globally (*i.e.,* at every line). **[Line A3]** We also remove

[7]The optimization uses the "dominance frontier" algorithm [18] in order to list up the blocks that require a ϕ-node. We omit the details for brevity.

Algorithm 2 RegisterPromotion(F:Function)

A1: **for** l_a: p:=alloca() **in** F **if** p's uses are loads/stores only **do**
A2: InsertEmptyPhinodesFor(F, p)
 // $\boxed{\text{Add the } \phi\text{-nodes to the maydiff set globally}}$
A3: Remove(l_a), $\boxed{\text{Nop}(l_a, tgt), \text{Assn}(\{\text{Uniq}(p_{src}), \text{MD}(p)\}, global)}$
A4: $\boxed{\text{Inf}(\texttt{intro\_ghost}(\hat{p}, \texttt{undef}), l_a)}$
A5: WL := [(Entry(F), undef, $\boxed{l_a}$)], MarkVisited(Entry(F))
A6: **while** NonEmpty(WL) **do**
A7: $(B, v, \boxed{l})$ :: WL := WL
A8: **for** $(l_i : i)$ **in** Instr(B) **do**
A9: **if** i **is** a store instruction $(*p := w)$ **then**
A10: Remove(l_i), $\boxed{\text{Nop}(l_i, tgt), \text{Inf}(\texttt{intro\_ghost}(\hat{p}, w), l_i)}$
A11: $v := w$, $\boxed{l := l_i}$
A12: **else if** i **is** a load instruction $(x := *p)$ **then**
A13: $\boxed{\text{Assn}(\{*p_{src} = \hat{p}_{src}, \hat{p}_{tgt} = v_{tgt}\}, l, l_i)}$
A14: $\boxed{\text{Inf}(\texttt{intro\_ghost}(\hat{x}, \hat{p}), l_i)}$
A15: **for** $(l_j : j)$ **in** Use(x) **do**
A16: Replace(F, l_j, x, v), $\boxed{\text{Assn}(\{x_{src} = \hat{x}_{src}, \hat{x}_{tgt} = v_{tgt}\}, l_i, l_j)}$
A17: **end for**
A18: Remove(l_i), $\boxed{\text{Nop}(l_i, tgt), \text{Assn}(\{\text{MD}(x)\}, global)}$
A19: **end if**
A20: **end for**
A21: **for** B' **in** Successor(B) **do**
A22: **if** B' has a ϕ-node $(z := \phi(\cdots))$ inserted at line A2 **then**
A23: $z[B] := v$, $\boxed{\text{Assn}(\{*p_{src} = \hat{p}_{src}, \hat{p}_{tgt} = v_{tgt}\}, l, \text{End}(B))}$
A24: **if not** IsVisited(B') **then** $WL := (B', z, \boxed{\text{Begin}(B')})$ :: WL
A25: **else**
A26: **if not** IsVisited(B') **then** $WL := (B', v, \boxed{l})$ :: WL
A27: **end if**
A28: MarkVisited(B')
A29: **end for**
A30: **end while**
A31: **end for**
A32: $\boxed{\text{Auto}(\texttt{transitivity})}$

the allocation, insert lnop at that line, and add Uniq(p_{src}) and MD(p) globally. **[Line A4]** In addition, we add the rule intro_ghost($\hat{p}$,undef) because the initial value undef in $*p$ may be used by some load from $*p$ (though it does not happen in Fig. 3). In that case, the code at line A13 would introduce $\{\, *p_{src} = \hat{p}_{src}, \hat{p}_{tgt} = \texttt{undef} \,\}$ at line l_a, which will be inferred with the help of intro_ghost($\hat{p}$,undef).

For example, in Fig. 3, the empty ϕ-node p1 := $\phi(-, -)$ is inserted in $\mathbf{B}_{\texttt{exit}}$ and p1 is added to the maydiff set globally; then the allocation at line 10 is removed, lnop is inserted, Uniq(p_{src}) is added and p is added to the maydiff set globally; and finally intro_ghost($\hat{p}$,undef) is added at line 10.

Block Traversal **[Lines A5-A7]** We traverse the blocks in DFS order starting from the entry block using the worklist WL. An element of WL consists of triple (B,v,l), where B is the block to visit, v is the value in $*p$ at the beginning of B, and l is the line number where the value v is stored in $*p$. **[Line A5]** Initially, we put (Entry(F), `undef`, line l_a) in *WL* and mark the entry block Entry(F) as visited. **[Lines A6-A7]** Then we process the blocks in *WL* one by one. For example, in Fig. 3, $\mathbf{B}_{\texttt{entry}}$, $\mathbf{B}_{\texttt{left}}$, $\mathbf{B}_{\texttt{exit}}$, and $\mathbf{B}_{\texttt{right}}$ are visited in order.

Instruction Traversal **[Line A8]** Given a work (B, v, l), we traverse each instruction $(l_i : i)$ in the block B as follows.

Store Instructions **[Lines A9-A11]** If i is a store instruction $*p := w$ (line A9), then we remove the instruction (line A10) and update v with the stored value w (line A11). The proof-generation code inserts `lnop`, adds `intro_ghost`($\hat{p}, w$) (line A10) and updates l with the store location l_i (line A11).

For example, in Fig. 3, when i is 11: `*p := 42`, the store i is replaced by `lnop`; `intro_ghost`($\hat{\texttt{p}}$, 42) is added at line 11; and v and l are updated to be 42 and line 11.

Load Instructions **[Lines A12-A18]** If i is a load instruction $x := *p$ (line A12), then we replace all the uses of x with the current value v (lines A15-A17), and remove the load instruction (line A18). The proof-generation code adds the relational assertion $*p_{src} = v_{tgt}$ from the store site l to the load site l_i (line A13) and the rule `intro_ghost`($\hat{x}, \hat{p}$) at l_i (line A14). Then it adds $x_{src} = v_{tgt}$ from the load site l_i to every use site l_j (line A16). It also inserts `lnop` at l_i in the target and adds x to the maydiff set globally (line A18).

For example, in Fig. 3, when i is 20: `a := *p`, the load i is replaced by `lnop`; the use of `a` is replaced by the current value 42 at line 21; $*\texttt{p}_{src} = 42_{tgt}$ is added from 11 to 20; `intro_ghost`($\hat{\texttt{a}}, \hat{\texttt{p}}$) is added at line 20; $\texttt{a}_{src} = 42_{tgt}$ is added from 20 to 21; and `a` is added to the maydiff set globally.

Successors **[Lines A21-A28]** Now we handle the successor (*i.e.,* outgoing) blocks of the current block B. **[Line A21]** We traverse each successor block B' as follows.

- If B' has a ϕ-node ($z := \phi(\cdots)$) that is inserted by the code at line A2 (line A22), then we update the ϕ-node z's component for the incoming block B with the value v of $*p$ at the end of B (line A23). In addition, if B' has not been visited yet, we add $(B', z, \text{Begin}(B'))$ to the worklist *WL* (line A24). Since the value v is used at the ϕ-node z, we add $*p_{src} = v_{tgt}$ from store location l to the end of B (line A23). For example, in Fig. 3, when (B, B') is $(\mathbf{B}_{\texttt{left}}, \mathbf{B}_{\texttt{exit}})$, the ϕ-node `p1` $:= \phi(-,-)$ is updated to `p1` $:= \phi(42,-)$ and $(\mathbf{B}_{\texttt{exit}}, \texttt{p1}, \text{Begin}(\mathbf{B}_{\texttt{exit}}))$ is added to the worklist *WL*. Also $*\texttt{p}_{src} = 42_{tgt}$ is added from line 11 to the end of $\mathbf{B}_{\texttt{left}}$.
- If B' has no such ϕ-node (line A25), then we simply add (B', v, l) to the worklist *WL* if B' has not been visited yet (line A26). For example, when (B, B') is $(\mathbf{B}_{\texttt{entry}}, \mathbf{B}_{\texttt{right}})$, the triple $(\mathbf{B}_{\texttt{right}}, 42, \text{line } 11)$ is added to the worklist.

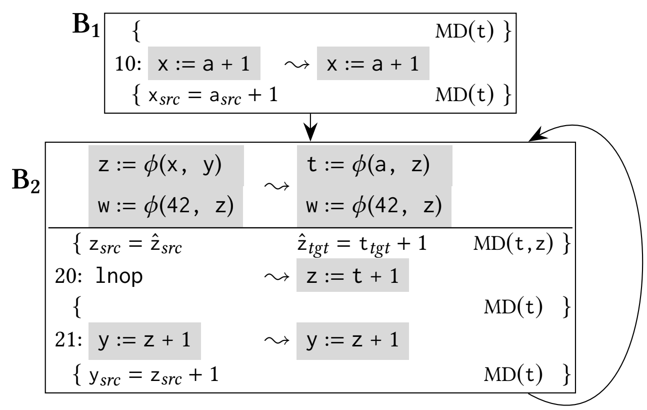
[Line A28] Finally the successor B' is marked as visited.

Inference Rules As shown in §3.3, the complete proof for `mem2reg` contains two inference rules, `intro_ghost` and `transitivity`. The `intro_ghost` rules are explicitly added by the proof-generation code shown in Algorithm 2, while the `transitivity` rules are automatically added by the automation function `transitivity` (line A32).

4 Reasoning about Cyclic Control Flows

In this section, using an example of `fold-`ϕ optimization, we discuss a challenge in ERHL validation arising from cyclic control flows and show how to address it.

Fold-ϕ Optimization Consider the translation below performed by the `fold-`ϕ optimization of `instcombine`, and its ERHL proof. This translation basically replaces `z` $:= \phi$(`x`, `y`) with `z` $:= \phi$(`a`, `z`)+1 using the temporary variable `t` $:= \phi$(`a`, `z`). This removes the dependence of `z` on `x` and `y`, thereby allowing `x` and `y` to be eliminated away by a subsequent optimization unless they are used elsewhere. This translation is correct because we have $\texttt{z}_{src} = \phi(\texttt{x}_{src}, \texttt{y}_{src}) = \phi(\texttt{a}_{src}{+}1, \texttt{z}_{src}{+}1) = \phi(\texttt{a}_{tgt} + 1, \texttt{z}_{tgt} + 1) = \phi(\texttt{a}_{tgt}, \texttt{z}_{tgt}) + 1 = \texttt{t}_{tgt} + 1 = \texttt{z}_{tgt}$.



Note that a set of ϕ-nodes can appear at the beginning of a block and are executed *simultaneously*. For example, in the source program above, when control flows from $\mathbf{B}_2$ to itself, the ϕ-nodes `z` and `w` are set to the *old values* of `y` and `z` just before executing the ϕ-nodes, respectively. In particular, `w` is set to the old value of `z`, not the new value stored in `z` at the first ϕ-node, and thus `w` contains the same value in the source and target programs.

Challenge The challenge here is that we should be able to express and reason about both old and new values of `z`. This is because `z` is used and defined at the same time in the ϕ-nodes, which is only possible due to cyclic control flows in the SSA form. Specifically, the proof checker should derive something like $\texttt{z}_{src} = \texttt{y}_{src}$ and $\texttt{w}_{src} = old(\texttt{z}_{src})$ as part of the strong post-condition after the ϕ-nodes when control flows from $\mathbf{B}_2$.

We address this challenge by expressing the old value of register $old(\texttt{z}_{src})$ using a ghost variable. Specifically, we reserve a set of ghost registers, denoted $\bar{r}$ and called *old registers*, for all registers r to represent the old value of r. Note, however, that old registers are just normal ghost registers and technically have nothing to do with physical old values of the corresponding registers.

Proof Validation We show how the ERHL proof checker systematically uses the old registers by validating the above proof in the most interesting case: the ϕ-nodes of $\mathbf{B}_2$ when control comes from itself.

First, it computes a post-assertion from the pre-assertion $\{\, \mathtt{y}_{src} = \mathtt{z}_{src} + 1, \mathrm{MD}(\mathtt{t})\,\}$ as follows.

1. It removes all assertions about old registers from the pre-assertion and copies all assertions about current registers into those about old ones.

$$\{\, \mathtt{y}_{src} = \mathtt{z}_{src} + 1, \bar{\mathtt{y}}_{src} = \bar{\mathtt{z}}_{src} + 1, \mathrm{MD}(\mathtt{t}, \bar{\mathtt{t}})\,\}.$$

2. It computes a post-condition from this new assertion as if the assignments `z := ȳ`, `w := z̄` are executed in the source and `t := z̄`, `w := z̄` in the target. Specifically, it (*i*) removes source assertions about `z`,`w` and target ones about `t`,`w` because those registers are updated; (*ii*) adds `t`,`z` to the maydiff set because they are updated differently in the source and target (note that `w` is updated equivalently since $\bar{\mathtt{z}}$ is not in the maydiff set); and (*iii*) adds the equalities corresponding to the executed assignments. Thus we have

$$\{\, \bar{\mathtt{y}}_{src} = \bar{\mathtt{z}}_{src} + 1, \mathtt{z}_{src} = \bar{\mathtt{y}}_{src}, \mathtt{w}_{src} = \bar{\mathtt{z}}_{src}, \\ \mathtt{t}_{tgt} = \bar{\mathtt{z}}_{tgt}, \mathtt{w}_{tgt} = \bar{\mathtt{z}}_{tgt}, \mathrm{MD}(\mathtt{t}, \bar{\mathtt{t}}, \mathtt{z})\,\}.$$

Then the proof gives the rule $\mathtt{intro\_ghost}(\hat{\mathtt{z}}, \bar{\mathtt{z}} + 1)$, which adds $\{\, \bar{\mathtt{z}}_{src} + 1 = \hat{\mathtt{z}}_{src}, \hat{\mathtt{z}}_{tgt} = \bar{\mathtt{z}}_{tgt} + 1\,\}$ because $\bar{\mathtt{z}}$ is not in the maydiff set. Then the automation function derives $\{\, \mathtt{z}_{src} = \hat{\mathtt{z}}_{src}, \hat{\mathtt{z}}_{tgt} = \mathtt{t}_{tgt} + 1\,\}$ by transitivity: $\mathtt{z}_{src} = \bar{\mathtt{y}}_{src} = \bar{\mathtt{z}}_{src} + 1 = \hat{\mathtt{z}}_{src}$ and $\hat{\mathtt{z}}_{tgt} = \bar{\mathtt{z}}_{tgt} + 1 = \mathtt{t}_{tgt} + 1$. Then it eliminates $\bar{\mathtt{t}}$ from the maydiff set after eliminating all assertions about $\bar{\mathtt{t}}$, which is sound because $\bar{\mathtt{t}}$ is just a ghost variable that has nothing to do with a physical value of the register `t`. Finally, the assertion after the ϕ-nodes $\{\, \mathtt{z}_{src} = \hat{\mathtt{z}}_{src}, \hat{\mathtt{z}}_{tgt} = \mathtt{t}_{tgt} + 1, \mathrm{MD}(\mathtt{t}, \mathtt{z})\,\}$ trivially follows by a simple inclusion check.

5 ERHL Proof Checker and Logic

In this section, we explain the proof checker in terms of the ERHL logic, and describe the soundness of the proof checker using the semantic interpretation of the logic. All our results are formally verified in Coq (see [1, §H] for details).

Proof Rules The checker is based on the proof rules presented in Fig. 4. The checker is given the source and target programs Prg_{src}, Prg_{tgt} and a translation proof Ψ, and tries to deduce $Prg_{src} \sim Prg_{tgt}$ using the (Sim) rule. Here, $Entry(F)$ denotes the entry block of the function F; $Prg[F].\zeta[B,i]$ the i-th instruction of the block B in F; and $Prg[F].\phi[B,B']$ the assignment instructions of the ϕ-nodes of B' when control comes from B (*e.g.,* in the source program in §4, $Prg[F].\phi[\mathbf{B}_1,\mathbf{B}_2] = \{\, \mathtt{z := x}, \mathtt{w := 42}\,\}$). Also, $\Psi[F].\alpha[B,i]$ denotes the assertion in the proof Ψ just before the i-th instruction of B in F (it denotes the last assertion when $i = -1$).

$$\boxed{Prg_{src} \sim Prg_{tgt}}$$

$$\text{(Sim)}\quad \frac{\begin{array}{c}\mathrm{CheckCFG}(Prg_{src}, Prg_{tgt}) \qquad \forall F \in Prg_{src}.\ \mathrm{CheckInit}(\Psi[F].\alpha[\mathrm{Entry}(F),0]) \\ \forall F,B,i.\ \{\Psi[F].\alpha[B,i]\}\ Prg_{src}[F].\zeta[B,i] \sim Prg_{tgt}[F].\zeta[B,i]\ \{\Psi[F].\alpha[B,i{+}1]\} \\ \forall F,B,B'.\ \{\Psi[F].\alpha[B,-1]\}\ Prg_{src}[F].\phi[B,B'] \sim Prg_{tgt}[F].\phi[B,B']\ \{\Psi[F].\alpha[B',0]\}\end{array}}{Prg_{src} \sim Prg_{tgt}}$$

$$\boxed{\{P\}\ I_{src} \sim I_{tgt}\ \{Q\}}$$

$$\text{(PostAssn)}\quad \frac{\begin{array}{c}\mathrm{CheckEquivBeh}(P, I_{src}, I_{tgt}) \\ Q = \mathrm{CalcPostAssn}(P, I_{src}, I_{tgt})\end{array}}{\{P\}\ I_{src} \sim I_{tgt}\ \{Q\}} \qquad \text{(Consequence)}\quad \frac{\begin{array}{c}\{P\}\ I_{src} \sim I_{tgt}\ \{Q\} \\ Q \Rrightarrow Q'\end{array}}{\{P\}\ I_{src} \sim I_{tgt}\ \{Q'\}}$$

$$\boxed{Q \Rrightarrow Q'}$$

$$\text{(Trans)}\quad \frac{\begin{array}{c}Q \Rrightarrow Q' \\ Q' \Rrightarrow Q''\end{array}}{Q \Rrightarrow Q''} \qquad \text{(ApplyInf)}\quad \frac{\begin{array}{c}rule \in \mathrm{CustomRules} \\ Q' = \mathrm{ApplyInf}(rule, Q)\end{array}}{Q \Rrightarrow Q'} \qquad \text{(Incl)}\quad \frac{\mathrm{CheckIncl}(Q, Q')}{Q \Rrightarrow Q'}$$

Figure 4. Proof Rules of ERHL

The checker first checks if Prg_{src} and Prg_{tgt} have the same CFG (CheckCFG), the assertion in the entry is satisfied by the initial states for each function (CheckInit), and the Hoare triple $\{P\}\ I_{src} \sim I_{tgt}\ \{Q\}$ is valid for all matching intra-block commands I_{src} and I_{tgt} and their pre- and post-assertions P and Q given by Ψ. For example, in Fig. 2, it checks at line 20 if $\{\mathtt{x}_{src} = a_{src} + 1, MD(\emptyset)\}\ \mathtt{y := x + 2} \sim \mathtt{y := a + 3}\ \{MD(\emptyset)\}$ is valid. It also checks for each inter-block edge from B to B' that $\{P\}\ Prg_{src}.\phi[B,B'] \sim Prg_{tgt}.\phi[B,B']\ \{Q\}$ is valid, where P is the last assertion in B and Q is the first assertion in B'.

To validate a Hoare triple $\{P\}\ I_{src} \sim I_{tgt}\ \{Q\}$, the checker first computes a post-assertion Q_0 with $\{P\}\ I_{src} \sim I_{tgt}\ \{Q_0\}$ using the rule PostAssn (see [1, §H] for the definition of CheckEquivBeh and CalcPostAssn). Then it suffices to validate $Q_0 \Rrightarrow Q$ by the rule Consequence.

For this, using the rules ApplyInf and Trans, the checker iteratively applies a sequence of inference rules $rule_1, \ldots, rule_n$ (either given by Ψ or generated by an automation function) and deduces $Q_0 \Rrightarrow Q_n$, where $Q_i = \mathrm{ApplyInf}(rule_i, Q_{i-1})$.

Finally, the checker validates $Q_n \Rrightarrow Q$ using the rule Incl, where CheckIncl performs a simple inclusion check.

Semantic Interpretation For the soundness of the proof checker, we give the semantic interpretation of the top-level judgment as semantics preservation, or behavior refinement:

$$[\![Prg_{src} \sim Prg_{tgt}]\!] \stackrel{\text{def}}{=} Beh(Prg_{src}) \supseteq Beh(Prg_{tgt})\ .$$

The soundness of (Sim) is proved using a local simulation in the style of [22], which is a simplification of parametric bisimulation [21]. First, we show that CheckInit(P) implies:

$$\forall \sigma_{src}, \sigma_{tgt}, \alpha.\ FInit(\sigma_{src}) \wedge FInit(\sigma_{tgt}) \implies [\![P]\!]_\alpha(\sigma_{src}, \sigma_{tgt})\ .$$

Here, $FInit(\sigma)$ means σ is a possible initial state of a function call, $[\![P]\!]$ is the semantic interpretation of the assertion P (see [1, §G] for details), and α is a CompCert-style memory injection [28], which basically maps a memory block in the source to an equivalent one in the target.

Second, we give the semantic interpretation of the Hoare triple for non-call instructions I_{src}, I_{tgt} as a simulation step:

$$\begin{aligned}\llbracket\{P\}\ I_{src} \sim I_{tgt}\ \{Q\}\rrbracket \stackrel{\text{def}}{=} & \forall\sigma_{src}.\ Instr(\sigma_{src}) = I_{src} \implies \\ & \forall\sigma_{tgt}.\ Instr(\sigma_{tgt}) = I_{tgt} \implies \\ \forall\alpha, \sigma'_{tgt}, \varepsilon.\ & \llbracket P\rrbracket_{\alpha}(\sigma_{src}, \sigma_{tgt}) \ \land\ \sigma_{tgt} \xrightarrow{\varepsilon} \sigma'_{tgt} \implies \\ \exists\sigma'_{src}, \alpha'.\ & \llbracket Q\rrbracket_{\alpha'}(\sigma'_{src}, \sigma'_{tgt}) \ \land\ \sigma_{src} \xrightarrow{\varepsilon} \sigma'_{src} \ \land\ \alpha \sqsubseteq \alpha'\end{aligned}$$

where, $Instr(\sigma)$ is the next instruction to execute in the program state σ, and $\sigma \xrightarrow{\varepsilon} \sigma'$ means the state σ steps to σ' emitting an observable event ε. Also, $\sqsubseteq$ is the extension relation of memory injection.

For call instructions I_{src}, I_{tgt}, $\llbracket\{P\}\ I_{src} \sim I_{tgt}\ \{Q\}\rrbracket$ basically states that Q is satisfied by all possible equivalent returns states when an arbitrary function is called from states satisfying P (see [1, §H] for details). We followed the basic approach of parametric bisimulation [21].

The semantic interpretation of $\Rrightarrow$ is as follows:

$$\begin{aligned}\llbracket Q \Rrightarrow Q'\rrbracket \stackrel{\text{def}}{=} \forall\sigma_{src}, \sigma_{tgt}, \alpha.\ & \llbracket Q\rrbracket_{\alpha}(\sigma_{src}, \sigma_{tgt}) \implies \\ & \exists\alpha'.\ \llbracket Q'\rrbracket_{\alpha'}(\sigma_{src}, \sigma_{tgt}) \ \land\ \alpha \sqsubseteq \alpha'\ .\end{aligned}$$

For the soundness of (APPLYINF), every custom *rule* should satisfy that $\llbracket Q \Rrightarrow \text{ApplyInf}(rule, Q)\rrbracket$ holds for all Q.

6 Implementation

We developed the CRELLVM framework for LLVM 3.7.1.

Coverage We wrote proof-generation code for register promotion in the `mem2reg` pass and GVN-PRE in the `gvn` pass implemented in the following files respectively:

- `lib/Transforms/Utils/PromoteMemoryToRegister.cpp`
- `lib/Transforms/Scalar/GVN.cpp`

For `mem2reg`, we covered the entire file, and for `gvn`, we covered all functions except for the following functions: `SimplifyInstruction`, `processLoad`, `splitCriticalEdges` and `MergeBlockIntoPredecessor`. These functions are not part of the main GVN-PRE algorithm because they are not technically related to value numbering (*i.e.,* neither using nor constructing value numbering). Other reasons why we omitted them are because `SimplifyInstruction` is a common function that just consists of many peephole optimizations and the others use features that are not currently supported by CRELLVM: `processLoad` uses the alias analysis module and `splitCriticalEdges` and `MergeBlockIntoPredecessor` change control-flow graphs. Note that the reason why those functions are used by the `gvn` pass is because they transform programs in such a way that opportunities for GVN-PRE optimizations are increased.

To demonstrate the generality of ERHL logic and the proof checker, we also covered a part of the loop-invariant code motion (`licm`) pass that can be currently supported by CRELLVM and 139 micro-optimizations of the instruction combining (`instcombine`) pass (see [1, §D] for details).

	`mem2reg`	`gvn`	`licm`	`instcombine`
Compiler (Covered)	568	1,092	706	702
Proof Generation	213	440	286	1,357

Figure 5. SLOC of Proof-Generation Code

Proof-Generation Code We explicitly mark as "not supported" for translations using operations not supported by VELLVM, or relying on deep analyses such as division-by-zero and alias analyses.

Fig. 5 shows the SLOC in C++ of the compiler and proof-generation code for each pass. The SLOC ratio of the proof-generation code to that of the corresponding compiler code is 37.5% for `mem2reg`, 40.3% for `gvn`, 40.5% for `licm`, and 193.3% for `instcombine`. The CRELLVM infrastructure for proof-generation consists of 1,708 lines for common library and 15,980 lines for JSON serialization library, of which 72.2% is automatically generated from 2,079 SLOC in a simple DSL.

Inference Rules In the proof checker we installed 221 custom inference rules, of which 202 are arithmetic rules like `assoc_add`. All 9 non-arithmetic rules used for `mem2reg`, `gvn`, and `licm`, including `transitivity` and `intro_ghost`, are formally verified in Coq (see [1, §I] for details).

Verification of Proof Checker In order to reduce TCB, we formally verified the soundness of the proof checker in Coq (see §5). It is worth noting that we achieved the same kind of guarantee as CompCert for the translations that are validated by the proof checker using only verified inference rules.

We used the formal semantics of LLVM IR from the VELLVM project [55], but significantly upgraded the semantics in various ways. In particular, VELLVM used the CompCert memory model [28] version 1.9 and we upgraded it to version 2.4 in order to use the notion of *permission* in the LLVM semantics; and added the `switch` instruction to the formalization of LLVM IR. Note that VELLVM has a simpler memory model than the LLVM's informal official one (*e.g.,* pointer-equality tests and pointer-integer casts are more undefined).

In total, our Coq development consists of 25,970 SLOC. The proof checker is 2,987 SLOC, and its verification is 18,934 SLOC. The 221 inference rules are 2,193 SLOC, and the verification of 9 rules took 1,856 SLOC. Note that the underlying semantics of VELLVM consists of 39,307 SLOC.

Experience Writing proof-generation code was an iterative process: we had to repeat bug-fix processes many times. When proof checking fails, it tells us a logical reason for the failure so that we could easily identify the bug in proof-generation code (or else in the compiler). We believe the iteration could be shortened if we collaborated with LLVM developers.

	Results			Time (sec.)			
	#V	**#F**	**#NS**	**Orig**	**PCal**	**I/O**	**PCheck**
`mem2reg`	76.79K	10	10.58K	8.59	322.18	13.16K	21.26K
`gvn`	365.99K	453	7.92K	41.81	249.85	41.96K	37.89K
`licm`	168.20K	0	24.93K	22.42	895.93	56.44K	11.36K
`instcombine`	1593.84K	0	528.75K	184.49	442.85	152.51K	105.40K

Figure 6. Experimental Results

Custom functions for automatically finding inference rules are greatly helpful for developing proof-generation algorithms. Using such automation, we could develop much simpler proof-generation algorithms for `mem2reg` and `gvn`, compared to our initial development, by making the code size less than half and speeding up more than twice.

Crellvm is less cost-effective for peephole optimizations in `instcombine`. We had to write 1.9 lines of proof-generation code for each line of the corresponding compiler code, and we did not verify arithmetic inference rules. Even though Crellvm achieves higher level of reliability, we think more automated approaches using an SMT solver such as Alive [30] would be more cost-effective for peephole optimizations.

7 Experiment

Benchmarks Using Crellvm, we validated the compilation of the SPEC CINT2006 C Benchmarks [15], LLVM nightly test suite, and five open-source projects written in C (the biggest benchmarks used in [37][8]), totaling 5.3 million LOC in C. We omitted 3 files from the benchmarks because they contain instructions currently not supported by Vellvm, including the `indirectbr` instruction.

Fig. 6 summarizes the validation results and the time spent on running the proof-generation codes and the proof checker for each optimization pass. In the experiment, we compiled each benchmark program with the `-O2` flag, and validated the intermediate translations with the generated proofs. For more detailed results, see [1, §A].

We show the total number of translation steps (**#V**), the number of not-supported translations (**#NS**), and the number of translations failed at validation (**#F**). The rest of the translations (*i.e.,* **#V** − **#F** − **#NS**) succeeded in validation. Also, all the successful translations were shown to be equivalent to the original translations using the `llvm-diff` tool. During the experiment, we also found and reported a bug in `llvm-diff`, which has been confirmed and fixed [8].

Out of 2,205K validations in total, 1632K (74.0%) are successfully validated. All 463 (0.01%) failures (**#F**) are due to compiler bugs: 10 are due to the `mem2reg` bug [5] we discussed in §1.2, 295 are due to the two `gvn` bugs [6, 7] we found, and 158 are due to a known `gvn` bug [11] that is currently fixed in the LLVM trunk. Note that there is no failure due to the other `mem2reg` bug [9] we found.

The other 572.2K (26.0%) translations (**#NS**) are currently not supported in our validator. Among them, 555.9K (97.1%) use instructions not supported by Vellvm: vector operations 515.1K (90.0%), aggregate type operations 30.4K (5.31%), debug attributes 8.7K (1.52%), and atomic operations 1.7K (0.29%). 13.0K (2.27%) use the alias and division-by-zero analysis modules of LLVM; 2.3K (0.41%) alter type declarations; and 0.7K (0.12%) require deeper analysis on functions such as read-only function analysis.

We measured the time spent on performing each optimization in the original compiler (**Orig**); on performing each optimization and calculating validation proofs in the modified compiler (**PCal**); on writing and reading the source and target programs with the proofs via files (**I/O**); and on validating the proofs by the proof checker (**PCheck**). The table shows total times aggregated over the entire run.

In the experiment, we embarrassingly parallelized compilation and validation jobs and fully utilized the 96 hardware threads from four identical workstations with Intel Xeon E5-2630 CPU (2.6GHz, 12 cores, 2 hardware threads per core), 128GB RAM, and 1TB SSD (Samsung 850 PRO). The whole experiment took about three hours in wall clock.

Validating Randomly Generated Programs We randomly generated 1,000 C programs using CSmith [53], compiled them with `-O2` flag, and validated the intermediate translations with the generated proofs. All 55,008 validations for `gvn` are successfully validated, except for one due to the `gvn` bug [6] we found. Out of 42,584 validations for `mem2reg`, 11,816 (27.7%) are currently not supported due to LLVM lifetime intrinsics, which is not supported by Vellvm. The other 30,768 (72.3%) are all successfully validated.

Performance Proof checking takes much more time than regular compilation, but we believe it is still reasonable for compiler writers to use Crellvm for stabilizing compilers. Also, as we have shown in the experiment, compiler writers can further reduce runtime by checking proofs in parallel. Furthermore, there is still a large room for performance improvement as we have not done any serious performance analysis and tuning for the proof checker. In particular, we believe we can significantly reduce I/O time, which is one of the current bottlenecks, by writing proofs in binary format rather than in plain-text JSON format and also by writing only the changes made between IR files rather than writing full IR files. In our benchmark, the Clang frontend generated 4,885 IR files with average size of 187.63 KB, from which 2,205K validations with average proof size of 17.5 KB were generated.

Bug Reports By November 2016 when we completed our initial implementation of Crellvm for LLVM 3.7.1, we reported three miscompilation bugs, one in `mem2reg` [5] and two in `gvn` [6, 7], which were immediately confirmed and subsequently fixed. Around July 2017 when we verified selected inference rules, we reported another miscompilation bug in `mem2reg` [9], which was immediately confirmed but has not been fixed yet (as of 14 April 2018) because it is

[8]We omitted Linux, since it is currently not compiled with LLVM (see [29]).

unlikely to occur in practice (it did not occur in our benchmark either) and there is no consensus on how to fix it. Around March 2018, we additionally covered the function `performScalarPREInsertion` in `gvn`, which was omitted initially because it is loosely related to value numbering: deciding whether to perform the transformation, not the transformation itself, depends on value numbering. The reason for this coverage is because we were informed of a new bug [11] found in the function. As we have seen above, CRELLVM successfully detected the bug by failing at 158 validations.

8 Discussion

8.1 Reliability

In order to see how effectively CRELLVM improved reliability of LLVM, we investigated all bug reports about miscompilation in `mem2reg` and `gvn` since the release of LLVM 3.7.1. To the best of our knowledge, other than the five bugs [5–7, 9, 11] detected by CRELLVM, there is no confirmed miscompilation bug that is (*i*) due to the code we covered in `mem2reg` and `gvn` and (*ii*) not related to any LLVM feature that is currently not supported by CRELLVM (as of 14 April 2018).

Specifically, we conducted our investigation as follows. We checked all relevant bug reports in the LLVM bug tracker [4] and OSS-Fuzz bug tracker [3]. Moreover, we asked the `llvm-dev` mailing list about relevant bugs [2]. We also posted a draft of this paper on our website in February 2018 and received comments. One of the most important comments was about the `gvn` bug [11] in the code we newly covered (*i.e.,* the function `performScalarPREInsertion`). The bug was discovered and fixed in October 2017 by Azul Systems via fuzz testing of the company's LLVM-based Java JIT compiler, using JavaFuzzer [10] (private communication with Philip Reames, March 2018).

8.2 Maintainability

To evaluate maintenance cost, we ported our full development of CRELLVM to LLVM 5.0.1 just omitting `instcombine` because it is not our main target. After the initial porting, which took two days, we found one validation fail in `gvn` due to insufficient proof generation. We fixed it by adding an automation function, which took 5 days by one person including analysis of the problem. After applying the `gvn` bug fix [11] in the main trunk to LLVM 5.0.1, our benchmark experiment produces no validation failures except for not-supported ones (see [1, §A] for details).

8.3 Limitations and Future Work

We discuss current limitations of CRELLVM, which also indicate a direction of future research.

Semantics VELLVM does not fully formalize the LLVM IR semantics. First, it does not support several features of LLVM IR, including atomic operations for concurrency, vector operations and attributes like `noalias`, `readonly` and `nsw`.

Second, VELLVM does not properly formalize casts between integers and pointers, which itself is a challenging research topic. Applying the idea of Kang *et al.* [22] would be interesting future research.

Finally, VELLVM does not properly formalize the *undef* and *poison* values, which is another research problem. Recently, Lee *et al.* [25] proposed a possible solution to this problem using a new instruction, called *freeze*. Applying it to VELLVM would be interesting work.

Analyses Our proof checker does not support various analysis passes such as division-by-zero analysis, alias analysis, read-only function analysis, and memory dependence analysis. We believe it would be possible to support them by adding appropriate predicates and inference rules in the underlying logic of proof checker.

CFG-Changing Optimizations CRELLVM relies on the condition that the source and target programs can be aligned line-by-line by inserting logical no-op instructions. While we think this condition holds for majority of LLVM optimizations, there are several important optimizations that break the condition by changing the control-flow graph. Examples include loop unrolling, loop unswitching and loop splitting. We believe it would be possible to support them by generalizing the proof checker following the ideas from existing translation validation works [36, 49–52, 57].

9 Related Work

A large number of prior work on improving reliability of compiler are roughly classified into the following categories.

Credible Compilation Rinard *et al.* [44], who coined the term *credible compilation*, proposed the framework of credible compilation and presented a relational Hoare logic, in which one can reason about register allocation and instruction scheduling optimizations in the presence of pointer aliasing. Independently, Benton [16] proposed a relational Hoare logic for a functional language. However, their logics are designed for simple languages, and the framework has not been implemented and applied to compilers.

Namjoshi *et al.* [33, 34] presented a "proof of concept" implementation of credible compilation (or a witnessing compiler in their terminology) for LLVM optimizations such as constant propagation, dead-code elimination, and LICM. However, the work can be seen as rather preliminary for the following reasons. First, their proof checker supports a small subset of LLVM IR, most notably ignoring memory operations. Second, it assumes that main functions of the compiler are correct. For example, it assumes that the constant-folding function of LLVM is correct.

Verified translation validation is similar to verified credible compilation but differs in that it develops a verified validator specialized for a particular optimization, rather than developing a proof checker for a general logic. Various verified translation validators have been developed for CompCert: instruction scheduling [50], lazy code motion [51], software pipelining [52]; register allocation [43]; SSA transformation [14]; and GVN and sparse conditional constant propagation (SCCP) [19].

(Foundational) proof carrying code (PCC) [12, 35] is similar to (verified) credible compilation, but it employs a (verified) unary logic for validating safety properties of the generated target program.

Translation Validation This approach develops a general validator that checks correctness of any given translation between IR programs without requiring any proof. Compared to credible compilation, translation validation is more scalable (*i.e.,* more easily applicable to different optimizations) because it requires much less manual effort due to no need for writing proof-generation code. On the other hand, though it can be used to guarantee correctness of certain compilations, it can hardly be used to find compiler bugs due to many false positives. The reason for false positives is that such a general validator is inherently incomplete since it is agnostic to the compiler's internal logic.

Due to such incompleteness, a variety of translation validators with different heuristics and trade-offs were proposed [20, 36, 38, 39, 45–47, 49, 54, 57, 58]. In particular, Tristan *et al.* [49] and Stepp *et al.* [46] developed translation validators for LLVM optimization passes, including dead-code elimination, GVN-PRE, constant propagation, and LICM. However, they failed at about 20% of the validations, most of which are likely to be false positives.

Compiler Verification Verified compilers provide the highest level of reliability by proving the semantics-preservation property for all possible source programs in a proof assistant. CompCert [26, 27] is the most sophisticated formally verified optimizing C compiler, whose correctness is proved in Coq [13], and CakeML [23] is an optimizing ML compiler formally verified in the HOL4 theorem prover [40]. However, verifying a full-fledged compiler is highly costly and verified compilers are usually much less performant than production compilers.

Zhao *et al.* [55, 56] implemented and verified the `vmem2reg` pass for LLVM in Coq, but its algorithm is significantly simplified compared to that in LLVM. Their simplified algorithm is based on a rewriting logic in which each rewriting step preserves semantics and each intermediate program is type-checked. On the other hand, LLVM's register-promotion algorithm temporarily breaks the semantics-preservation property and even the intermediate programs are not type-checked, because ill-formed empty ϕ-nodes are inserted in the middle and their arguments are filled later. According to the authors, this renders the formal verification hard for the register-promotion implementation in LLVM.

DSL for Optimizations Lopes *et al.* [30–32] presented Alive, a DSL for writing peephole optimizations using the SMT solver Z3 [41]. With Alive, one can either prove the correctness of an optimization or find a counterexample. They ported 300 micro-optimizations of `instcombine` to Alive, and in doing so they found 8 bugs in `instcombine`. However, the Alive DSL is not expressive enough to describe complex algorithms such as `mem2reg` and `gvn`, and limited to supporting only peephole optimizations that do not involve reasoning about cyclic control flows. In addition, Alive makes simplifying assumptions on the LLVM semantics, and their encoding of an optimization into SMT queries is a part of the TCB. Furthermore, since there is a gap between an actual implementation in C++ and a corresponding algorithm description in Alive DSL, implementation bugs cannot be detected. Tatlock and Lerner [48] also presented a DSL for writing CompCert optimizations based on a rewriting logic, but it is not general enough to support register promotion and GVN-PRE.

Compiler Testing Random testing tools such as CSmith [17, 42, 53] and EMI [24] have been very successful. They have found hundreds of bugs in GCC and LLVM. However, most of them are found in the `instcombine` pass and none of them are miscompilation bugs in `mem2reg` and `gvn`.

10 Conclusion

We have demonstrated that the credible-compilation approach scales to the production compiler LLVM by developing our Crellvm framework. We also empirically demonstrated that Crellvm can be an effective tool for achieving high reliability of major optimizations by discovering four long-standing bugs in the `mem2reg` and `gvn` passes.

Acknowledgments

We thank Daniel Berlin, Davide Italiano, Yeonwoo Kim, Philip Reames, John Regehr, and anonymous reviewers for very helpful feedback, and Sung-hwan Lee for his contribution to early development of Crellvm. This research was supported by Samsung Research Funding Center of Samsung Electronics under Project Number SRFC-IT1502-07. Jeehoon Kang, Yoonseung Kim, and Juneyoung Lee have been supported by Korea Foundation for Advanced Studies Scholarships.

References

[1] Supplementary material for this paper, available at http://sf.snu.ac.kr/crellvm/.
[2] http://lists.llvm.org/pipermail/llvm-dev/2018-April/122482.html.
[3] https://bugs.chromium.org/p/oss-fuzz.
[4] https://bugs.llvm.org/.
[5] https://bugs.llvm.org/show_bug.cgi?id=24179.
[6] https://bugs.llvm.org/show_bug.cgi?id=28562.
[7] https://bugs.llvm.org/show_bug.cgi?id=29057.
[8] https://bugs.llvm.org/show_bug.cgi?id=33623.
[9] https://bugs.llvm.org/show_bug.cgi?id=33673.

[10] https://github.com/AzulSystems/JavaFuzzer.
[11] https://reviews.llvm.org/D38619.
[12] Andrew W. Appel. 2001. Foundational Proof-Carrying Code *(LICS '01)*.
[13] The Coq Proof Assistant. https://coq.inria.fr/.
[14] Gilles Barthe, Delphine Demange, and David Pichardie. 2014. Formal Verification of an SSA-Based Middle-End for CompCert. *ACM Trans. Program. Lang. Syst.* 36, 1 (March 2014).
[15] The SPEC CINT2006 Benchmark. https://www.spec.org/cpu2006/CINT2006/.
[16] Nick Benton. 2004. Simple Relational Correctness Proofs for Static Analyses and Program Transformations *(POPL '04)*.
[17] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. 2013. Taming Compiler Fuzzers *(PLDI '13)*.
[18] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (Oct. 1991).
[19] Delphine Demange, David Pichardie, and Léo Stefanesco. 2016. Verifying Fast and Sparse SSA-Based Optimizations in Coq *(CC '16)*.
[20] Chris Hawblitzel, Shuvendu K. Lahiri, Kshama Pawar, Hammad Hashmi, Sedar Gokbulut, Lakshan Fernando, Dave Detlefs, and Scott Wadsworth. 2013. Will You Still Compile Me Tomorrow? Static Cross-version Compiler Validation *(ESEC/FSE '13)*.
[21] Chung-Kil Hur, Derek Dreyer, Georg Neis, and Viktor Vafeiadis. 2012. The Marriage of Bisimulations and Kripke Logical Relations. In *POPL*.
[22] Jeehoon Kang, Chung-Kil Hur, William Mansky, Dmitri Garbuzov, Steve Zdancewic, and Viktor Vafeiadis. 2015. A Formal C Memory Model Supporting Integer-pointer Casts *(PLDI '15)*.
[23] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A Verified Implementation of ML *(POPL '14)*.
[24] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler Validation via Equivalence Modulo Inputs *(PLDI '14)*.
[25] Juneyoung Lee, Yoonseung Kim, Youngju Song, Chung-Kil Hur, Sanjoy Das, David Majnemer, John Regehr, and Nuno P. Lopes. 2017. Taming Undefined Behavior in LLVM *(PLDI '17)*.
[26] Xavier Leroy. 2006. Formal Certification of a Compiler Back-end or: Programming a Compiler with a Proof Assistant *(POPL '06)*.
[27] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* (2009).
[28] Xavier Leroy, Andrew W. Appel, Sandrine Blazy, and Gordon Stewart. 2012. *The CompCert Memory Model, Version 2*. Research report RR-7987. INRIA.
[29] LLVM Linux. http://llvm.linuxfoundation.org.
[30] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. 2015. Provably Correct Peephole Optimizations with Alive *(PLDI '15)*.
[31] David Menendez and Santosh Nagarakatte. 2017. Alive-Infer: Data-driven Precondition Inference for Peephole Optimizations in LLVM *(PLDI '17)*.
[32] David Menendez, Santosh Nagarakatte, and Aarti Gupta. 2016. Alive-FP: Automated Verification of Floating Point Based Peephole Optimizations in LLVM *(SAS '16)*.
[33] Kedar S. Namjoshi, Giacomo Tagliabue, and Lenore D. Zuck. 2013. A Witnessing Compiler: A Proof of Concept *(RV '13)*.
[34] Kedar S. Namjoshi and Lenore D. Zuck. 2013. Witnessing Program Transformations *(SAS '13)*.
[35] George C. Necula. 1997. Proof-carrying Code *(POPL '97)*.
[36] George C. Necula. 2000. Translation Validation for an Optimizing Compiler *(PLDI '00)*.
[37] Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, Daejun Park, Jeehoon Kang, and Kwangkeun Yi. 2014. Global Sparse Analysis Framework. *ACM Trans. Program. Lang. Syst.* 36, 3 (Sept. 2014).
[38] Amir Pnueli, Michael Siegel, and Eli Singerman. 1998. Translation Validation *(TACAS '98)*.
[39] Amir Pnueli, Ofer Strichman, and Michael Siegel. 1998. The Code Validation Tool CVT: Automatic Verification of a Compilation Process *(STTT '98)*.
[40] HOL Interactive Theorem Prover. https://hol-theorem-prover.org/.
[41] The Z3 Theorem Prover. https://github.com/Z3Prover/z3.
[42] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs *(PLDI '12)*.
[43] Silvain Rideau and Xavier Leroy. 2010. Validating Register Allocation and Spilling *(CC '10)*.
[44] Martin C. Rinard and Darko Marinov. 1999. Credible Compilation with Pointers *(RRV '99)*.
[45] Hanan Samet. 1978. Proving the Correctness of Heuristically Optimized Code *(ACM '78)*.
[46] Michael Stepp, Ross Tate, and Sorin Lerner. 2011. Equality-based Translation Validator for LLVM *(CAV '11)*.
[47] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality Saturation: A New Approach to Optimization *(POPL '09)*.
[48] Zachary Tatlock and Sorin Lerner. 2010. Bringing Extensibility to Verified Compilers *(PLDI '10)*.
[49] Jean-Baptiste Tristan, Paul Govereau, and Greg Morrisett. 2011. Evaluating Value-graph Translation Validation for LLVM *(PLDI '11)*.
[50] Jean-Baptiste Tristan and Xavier Leroy. 2008. Formal Verification of Translation Validators: A Case Study on Instruction Scheduling Optimizations *(POPL '08)*.
[51] Jean-Baptiste Tristan and Xavier Leroy. 2009. Verified Validation of Lazy Code Motion *(PLDI '09)*.
[52] Jean-Baptiste Tristan and Xavier Leroy. 2010. A Simple, Verified Validator for Software Pipelining *(POPL '10)*.
[53] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers *(PLDI '11)*.
[54] Anna Zaks and Amir Pnueli. 2008. CoVaC: Compiler Validation by Program Analysis of the Cross-Product *(FM '08)*.
[55] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. 2012. Formalizing the LLVM Intermediate Representation for Verified Program Transformations *(POPL '12)*.
[56] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. 2013. Formal Verification of SSA-based Optimizations for LLVM *(PLDI '13)*.
[57] Lenore Zuck, Amir Pnueli, Benjamin Goldberg, Clark Barrett, Yi Fang, and Ying Hu. 2002. Translation and Run-Time Validation of Loop Transformations *(RV '02)*.
[58] Lenore D. Zuck, Amir Pnueli, and Benjamin Goldberg. 2003. VOC: A Methodology for the Translation Validation of Optimizing Compilers *(J. UCS '03)*.