Master's Thesis

# Verified Credible Compilation Framework For Early CSE in LLVM

LLVM 내의 Early CSE 를 위한
검증되고 신뢰할 수 있는 컴파일러 프레임워크

August 2018

Graduate School of Seoul National University
Department of Computer Science and Engineering
College of Engineering

Mark Dongyeon Shin

Master's Thesis

# Verified Credible Compilation Framework For Early CSE in LLVM

LLVM 내의 Early CSE 를 위한
검증되고 신뢰할 수 있는 컴파일러 프레임워크

August 2018

Graduate School of Seoul National University
Department of Computer Science and Engineering
College of Engineering

Mark Dongyeon Shin

# Verified Credible Compilation Framework For Early CSE in LLVM

LLVM 내의 Early CSE 를 위한
검증되고 신뢰할 수 있는 컴파일러 프레임워크

Academic advisor Chung-Kil Hur

Submitting a master's thesis of Engineering

May 2018

Graduate School of Seoul National University

Department of Computer Science and Engineering
College of Engineering

Mark Dongyeon Shin

Confirming the master's thesis
written by Mark Dongyeon Shin

July 2018

| | | |
|---|---|---|
| Chair | Kwangkeun Yi | (Seal) |
| Vice Chair | Chung-Kil Hur | (Seal) |
| Examiner | Sungjoo Yoo | (Seal) |

# Abstract

## Verified Credible Compilation Framework For Early CSE in LLVM

Mark Dongyeon Shin
School of Computer Science and Engineering
College of Engineering
The Graduate School
Seoul National University

Compiler verification is important when obtaining a high level of reliability through software verification. Compiler bugs are crucial for software verification because code that running programs are not source code but execution code. However, many C/C++ mainstream compilers, including GCC and LLVM focus on efficiency rather than reliability. Although testing is an effective method to identify bugs, it does not guarantee a high level of reliability. Various approaches have been proposed to examine compiler internal logics, but as yet none have been very successful.

CRELLVM is a compiler framework that validates optimization passes in LLVM to ensure high reliability of LLVM optimizations. It is able to validate major optimizations of LLVM such as Register Promotion and Global Value Numbering.

This thesis shows validation of Early CSE optimization in LLVM, using CRELLVM. For the validation, proof generation code which corresponds to Early CSE in LLVM is implemented and the proof checker has been extended. Early CSE is one of the basic optimizations in LLVM that removes the repeated computations by erasing duplicated instructions.

Based on 5.40 million lines of C code benchmarks, the experiment result shows there is no mis-compilation for Early CSE, which guarantees a high level of reliability of Early CSE in the benchmarks.

**Keywords**: Credible Compilation, Compiler Verification, LLVM, CRELLVM, Early CSE

**Student ID.**: 2016-21215

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Compiler verification is critical, however, there has been considerably less interests in compiler verification compared with source code verification. The actual running software is execution code, hence source code verification is pointless if the source code is not correctly translated to its execution code.

Most mainstream C/C++ compilers concentrate on improved efficiency rather than focusing on reliability, including GCC and LLVM. Therefore, many optimizations are performed during compilation and new optimizations are aggressively added. Because of unverified optimizations, bug reports are continuously updated in the LLVM compiler community[6], and modifying code in one part can affect the other parts, causing unexpected errors. These compiler bugs are difficult to find and can significantly reduce software reliability.

Many approaches were proposed for high reliability in compilers. The verified compiler CompCert[8], developed by INRIA guarantees high reliability, but has lower performance than mainstream compilers. Therefore, it is not widely used for real world applications.

Our approach is not to lower performance and to provide a high level of reliability for optimizations in LLVM, which generates their correctness proofs and checks them with a proof checker. We developed CRELLVM: a verified credible compilation framework for LLVM [10] that provides a high level of reliability in optimizations such as `Global Value Numbering(GVN), Register Promotion, Loop Invariant Code Motion(LICM)` and `InstCombine`.

This paper describes the validation of Early CSE optimization in LLVM 3.7.1 with CRELLVM.

## 1.1 CRELLVM Framework

The framework of CRELLVM is shown in Figure 1.1. Compilation and validation phases are separated. For compilation, the original optimizer optimizes the source Intermediate Representation(IR) code `src.ll` into the target `tgt.ll`. For validation, the optimizer is instrumented with proof generation code. Proof generation code in the LLVM optimizer only creates its proof and does not affect LLVM optimization. The instrumented optimizer will return `tgt'.ll` and the proof. Then, the proof is given to the proof checker which validates `src.ll` is correctly translated into `tgt'.ll`. If `tgt.ll` and `tgt'.ll` are identical and the proof checker succeeds validation, then the optimization is correct. Equality checker llvm-diff checks identicalness of `tgt.ll` and `tgt'.ll` according to alpha-equivalence.

Equality checker is Trust Computing Base(TCB) in this framework. Also, the proof checker is TCB, which is verified in Coq. Therefore, if validation fails, there must be a bug in LLVM optimization or something wrong with the proof generation.

This framework can be independent of the original compiler. Therefore, programmers can use the original compiler and compiler developers can use the framework to validates their newly added optimization and increases the reliability.

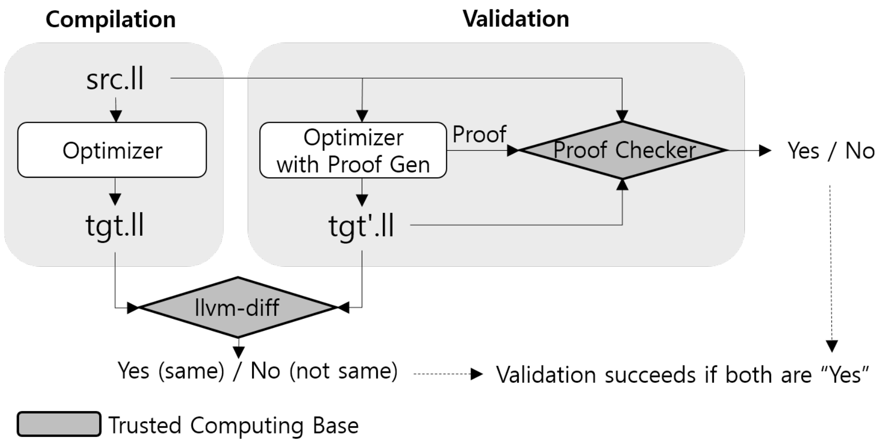

Figure 1.1: The CRELLVM Framework

## 1.2 Proof Checker

The proof checker reasons about LLVM optimizations by a variant of relational Hoare logic [5], called Extensible Relational Hoare Logic(ERHL) [10]. This proof checker's logic and inference rules are verified in the style of CompCert using the Coq [2] formalization of LLVM IR from the `VELLVM` project [32][10].

One of advantages of the CRELLVM is that one proof checker is able to validates various optimization passes. The proof checker's logic can be extended by adding custom inference rules. Therefore, it can be used for general purpose, with no need to make a new proof checker to validate a new pass. The new proof generation code need only be inserted in the instrumented LLVM optimizer to validate a new optimization, without changing the proof checker.

## 1.3 Early CSE

Common Subexpression Elimination(CSE) optimization erases an instruction which has duplicated rhs expression within the code, and replaces its usage with the corresponding register. Early CSE is one of the basic optimizations in the first Clang optimization level, *i.e.,* Clang optimization option O1. The world

"Early" means that it deals with simple cases and shifts more complicated ones on the later GVN pass.

```
x := add a b;        x := add a b;
foo(x);              foo(x);
                 ⇝
y := add a b;        (lnop;)
goo(y);              goo(x);
```

In the above example, lnop instruction is just a logical instruction that aligns the source and the target code(more details in Section 3.1). Early CSE optimization examines instructions one by one to check for duplicated instructions. In this example, after going through the instruction x and foo(x), it found out that the y instruction's rhs is same as the instruction x's rhs. Therefore, it erases the instruction y and replaces every uses of the y with the x, so goo(y) becomes goo(x).

## 1.4 Result

Early CSE optimization in LLVM 3.7.1 was validated using benchmarks comprising 5.40 million lines of C code and found 97.13% to be successful, 2.87% not supported, and no validation failure. This indicates there is no mis-compilation bug of Early CSE optimization in the benchmarks.

## 1.5 Outline

This paper is about validating Early CSE optimization in LLVM 3.7.1 using CRELLVM. The rest of the paper is as follows. Section 2 gives overview of the Early CSE optimization, which shows how Early CSE it works optimization. Section 3 explains about proof validation and proof generation. Validation process and instrumented proof generation code are shown with optimization examples. Section 4 shows results of the experiment. Section 5 is the related work and Section 6 concludes.

# Chapter 2

# Early CSE

Early CSE optimization deletes an instruction which has duplicated rhs expression and replaces uses of the deleted instruction with the corresponding register. Early CSE is one of optimizations in the Clang optimization level O1. It indicates that Early CSE optimization is the one of the basic optimizations in LLVM.

## 2.1 Early CSE translation example

Early CSE optimization is about deleting duplicated instructions. The following translation is a simple example that shows Early CSE.

$$\begin{array}{lcl} \texttt{x} := \texttt{add a, b;} & & \texttt{x} := \texttt{add a, b;} \\ \texttt{y} := \texttt{add a, b;} & \rightsquigarrow & (\texttt{lnop;}) \\ \texttt{z} := \texttt{add x, y;} & & \texttt{z} := \texttt{add x, x;} \end{array}$$

The instruction `x` := `add a, b` consists of register `x` and righthand side(rhs) expression `add a, b`. Sometimes we call this instruction as instruction `x` because LLVM instructions are of Static Single Assignment(SSA) forms, thus register `x` can be the name of the instruction.

Instructions `x` and `y` have the same rhs. Both instruction has the same rhs, which means the instruction `y` is duplicated. Then, Early CSE deletes the instruction `y` and replaces uses of the `y` with the register `x`. The instruction `z` becomes `add x, x` after optimization.

## 2.2 Early CSE optimization

In LLVM, Early CSE can handle the instructions of `CastInst`, `BinaryOperator`, `GetElementPtrInst`, `CmpInst`, `SelectInst`, `ExtractElementInst`, `Insert-ElementInst`, `ShuffleVectorInst`, `ExtractValueInst`, `InsertValueInst`, `LoadInst` and `CallInst`.

This section discusses the optimization processes for them. Here let us call the instructions except for `LoadInst` and `CallInst`, *simple instructions*. Early CSE handles `LoadInst` and `CallInst` slightly in different ways from the simple instructions because these two instructions are affected by memory write operations. Also, Early CSE includes the optimization for branch instruction and extra optimizations not related to CSE. Discussions about them are added later.

***Simple instructions*** Early CSE deals with every simple instruction in a similar way. It maintains `AvailableValues` hash table to check if an instruction has been duplicated or not. If the current instruction is a new one, the register and its rhs are inserted in `AvailableValues`. If it is duplicated, thus rhs exists in the hash table, the optimizer erases the instruction and replace all uses of it in the code with the corresponding register from `AvailableValues`.

***LoadInst*** There are a few conditions needed for Early CSE optimization on `LoadInst`. First, the allocated memory location of `LoadInst` must not be volatile. Second, `LoadInst` should load from a same memory location. Values loaded from the same memory location can be duplicated. Values loaded from the same memory location are equal when there is no memory write operation to the

| Instruction types | Syntax |
|---|---|
| `CastInst` | %x := `bitcast i64 %a to i32` |
| `BinaryOperator` | %x := `add i32 %a, %b` |
| `GetElementPtrInst` | %x := `getelementptr %struct.ST, %struct.ST* %s, i32 1` |
| `CmpInst` | %x := `icmp eq i32 a, b` |
| `SelectInst` | %x := `select i1 true i32 %a, i32 %b` |
| `ExtractElementInst` | %x := `extractelement <4 x i32> %vec, i32 0` |
| `InsertElementInst` | %x := `insertelement <4 x i32> %vec, i32 1, i32 0` |
| `ShuffleVectorInst` | %x := `shufflevector <4 x i32> %v1, <4 x i32> %v2,` |
| | `<4 x i32> <i32 0, i32 4, i32 1, i32 5>` |
| `ExtractValueInst` | %x := `extractvalue i32, float %agg, 0` |
| `InsertValueInst` | %x := `insertvalue i32, float %agg, float %val, 1` |
| `LoadInst` | %x := `load i32, i32* %ptr` |
| `CallInst`(non-void) | %x := `call i32 %foo(i32 %argc)` |
| `CallInst`(void) | `call void %foo(i32 %argc)` |

Table 2.1: LLVM Instructions

location between successive `LoadInsts`. Therefore, if first, second conditions are met and the `currentgeneration` variable, which increments when there is any memory write operation, is same as the previous `LoadInst`, then it is possible to optimize.

Rather than checking if a memory write operation has occurred at the indicated memory location, it assumes a `LoadInst` value can be different from any memory write operation, which seems overly strict. However, this is not too much because Early CSE does not use alias analysis result. It does not know the memory write operation has alias with the location, thus Early CSE checks every memory write operation. Therefore, if `currentgeneration` is unchanged, there must be no memory write operation between `LoadInsts`, thus `LoadInst` has the same value; whereas if there was a memory write operation between successive `LoadInsts`, `currentgeneration` is changed and Early CSE optimization is disallowed.

When there is no memory write operation in between and two `LoadInsts` are loading from the same memory location, then Early CSE considers the instruction as duplicated, and deletes the instruction and replace all usages with existing register. Optimization process proceeds as for the simple instructions.

For different memory location, it is obvious that these are different instructions. Furthermore, for different values of the `currentgeneration`, Early CSE considers this to be a different instruction, even if it is loaded from the same memory location. Therefore, it inserts the register, rhs and `currentgeneration` into the hash table `AvailableLoads`, which is similar to `AvailableValues` hash table but specified for `LoadInst`.

***CallInst*** For `CallInst`, it is available to optimize when it is non-void type and read only call instruction. Early CSE optimizes `CallInst` similar to `LoadInst`. Optimization is available when two call instructions are calling same function and the `currentgeneration` values are equal. If it is not, it considers as a different instruction and inserts into `AvailableCalls` hash table.

***Branch Instruction*** LLVM includes the branch instruction(`BranchInst`), which has syntax `br %c %B1, %B2`. If the condition `c` is true, the next block will be block `B1`, otherwise block `B2`. If `c` is true and control flow flows to block `B1` then all values of `c` in blocks dominated by `B1` will be true. If `c` is false, control flow flows to `B2` then all values of `c` in blocks dominated by `B2` will be false. Early CSE optimizes this case as follows. In the blocks where block `B1` dominates, it substitutes every use of `c` as true, conversely as false when use of `c` is dominated by block `B2`.

***Extra Optimizations*** There are several optimizations unrelated to CSE that Early CSE optimization also performs. These are SimplifyInstruction, Dead Code Elimination(DCE) and Dead Store Elimination(DSE).

SimplifyInstruction, as the name suggests, simplifies instruction that can be simply computed or converted. This optimization frequently exists in front of or back of many others optimization passes. Since this is not related to CSE, this paper does not validate this optimization. It is only a simple calculation, so omitting this validation does not critically affects overall Early CSE reliability.

The DCE operation erases a dead code. It is a simple work and there exists the validation function available, created when validating other passes. Therefore, It is simply added without further modification.

The DSE erases a dead store instruction, *i.e.,* when two instructions store into the same memory location but there was no intervening `LoadInst`, the first store instruction becomes the dead store instruction and is removed. This optimization is also not relevant to CSE, would require extending the proof checker to validate. However, this optimization only occurs 0.1% of the employed benchmarks. Therefore, DSE is not validated in this paper but marked as not supported validation for the future work.

## 2.3 Block Traversal

Early CSE performs optimization by traversing all blocks in each function. This section discusses how Early CSE traverses blocks in a function.

***Dominator*** In control flow graphs, if every path from the entry node to node `n` must pass through node `d`, then we say node `d` dominates `n`. Also, if `d` dominates `n` but `d` and `n` are the not same node, then `d` is a strict dominator of `n`. There can be more than one strict dominator of node `n`, however if one strict dominator does not strictly dominates other strict dominator of `n`, then it is called the immediate dominator of `n`.

To erase a duplicate instruction it should be satisfied that this instruction is dominated by the existing instruction. Thus, the dominance is the key for Early CSE.

Early CSE traverses blocks using data structure `DomTreeNode` from LLVM which is a dominator tree of blocks in the function. Early CSE visits blocks using depth first walk over the dominator tree, which is similar to, but slightly different from a depth first search.

To perform the depth first walk over the dominator tree, Early CSE uses the `nodeToProcess` stack, which includes `AvailableValues, AvailableLoads, AvailableCalls` hash tables, `currentgeneration` and dominator tree data.

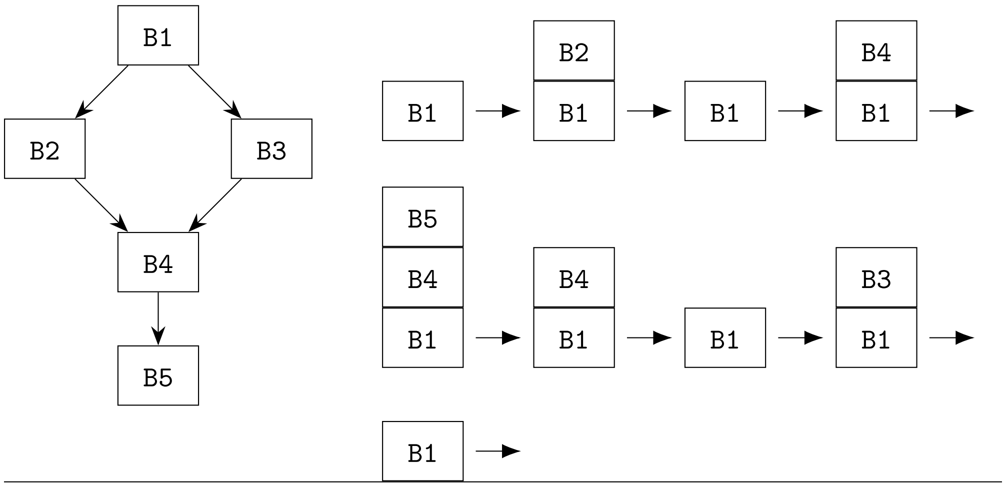

Figure 2.1: Block Traversal Example

Consider the example in Figure 2.1. Initially, the optimizer pushes empty hash tables and other data into the stack. Then it performs optimization at block B1, fills the hash tables and updates the `currentgeneration` value. After finishing block B1, it checks which block is dominated by B1. In this example, block B2, B3, B4 and B5 are all dominated by B1. However, LLVM `DomTreeNode` only considers the immediate dominance. Therefore, `DomTreeNode` considers only blocks B2, B4 and B3. Since block B2 is the first block dominated by B1, it pushes block B2 data and updated hash tables into the stack and optimizes B2. After optimization at block B2, it then checks if B2 is dominating any blocks. Since B2 does not dominate anything, it pops the top of the stack that was about B2. Next, it checks the next dominated block of B1 because B1 is still in the stack. Block B4 becomes the next target and is pushed into the stack. After completing block B4, it checks if B4 has any dominate block and it pushes B5 data and hash tables to the stack. When optimization of B5 is done, it pops B5 since there is no successor. It also pops block B4 since all the dominate successors are gone. The stack still has B1, with the final dominated block being B3. It pushes B3 and performs optimization. When optimization of B3 is done, it pops B3, and finally pops B1 then finishes optimization of Early CSE.

# Chapter 3

# Proof Generation and Validation for Early CSE

This chapter discusses how CRELLVM validates Early CSE. Proof generation code associated with Early CSE optimizer generates a proof which consists of a set of assertions and a list of inference rules. Then, proof checker validates the proof's correctness by Extensible Relational Hoare Logic(ERHL) [10].

Section 3.1 introduces ERHL, Section 3.2 explains the validation of Early CSE optimization, and Section 3.3 details the proof generation.

## 3.1 ERHL Proof

This section explains some interesting features of the ERHL proof.

***Hoare Logic*** Since ERHL is an extension of Hoare logic, we explain about Hoare logic briefly.

Hoare logic is a formal method to assess the correctness of a program. It uses a Hoare triple {P} C {Q} to express a correctness property where P and Q are assertions and C is a command. P is called the pre-assertion, and Q the post-assertion of C. When pre-assertion P is met, then post-assertion Q should be established after executing C.

***ERHL Assertions*** In ERHL, one Hoare logic assertion divides into source, target, and the maydiff assertions. Source and target assertions are as the same as Hoare logic that holds for the source state and the target state. Maydiff is a relational assertion that relates the source and the target state. Maydiff is a set of registers that contains registers that does not have same value in the source and the target. A register not in the maydiff set has same value in the source and the target. To satisfy the ERHL, all assertions of source, target, and the maydiff should be correct.

***Lessdef*** The `Undef` value included in LLVM IR is a superset of all the values and arithmetic of `Undef` is also `Undef`. Correctness of a translation does not require its value to be equal to the source value. As long as target's behavior is smaller than source, correctness is satisfied. Hence, it is possible that compiler can change `Undef` value in the source to any value in the target. The LLVM compiler actually uses this property for optimization, which can break equivalence between the source and target. Therefore, we use *lessdef* rather than equivalence to validate `Undef`, a concept taken from CompCert [14]. a $\sqsupseteq$b is a *lessdef* b where a might be `Undef` or if a is not `Undef` then a $=$ b.

The examples in this thesis do not include `Undef` value hence $=$ is used rather than *lessdef* here for simplicity.

***lnop*** `lnop` is a logical instruction because it does not exist in real IR code and is interpreted as nothing during validation. Its only purpose is alignment. During optimization, the optimizer can erase or add instructions, which causes source and target instructions to misalign. Therefore, `lnop` instruction can be inserted in the source or the target either. `lnop` is inserted into the target code instead of the instruction that the optimizer erases. Also, there are optimizations that add an instruction and one of them is `trunc_onebit` optimization in `InstCombine`.

It optimizes `z = trunc x to i1` into `y = and x, 1; z = icmp ne y, 0`. It adds one more instruction therefore, it needs `lnop` instruction in the source code for alignment.

## 3.2 Proof Validation

As mentioned above, the proof checker follows ERHL and a proof is consists of a set of assertions and a list of inference rules. Inserted proof generation code returns the proof and the proof checker checks the correctness of the proof. This section discusses how the proof checker validates Early CSE with assertions and inference rules.

Checking each line of ERHL consists of 5 steps. It will be shown with the example in Figure 3.1 . The boxed assertions and inference rules are what the proof generation code produced for the validation.

1. **Strong Post-Assertion Computation** The proof checker computes strong post-assertion of the source and the target using the standard post-assertion computation algorithm of the Hoare logic.

   In line 10, since there is no pre-assertion, the proof checker simply returns $x_{src}$ = `add` $a_{src}$ $b_{src}$ as the strong post-assertion for the source only by using the instruction `x := add a b`. It returns the similar assertion for the target also. However, line 15 has the pre-assertion $x_{src}$ = `add` $a_{src}$ $b_{src}$ in the source. The proof checker returns strong post-assertions $x_{src}$ = `add` $a_{src}$ $b_{src}$ and $y_{src}$ = `add` $a_{src}$ $b_{src}$ by using the pre-assertion and the instruction `y := add a b` together. No target strong post-assertion is returned here.

   In addition to return strong post-assertions using pre-assertion and an instruction, the proof checker checks if some registers have different values in the source and the target. In line 15, the register `y` has a value in the source but nothing in the target because it is erased in the target.

Therefore, the register `y` is included in the maydiff. Strong post-assertions at line 20 also includes the register `z` in the maydiff, because the source has $z_{src} = \texttt{add}\ x_{src}\ \ y_{src}$ but the target has $z_{tgt} = \texttt{add}\ x_{tgt}\ \ x_{tgt}$. First operand $x_{src}$ and $x_{tgt}$ is equal because the register `x` is not in the maydiff which indicates that it has a same value in the source and the target. However, second operand $y_{src}$ and $x_{tgt}$ are different registers. Therefore, the proof checker considers the source and the target has different value of the `z` and add register `z` in the maydiff set.

2. **Apply Inference Rules** Proof generation code decides which inference rule should be inserted into a given location. The selected inference rule can make a new strong post-assertion under a certain condition. The rule `transitivity`($x_{src}$, add $a_{src}$ $b_{src}$, $y_{src}$) in line 15 derives $x_{src}$ = $y_{src}$ from $x_{src}$ = `add` $a_{src}$ $b_{src}$ and $y_{src}$ = `add` $a_{src}$ $b_{src}$.

$$\frac{\texttt{TRANSITIVITY}(x_{src}, add\ \ a_{src}\ \ b_{src}, y_{src}) \qquad \qquad}{\begin{array}{c} x_{src} = \texttt{add}\ a_{src}\ \ b_{src} \qquad y_{src} = \texttt{add}\ a_{src}\ \ b_{src} \\ \hline \text{add}\ \{\, x_{src} = y_{src} \,\} \end{array}}$$

Also, `substitute` rule is applied to the source strong post-assertions $z_{src}$ = `add` $x_{src}$ $y_{src}$ and $x_{src}$ = $y_{src}$ of line 20. Using the strong post-assertions in the source, the proof checker can apply `substitute` rule that derives $z_{src}$ = `add` $x_{src}$ $x_{src}$.

$$\texttt{SUBSTITUTE}(z_{src}, x_{src}, y_{src})$$
$$\frac{z_{src} = \texttt{add}\ x_{src}\ \ y_{src} \qquad x_{src} = y_{src}}{\text{add}\ \{\, z_{src} = \texttt{add}\ x_{src}\ \ x_{src} \,\}}$$

`Transitivity` rule, `substitute` rule and other rules can be applied only under appropriate conditions. Otherwise, it fails to make a new strong post-assertion.

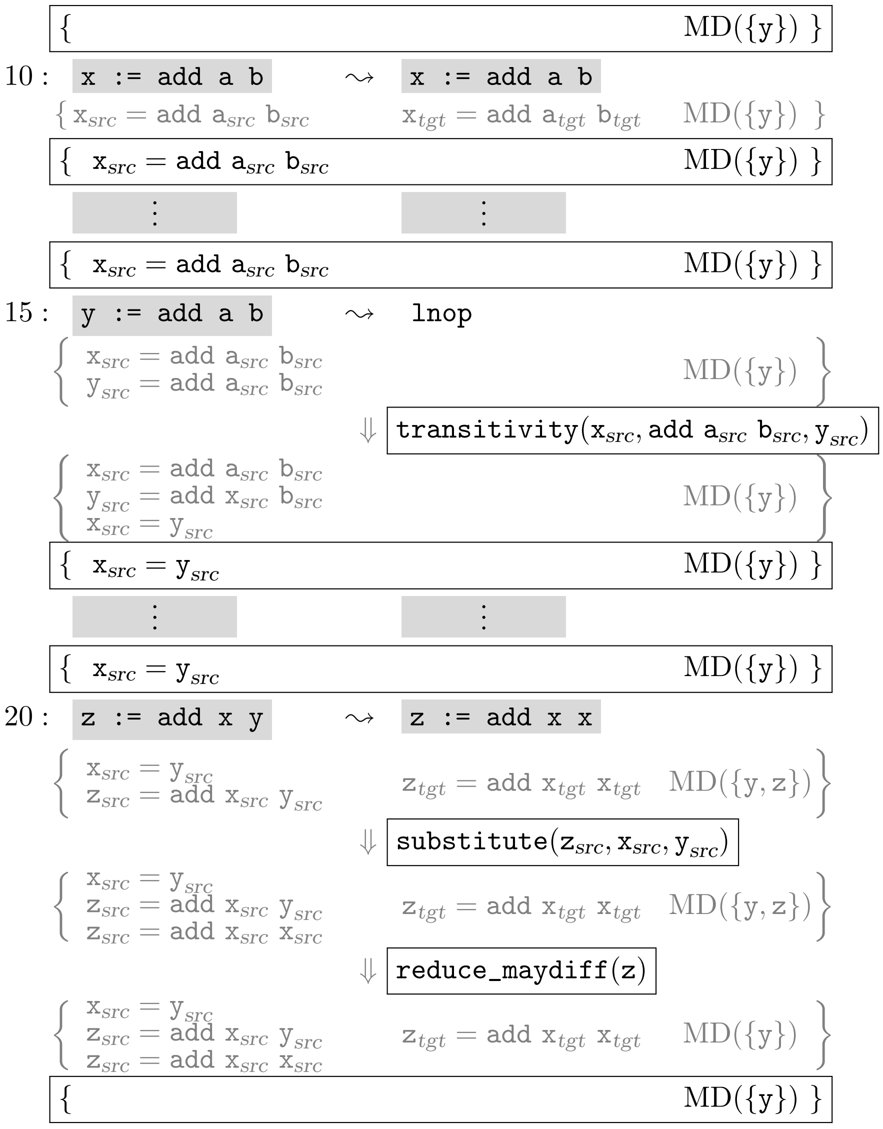
{ MD({y}) }
10 : x := add a b ⇝ x := add a b
{ $x_{src}$ = add $a_{src}$ $b_{src}$ $x_{tgt}$ = add $a_{tgt}$ $b_{tgt}$ MD({y}) }
{ $x_{src}$ = add $a_{src}$ $b_{src}$ MD({y}) }
⋮ ⋮
{ $x_{src}$ = add $a_{src}$ $b_{src}$ MD({y}) }
15 : y := add a b ⇝ lnop
{ $x_{src}$ = add $a_{src}$ $b_{src}$, $y_{src}$ = add $a_{src}$ $b_{src}$ MD({y}) }
⇓ transitivity($x_{src}$, add $a_{src}$ $b_{src}$, $y_{src}$)
{ $x_{src}$ = add $a_{src}$ $b_{src}$, $y_{src}$ = add $x_{src}$ $b_{src}$, $x_{src}$ = $y_{src}$ MD({y}) }
{ $x_{src}$ = $y_{src}$ MD({y}) }
⋮ ⋮
{ $x_{src}$ = $y_{src}$ MD({y}) }
20 : z := add x y ⇝ z := add x x
{ $x_{src}$ = $y_{src}$, $z_{src}$ = add $x_{src}$ $y_{src}$ $z_{tgt}$ = add $x_{tgt}$ $x_{tgt}$ MD({y, z}) }
⇓ substitute($z_{src}$, $x_{src}$, $y_{src}$)
{ $x_{src}$ = $y_{src}$, $z_{src}$ = add $x_{src}$ $y_{src}$, $z_{src}$ = add $x_{src}$ $x_{src}$ $z_{tgt}$ = add $x_{tgt}$ $x_{tgt}$ MD({y, z}) }
⇓ reduce_maydiff(z)
{ $x_{src}$ = $y_{src}$, $z_{src}$ = add $x_{src}$ $y_{src}$, $z_{src}$ = add $x_{src}$ $x_{src}$ $z_{tgt}$ = add $x_{tgt}$ $x_{tgt}$ MD({y}) }
{ MD({y}) }

Figure 3.1: Validation of Early CSE

3. **Automation for Applying Inference Rules** The proof checker can automatically apply simple inference rules, such as `transitivity` and `substitute`. Therefore, the code for applying these rules can be omitted from LLVM proof generation code. The proof checker makes a new assertion if some appropriate conditions holds for these rules.

**4. Reduce Maydiff** After adding strong post-assertions in the inference rule and automation step, the proof checker reduces the maydiff set by removing some registers. It compares the source and the target strong post-assertions and choose some registers to be removed. After `substitute` rule is applied at line 20, $z_{src} = \texttt{add}\ x_{src}\ \ x_{src}$ appears in the source. Since the target has condition $z_{tgt} = \texttt{add}\ x_{tgt}\ \ x_{tgt}$, and the register `x` is not in the maydiff set, the register `z` can be remove from the maydiff set.

**5. Hoare Triple Check** After all the steps have completed, the proof checker uses a simple inclusion test to validate the boxed assertions from the strong post-assertions. The boxed assertion after line 10 is valid because the strong post-assertion of line 10 has $x_{src}$ = `add` $a_{src}$ $b_{src}$. In line 15, the boxed assertion $x_{src}$ = $y_{src}$ implies the strong post-assertion by the inclusion. Finally, the boxed assertion in the last line has an empty assertion in the source and the target respectively, these are valid since they can be included in any set. All the maydiff sets of boxed assertions are also valid by the inclusion relation with the previous strong post-assertions.

## 3.3 Proof Generation

This section discusses proof generation code in the `EarlyCSE.cpp` file.

Algorithm 1 shows the pseudocode of Early CSE algorithm. SimplifyInstruction optimization is omitted for brevity's sake. The box contains the inserted proof generation code. Importantly, the proof generation code does not affect the original Early CSE optimization.

As in Figure 1.1, the proof checker requires a source IR file, a target IR file, and a proof. To validate optimization, the source file should be the file before optimization and the target should be after optimization. Thus, the source file has changed to the target file by optimization. We can select points during whole optimization process where should be source and target. Therefore, one validation which can hold only one optimization or several optimizations together like `Register Promotion`. For simple instructions optimization in the Algorithm 1, the source is pointed before line 13 and after line 17 for the target. From the experience of validating `Register Promotion`, validation of serval optimizations together is more difficult to validate [22]. A lot of validation units may cause performance problems(see Section 4.3 for details), but validation is simpler because only small amount of code changes occur. Therefore, in the Early CSE a validation is limited to a single optimization at a time, *e.g.,* deleting a redundant instruction and replacing it with the corresponding register is one validation unit.

In the following, we consider the simple instructions optimization, but the other optimizations proceed similarly.

Early CSE uses `AvailableValues` hash table to find an instruction has redundant rhs. It looks hash table by `AvailableValues.lookup` in line 13. If a matched rhs is found, then it returns the corresponding register $V$. All uses of the redundant instruction are replaced by the matching register in line 16 and the redundant instruction is deleted in line 17. At line 18, it inserts the register and its rhs into `AvailableValues` hash table when the rhs is new.

For validation, we need to know that replacing the uses of duplicated instruction is valid. In order to do so, the duplicated instruction and the existing instruction should have the same rhs.

First, if from the position of the duplicated instruction to all uses of the instruction, the assertion that the corresponding register and the duplicated instruction are identical is satisfied, we can replace the usages into the corresponding register. This assertion is inserted in line 15, $\boxed{\text{Assn}(i = V,\, l2,\, UseSet)}$, where $UseSet$ is the set of all usage of the instruction $i$ and $l2$ is line number of $i$. Using this assertion and automation for applying inference rules, the proof checker can apply the `substitute` inference rule and make the strong post-assertion $z_{src} = \texttt{add}\ x_{src}\ \ x_{src}$ in line 20 of Figure 3.1 .

Second, to check duplicated instruction and existing instruction has the same rhs which to satisfy the assertion after the deleted instruction, we needs information of the existing instruction. By looking at the hash table, we can get the existing instruction and propagate the assertion. The existing instruction and its rhs are propagated to the deleted instruction: $\boxed{\text{Assn}(V = rhs,\, l1,\, l2)}$, where $l1$ is line number of $V$. Because of automation, the proof checker can make the strong post-assertion using the `transitivity` rule that `a = b` in line 15 of Figure 3.1.

The assertion of the maydiff $\boxed{\text{Assn}(\{\text{MD}(i)\},\, \textit{global})}$ is also inserted because of the instruction deletion at line 17.

**Algorithm 1** Early CSE(Node:DomTreeNode)

```
 1: $BB := current\ BasicBlock$
 2: if BB's pred has BranchInst then
 3:     $ConditionValue := true$ or $false$
 4:     $\boxed{l1 = \text{line of } BranchInst,\ UseSet := \text{Dominate Usage Set of } BranchInst}$
 5:     $\boxed{\text{Assn}(BranchInst = true \text{ or } false,\ l1,\ UseSet)}$
 6:     ReplaceDominateUse($BranchInst$, $ConditionValue$)
 7: end if
 8: for ($l_i : i$) in Instr($BB$) do
 9:     match $i$ type with
10:     | $i$ is DCE ⇒
11:         eraseInst($i$) $\boxed{\text{generateProofForDCE(i)}}$
12:     | $i$ is simple instruction ⇒
13:         if $V$ := AvailableValues.lookup($i$) then
14:             $\boxed{rhs = \text{rhs of } V,\ l1 = \text{line of } V,\ l2 = \text{line of } i,\ UseSet := \text{Usage Set of } i}$
15:             $\boxed{\text{Assn}(V = rhs,\ l1,\ l2);\ \text{Assn}(i = V,\ l2,\ UseSet);\ \text{Assn}(\{\text{MD}(i)\},\ global)}$
16:             replaceInst($i$,$V$)
17:             eraseInst($i$)
18:         else AvailableValues.push($i$)
19:         end if
20:     | $i$ is load & non-volatile ⇒
21:         if $V$ := AvailableLoads.lookup($i$) & currentgeneration is same then
22:             $\boxed{rhs = \text{rhs of } V,\ l1 = \text{line of } V,\ l2 = \text{line of } i,\ UseSet := \text{Usage Set of } i}$
23:             $\boxed{\text{Assn}(V = rhs,\ l1,\ l2);\ \text{Assn}(i = V,\ l2,\ UseSet),\ \text{Assn}(\{\text{MD}(i)\},\ global)}$
24:             replaceInst($i$,$V$)
25:             eraseInst($i$)
26:         else AvailableLoads.push($i$)
27:         end if
28:     | $i$ is call & non-void type & readOnly ⇒
29:         if $V$ := AvailableCalls.lookup($i$) & currentgeneration is same then
30:             replaceInst($i$,$V$) $\boxed{\text{NotSupported}}$
31:             eraseInst($i$)
32:         else AvailableCalls.push($i$)
33:         end if
34:     end match
35:     if $i$ may write to memory then currentgeneration++
36:     end if
37:     if $i$ is DSE then
38:         eraseInst($i$) $\boxed{\text{NotSupported}}$
39:     end if
40: end for
```

# Chapter 4

# Results

This section shows the result of validation of Early CSE in LLVM 3.7.1. Proof generation code was inserted at `EarlyCSE.cpp` and validated with 5.40 million lines of C code benchmarks.

## 4.1 Implementation

To validate Early CSE, the proof checker and proof generation code in LLVM is necessary.

The proof checker is extended for automation to find inference rules for Early CSE and to support some instruction types that were not supported previously.

Table 4.1 shows the lines of the proof generation code. The entire `EarlyCSE.cpp` file is 522 lines long, and everything is covered except SimplifyInstruction optimization. The inserted code was 159 lines which is 30.46% of the original code length. The CRELLVM version which submitted in PLDI 2018 had infrastructure for proof generation consists of 1,708 lines for common library. Moreover, 72.2% of 15,980 lines for JSON serialization library can be automatically generated from 2,079 SLOC in a simple DSL [10]. For Early CSE proof generation, code

| | LOC of Original Compiler | Inserted Proof Generation Codes |
|---|---|---|
| `Early CSE` | 522 | 159 |

* 1,833 common library codes for proof generation infrastructure

Table 4.1: Lines of Original Compiler and Proof Generation Code

was inserted additional 125 lines for common library and 101 lines for JSON serialization library.

## 4.2 Validation Result

Benchmarks such as Spec2006 [4], LLVM Nightly Test(LNT) [27], and 7 other open source projects (a2ps-4.14, emacs-25.1, ghostscript-9.14.0, gimp-2.8.18, screen-4.4.0, sendmail-8.15.0 and python-3.4.1) [21] have been used. Table 4.2 shows the results of validations. Among total 2,091,997 validation, 97.13% were successful and 2.87% were not supported, and there was no failed validation. These outcomes were consistent with LLVM Bugzila, which showed there was no reported Early CSE mis-compilation bugs in LLVM 3.7.1 [6].

Of the 60,064 not supported validations, 82.29% were about instruction types not supported by `VELLVM`, 0.65% for Early CSE `CallInst` optimization and 3.59% for DSE. Moreover, there were 13.46% of not supported validations during Early CSE for `LoadInst`. However, this not supported validations occur not because of `LoadInst` but `VELLVM`.

`VELLVM` can not distinguish whether a call instruction is read only or not. For soundness, there is no guarantee that a pre-assertion holds in a strong post-assertion after a call instruction. There may have been a change due to the call instruction. Therefore, the proof checker does not maintain pre-assertion in the strong post-assertion after a call instruction. During Early CSE `LoadInst` optimization, `CallInst` occurring between two `LoadInsts` does not affect optimization when the `CallInst` is read only. However, since the proof checker cannot distinguish a read only call, the assertion cannot be hold, and validation fails. Thus, the proof checker cannot validate the case when `CallInst` is located between `LoadInsts` and left as not supported validation.

| | **LOC** | **Validation Result** | | |
|---|---|---|---|---|
| | | **#Success** | **#Failure** | **#Not Supp.** |
| `400.perlbench` | 168.16K | 90,349 | 0 | 534 |
| `401.bzip2` | 8.29K | 6,068 | 0 | 21 |
| `403.gcc` | 517.52K | 241,594 | 0 | 1214 |
| `429.mcf` | 2.69K | 689 | 0 | 0 |
| `433.milc` | 15.04K | 7,653 | 0 | 7 |
| `445.gobmk` | 196.24K | 48,999 | 0 | 248 |
| `456.hmmer` | 35.99K | 23,180 | 0 | 283 |
| `458.sjeng` | 13.85K | 8,653 | 0 | 169 |
| `462.libquantum` | 4.36K | 548 | 0 | 1209 |
| `464.h264ref` | 51.58K | 41,096 | 0 | 290 |
| `470.lbm` | 1.16K | 1,685 | 0 | 0 |
| `482.sphinx3` | 25.09K | 9,203 | 0 | 32 |
| `999.specrand` | 0.07K | 15 | 0 | 0 |
| `LLVM nightly test` | 1,358.76K | 513,590 | 0 | 4,360 |
| `a2ps-4.14` | 61.60K | 14,690 | 0 | 236 |
| `emacs-25.1` | 463.54K | 141,414 | 0 | 1,430 |
| `ghostscript-9.14.0` | 797.65K | 316,737 | 0 | 34,760 |
| `gimp-2.8.18` | 1,004.20K | 283,051 | 0 | 13,486 |
| `screen-4.4.0` | 47.74K | 29,772 | 0 | 425 |
| `sendmail-8.15.2` | 138.68K | 32,612 | 0 | 554 |
| `python-3.4.1` | 486.38K | 220,335 | 0 | 806 |
| Total | 5,398.59K | 2,031,933 | 0 | 60,064 |

Table 4.2: Validation Results

Similarly, Early CSE `CallInst` optimization cannot be validated, cannot distinguish read only call. Both cases can be validated if `VELLVM` updates to recognized read only calls, but for now these remain with not supported validation.

Another not supported optimization was DSE. The proof checker cannot validate the case when an memory location is still remained but one of the store instruction for the location is removed. The proof checker is possible to validate like `Register Promotion` optimization case, which erases the allocation instruction and all store instructions [22]. Therefore, changes in the proof checker would be required to validate this case, but since this occurred for only 0.1% of the total validations, and since it is extra optimization of Early CSE, this was left as not supported validation.

## 4.3 Performance Result

The experiment used an Intel Xeon E5-2630 CPU (2.6GHz, 12 cores with hyper-threading, 128GB RAM, and 1TB SSD Samsung 850 PRO). Table 4.3 shows the performance results. `Orig` is the time spent of Early CSE in the original compiler, *i.e.,* without proof generation. `PCal` is the time for Early CSE in the instrumented compiler including proof generation. `I/O` is for reading and writing the source, target and proof files. Lastly, `PCheck` is the time spent for the proof checker to validate the correctness of the proof.

The instrumented compiler is relatively slower than the original compiler. The result shows that `PCal` time is 16.45-fold comparing to `Orig` time. There can be possible reasons for this slowdown. One of them is using smart pointers which uses reference counting to free allocated memories automatically in C++. We use the smart pointers for convenience while writing proof generation code. When we tried to use normal pointers without automatic deallocation for validating `Register Promotion` [22], it turned that the `PCal` time had reduced to a half. We can know that he observed slowdown is partially due to the overhead of reference counting in smart pointers.

Another possibility might be because of too many validation units in Early CSE. While validating `Register Promotion` we found out that the initiation of each validation unit takes approximately 230 `ns` long which is much slower than other lines. The overhead can be mitigated when the initiation can be executed only once for many optimizations.

|  | LOC | Time (sec.) |  |  |  |
|---|---|---|---|---|---|
|  |  | Orig | PCal | I/O | PCheck |
| `400.perlbench` | 168.16K | 0.50 | 10.74 | 5.86K | 12.02K |
| `401.bzip2` | 8.29K | 0.02 | 0.78 | 0.22K | 0.53K |
| `403.gcc` | 517.52K | 1.57 | 40.11 | 20.82K | 16.18K |
| `429.mcf` | 2.69K | $< 0.01$ | 0.04 | $<$0.01K | $<$ 0.01K |
| `433.milc` | 15.04K | 0.04 | 0.49 | 0.08K | 0.11K |
| `445.gobmk` | 196.24K | 0.30 | 3.90 | 4.62K | 3.53K |
| `456.hmmer` | 35.99K | 0.12 | 1.69 | 0.81K | 0.39K |
| `458.sjeng` | 13.85K | 0.04 | 0.65 | 0.30K | 0.25K |
| `462.libquantum` | 4.36K | 0.02 | 0.12 | 0.02K | $<$ 0.01K |
| `464.h264ref` | 51.58K | 0.20 | 5.22 | 2.91K | 2.01K |
| `470.lbm` | 1.16K | $< 0.01$ | 0.10 | 0.01K | 0.02K |
| `482.sphinx3` | 25.09K | 0.06 | 0.69 | 0.10K | 0.14K |
| `999.specrand` | 0.07K | $< 0.01$ | $< 0.01$ | $<$ 0.01K | $<$ 0.01K |
| `LLVM nightly test` | 1,358.76.K | 3.40 | 70.44 | 34.26K | 48.81K |
| `a2ps-4.14` | 61.60K | 0.14 | 1.75 | 0.43K | 0.89K |
| `emacs-25.1` | 463.54K | 1.20 | 19.23 | 12.04K | 8.76K |
| `ghostscript-9.14.0` | 797.65K | 2.17 | 30.77 | 20.83K | 9.70K |
| `gimp-2.8.18` | 1004.20K | 2.62 | 23.34 | 12.27K | 6.79K |
| `screen-4.4.0` | 47.74K | 0.17 | 3.32 | 1.91K | 6.79K |
| `sendmail-8.15.2` | 138.68K | 0.22 | 2.76 | 1.91K | 1.82K |
| `python-3.4.1` | 486.38K | 1.43 | 17.86 | 26.20K | 9.22K |
| Total | 5,398.59K | 14.22 | 234.00 | 145.58K | 124.74K |

Table 4.3: Performance Results

# Chapter 5

# Related Works

A verified compiler provides a high quality of reliability. CompCert [12, 13] is a verified C complier of which correctness has been proved in Coq. The verified compiler provides high reliability but has high verification costs. In addition, performance is relatively low compared to production compilers. CompCert is 10% slower than GCC -O1, 15% slower than GCC -O2, and 20% slower than GCC -O3 [8].

CSmith [31, 7, 24] and EMI [11] are good random testing tools. They have found many GCC and LLVM bugs. However, no appearance of bug founded by CSmith and EMI does not guarantees reliability. Also, it considers a compiler as a black box without examining internal logic. Thus, it is not easy to find the source for any identified bugs.

Rinard *et al.* [26] proposed a framework of credible compilation and relational Hoare logic. It was able to prove the correctness of register allocation and instruction scheduling even in the presence of aliased pointers. However, it is only designed for a simple language, rather than any practical programming language.

Namjoshi *et al.* proposed a witnessing compiler [18, 19] which is an implementation of credible compilation. It targets LLVM compiler such as CRELLVM but is not yet mature because it only supports limited types of instructions, such as binary operation over integers, return, branch, compare and phi nodes. Also, the work only ensures the correctness for simple constant propagation, dead code elimination, and loop invariant code motion which are all possible in CRELLVM.

Verified translation validation is similar to verified credible compilation. Both need the input code and the generated code to a validator and the validator is independent from the compiler. Various verified validators have been developed for two instruction schedulings(list scheduling and trace scheduling) [28], lazy code motion [29], software pipelining [30], register allocation [25], SSA-based middle-end which converts CompCert intermediate form to SSA [3], GVN and sparse conditional constant propagation [9]. However, the verified translation validator is for one particular optimization, but CRELLVM can validate many other optimizations.

Proof-carrying code(PCC) [20, 1] is also proposed for validating untrusted code. Although PCC seems similar to credible compilation, it does not validate the translation. It also generates proofs for safety policy, but not for translation.

Alive [15, 17, 16] developed by Lopes *et al.* is also focusing on compiler correctness. To prevent mis-compilation, Alive was developed for correct LLVM optimizations, particularly to ensure correct peephole optimization in LLVM. Thus, Alive helps compiler writers to specify peephole optimizations, proves their correctness with satisfiability module theory(SMT) solvers [23] or provides counter examples, and automatically generates C++ code. Alive successfully translated more than 300 optimizations and identified 8 bugs in `InstCombine`. However, Alive can only support relatively uncomplicated algorithms, such as peephole optimization, and cannot address cyclic control flows.

# Chapter 6

# Conclusion

In this thesis, by using CRELLVM, Early CSE optimization which is one of basic optimizations in LLVM has been validated. The result of testing benchmarks shows 0 validation fail in Early CSE, implying there is no mis-compilation bugs in the benchmarks for Early CSE in LLVM 3.7.1. CRELLVM is an effective tool that can guarantee high reliability of compiler optimization.

***Limitation*** The optimization passes that CRELLVM can validate for now are `GVN, Register Promotion, LICM, InstCombine` and `Early CSE`. All of these optimizations do not modify the original control flow graphs. The current proof checker does not support modification of control flow graphs during optimizations, because source and target programs should be aligned to validate. However, people who participated in CRELLVM research are optimistic that the proof checker can be extended to support the change of control flow graphs.

The current proof checker depends greatly on `VELLVM` [32]. Therefore, the proof checker cannot support cases when `VELLVM` does not support it. As discussed previously, `CallInst` is not fully supported by `VELLVM`. Also, `VELLVM` does not fully supports LLVM instruction types. Since `VELLVM` does not fully supports

LLVM, `VELLVM` update should be a prerequisite to extend the proof checker. After completing development in `VELLVM`, the more optimization passes can be validated by CRELLVM.

# References

[1] Andrew W. Appel. Foundational proof-carrying code. *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*, June 2001.

[2] The Coq Proof Assistant. `https://coq.inria.fr/`.

[3] Gilles Barthe, Delphine Demange, and David Pichardie. Formal verification of an SSA-based middle-end for CompCert. *ACM Transactions on Programming Languages and Systems*, 36(1), March 2014.

[4] The SPEC CINT2006 Benchmark. `https://www.spec.org/cpu2006/CINT2006/`.

[5] Nick Benton. Simple relational correctness proofs for static analyses and program transformations. volume 39, January 2004.

[6] LLVM bugzilla. `https://bugs.llvm.org/`.

[7] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. Taming compiler fuzzers. *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 48(6), June 2013.

[8] CompCert. `https://compcert.inria.fr/`.

[9] Delphine Demange, David Pichardie, and Léo Stefanesco. Verifying fast and sparse SSA-based optimizations in Coq. *International Conference on Compiler Construction*, 9031, 2015.

[10] 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. Crellvm: Verified credible compilation for llvm. *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2018.

[11] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 49(6), June 2014.

[12] Xavier Leroy. Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. *Proceedings of the 33rd annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 41(1), January 2006.

[13] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7), July 2009.

[14] Xavier Leroy, Andrew W. Appel, Sandrine Blazy, and Gordon Stewart. The CompCert memory model, version 2. Research report RR-7987, INRIA, 2012.

[15] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. Provably correct peephole optimizations with alive. *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 50(6), June 2015.

[16] David Menendez and Santosh Nagarakatte. Alive-infer: Data-driven precondition inference for peephole optimizations in llvm. *Proceedings of the*

*38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 52(6), June 2017.

[17] David Menendez, Santosh Nagarakatte, and Aarti Gupta. Alive-fp: Automated verification of floating point based peephole optimizations in LLVM. *International Static Analysis Symposium*, 9837, August 2016.

[18] Kedar S. Namjoshi, Giacomo Tagliabue, and Lenore D. Zuck. A witnessing compiler: A proof of concept. *International Conference on Runtime Verification*, 8174, 2013.

[19] Kedar S. Namjoshi and Lenore D. Zuck. Witnessing program transformations. *International Static Analysis Symposium*, 7935, 2013.

[20] George Necula. *Proof-Carrying Code*, pages 984–986. Springer US, Boston, MA, 2011.

[21] Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, Daejun Park, Jeehoon Kang, and Kwangkeun Yi. Global sparse analysis framework. *ACM Transaction on Programming Languages and Systems*, 36(3), September 2014.

[22] Sanghoon Park. Verified translation validation for register promotion in llvm. *Master Thesis*, June 2018.

[23] The Z3 Theorem Prover. `https://github.com/Z3Prover/z3`.

[24] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-case reduction for C compiler bugs. *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 47(6), June 2012.

[25] Silvain Rideau and Xavier Leroy. Validating Register Allocation and Spilling. *International Conference on Compiler Construction*, 6011, 2010.

[26] Martin C. Rinard and Darko Marinov. Credible compilation with pointers. RRV '99, 1999.

[27] LLVM Test Suite. `http://llvm.org/docs/TestingGuide.html#test-suite`.

[28] Jean-Baptiste Tristan and Xavier Leroy. Formal verification of translation validators: A case study on instruction scheduling optimizations. *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 43(1), January 2008.

[29] Jean-Baptiste Tristan and Xavier Leroy. Verified validation of lazy code motion. *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 44(6), June 2009.

[30] Jean-Baptiste Tristan and Xavier Leroy. A simple, verified validator for software pipelining. *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 45(1), June 2010.

[31] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 46(6), June 2011.

[32] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 47(1), January 2012.

# 요약

소프트웨어 검증을 통해 높은 신뢰성을 얻고자 할 때, 컴파일러 검증은 매우 중요하다. 실행되는 프로그램은 소스코드가 아닌 실행코드이므로 컴파일러 버그는 소프트웨어 검증에 치명적이다. 그러나 GCC 와 LLVM 같은 많은 C/C++ 주류 컴파일러들은 성능향상에 집중하며 높은 신뢰성에는 상대적으로 소홀한 편이다. 테스팅은 버그를 찾는데 도움이 되지만 이 것만으로는 높은 신뢰성을 보장할 수 없다. 컴파일러 내부 로직을 확인하는 많은 방법이 제안되었지만 아직 성공적으로 적용된 것은 없다.

CRELLVM 은 LLVM 최적화 패스를 검산하여 최적화의 높은 신뢰성을 제공하는 컴파일러 프레임워크이다. CRELLVM 은 LLVM 내의 주요 최적화인 Register Promotion, Global Value Numbering 등을 검산할 수 있다.

이 논문은 CRELLVM 을 이용하여 LLVM 최적화인 Early CSE 를 검산하는 과정을 보여준다. 검산을 위해 LLVM 의 Early CSE 최적화 코드에 대응하는 증명 생성 코드를 구현하고 증명 확인기를 확장하였다. Early CSE 는 중복된 명령어를 지워서 반복 계산을 제거하는 최적화로 LLVM 내의 기본적인 최적화 중 하나이다.

540 만 C 코드 벤치마크를 이용한 실험 결과에서 Early CSE 내의 잘못된 컴파일 결과가 없다는 것을 확인하였고, 이 결과는 벤치마크 내에서 Early CSE 의 신뢰성을 보증한다.

**주요어**: 검증되고 신뢰할 수 있는 컴파일러, 컴파일러 검증, LLVM 컴파일러, CRELLVM, Early CSE

**학번**: 2016-21215