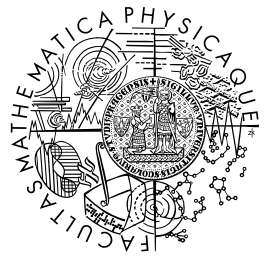

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Jakub Kubík

Loop Analysis for LLVM IR Translation Validation Framework

Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis:	doc. RNDr. Jan Kofroň, Ph.D.
Study programme:	Computer Science
Study branch:	General Computer Science

Prague 2024

I declare that I carried out this bachelor thesis on my own, and only with the cited sources, literature and other professional sources. I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I would like to thank my advisor doc. RNDr. Jan Kofroň, Ph.D. and my consultant Mgr. Martin Blicha, Ph.D. for their valuable advice and patience. My thanks also goes to Nuno P. Lopes, the creator of Alive2, for being extremely helpful in the process and for creating such an amazing tool. Last but not least, I would like to thank my parents for being my parents.

Title: Loop Analysis for LLVM IR Translation Validation Framework

Author: Jakub Kubík

Department: Department of Distributed and Dependable Systems

Supervisor: doc. RNDr. Jan Kofroň, Ph.D., Department of Distributed and Dependable Systems

Abstract: Bugs in compilers can have severe consequences. Apart from traditional methods like testing, one of the ways of keeping compilers correct that gained traction only in recent years is *translation validation*, a technique ensuring the semantic correctness of optimizations in compilers. *Alive2* is an open-source translation validation framework for LLVM that is currently widely used by LLVM developers. In order to make any static analysis tool usable, the frequency of false alarms must be kept to a minimum. Alive2 was designed to have zero false alarms and has been very successful in this endeavor except in the case of certain loops. Our aim in this thesis is to analyze Alive2's loop algorithms in an attempt to find the cause of these false alarms. This was motivated by personal communication with authors of Alive2 who presented the false alarm issue in loops as one of the more challenging and pressing issues in Alive2. We were successful in pinpointing the cause of false alarms and even providing a fix for the issue. Our solution is now a part of the Alive2 framework. Furthermore, we have identified other potential issues in Alive2 which we discuss in the thesis as well.

Keywords: compilers LLVM translation validation formal verification

Contents

Introduction

Software engineers rely on the correctness of their compilers. In a typical scenario, a compiler transforms a program many times over in order to make it as efficient as possible. Hence, it is not obvious that the final program is equivalent to the original one. In 2015, a bug in LLVM was exploited (for academic purposes) to create a backdoor in Linux `sudo` [1]. A slightly amusing (but nonetheless serious) example is a compiler miscompiling itself which resulted in miscompilation of another program[1]. Many other such examples that motivate researchers and compiler engineers to take compiler correctness seriously can be found in literature [2] [3] [4].

With the advent of large language models (LLMs), verifying software is now more crucial than ever [5]. Consider the classical prompt "make my program faster". We may get a program that is more efficient but how do we know it is equivalent to the input program? This example aptly illustrates the task of *translation validation*. Translation validation is a technique that tries to determine whether a given input program is equivalent to the transformed program.

Alive2 [6] is a highly successful translation validation framework for LLVM as witnessed by the number of bugs it has found in LLVM [7]. It has also been selected for the Distinguished Paper Award at PLDI'21[2] and it is now even a part of a recent guide for new contributors to LLVM [8].

One of the most important factors that makes a static analysis tool such as Alive2 practically usable is the absence of false alarms which means avoiding reporting an incorrect transformation when the transformation was actually correct. Reporting false alarms is one of the factors preventing verification tools from being widely adopted among developers [9]. Thus, Alive2 was designed with this goal in mind from its inception. However, it was not always successful in this endeavor in the presence of loops. Loops are notoriously difficult to deal with in the area of software verification but loop optimizations are crucial for modern compilers so it is necessary to handle them. The aim of this thesis is to analyze Alive2's loop algorithms and attempt to pinpoint and fix the cause of

[1] `https://lists.llvm.org/pipermail/llvm-dev/2017-July/115497.html`

[2] `https://pldi21.sigplan.org/track/pldi-2021-papers`

these false alarms.

Our goals and thesis structure

Despite its aim to be free of false alarms, Alive2 reported them in the case of certain nested loops. Finding the root cause of these alarms is a difficult task as witnessed by the long-standing unsolved issues in Alive2 codebase related to false alarms[3,4]. Our work was motivated by personal communication with Nuno P. Lopes, the author of Alive2, who mentioned false alarms in loops as one of the more pressing and difficult problems in Alive2 at the time. Tackling this problem required deep understanding of Alive2's loop algorithms and the difficulty was exacerbated by the fact that false alarms manifested seemingly arbitrarily.

The aim of this thesis is to study loop analysis algorithms in Alive2, try to find and possibly fix the cause of false alarms with nested loops present in Alive2. Not only have we achieved these goals, but through our analysis we discovered that the issue was present in an even broader class of programs, namely all programs containing nested loops. Nevertheless, in most programs this issue lay dormant which could have posed even greater problems in the future. Furthermore, we provided a fix for this problem and our solution is now a part of the Alive2 framework. We have also encountered other issues in Alive2 which we briefly discuss at the end and thus providing solid ground for future work in this area.

The thesis is divided into six parts:

- Chapter 1 provides the theoretical background necessary to understand the problem.
- Chapter 2 gives an overview of LLVM and Alive2 and provides a detailed description of how the concepts discussed in the previous chapter are implemented in Alive2.
- Chapter 3 provides detailed analysis of loop algorithms in Alive2 and explains the core of the issue causing false alarms in Alive2.
- Chapter 4 describes why the issue described in Chapter 3 caused false alarms and demonstrates the problem on a real example from the LLVM test suite.
- Chapter 5 presents our solution to the problem and the implementation as well as concrete results demonstrating how our fix helped eliminate false alarms in LLVM.

[3] `https://github.com/AliveToolkit/alive2/issues/748`
[4] `https://github.com/AliveToolkit/alive2/issues/762`

- Chapter 6 describes further results we obtained from our work and and mentions possible future work in this area.

Chapter 1

Theoretical background

In this chapter, we present the theoretical background necessary to understand this thesis. Our topic is at the intersection of compilers and verification of software, thus firstly we need some mathematical preliminaries and then we provide an introduction to the area of compilers and compiler correctness.

1.1 Preliminaries

We assume the reader is familiar with set notation, basic set operations, functions and relations. We will denote the set $\{1, \dots, n\}$ as $[n]$. $\binom{X}{k}$ denotes the set of all k-element subsets of X. We assume all our sets are finite unless otherwise specified.

One of the most important structures in our work is that of a *graph*. The area of graph theory is one of the most established parts of mathematics within computer science. We expect the reader to be familiar with the foundations of graph theory, so we will just quickly go through the definitions and our assumptions. A *directed graph* is a pair $G = (N, E)$, where N is a set of nodes[1] and $E \subseteq N \times N$ is the set of edges. For ease of notation, we denote the directed edge (u, v) simply as uv. An *undirected graph* is an analogical concept, except its edges are members of the set $\binom{N}{2}$. Our graphs will be strictly directed unless stated otherwise.

1.1.1 Depth-first search

Depth-first search (DFS) is one of the most basic graph traversal algorithms. The basic idea of DFS is that we visit a node's neighbor, recursively perform DFS on this neighbor and then do the same for other neighbors, starting from a given

[1] In most parts of graph theory, these are referred to as *vertices* but in compiler theory it is more common to call them nodes

root node. The edges that DFS uses to visit yet unvisited nodes are called *tree edges* because they form a spanning tree of graph which is called the *DFS tree*. The root of the DFS tree is the starting node of DFS. We say that a node u is an *ancestor* of a node v if u is on some path from the root to v. An edge uv is called a *backedge* if v is an ancestor of u in the DFS tree.

Preorder numbering

Preorder traversal of a tree is a type of tree traversal where we visit the root node first and then recursively traverse all its subtrees from the left.

Definition 1 (Preorder numbering)**.** Preorder numbering *of a graph is the numbering of its nodes by preorder traversal of its DFS tree.*

We can obtain a preorder numbering directly from the definition, that is by a simple modification of DFS [10]:

Algorithm 1 Preorder numbering of a directed graph

```
1:  function PreOrder(G: a directed graph)
2:      currentPreorder ← 1
3:      preorderNumber[∗] ← None
4:      function Visit(u: starting node)
5:          preorderNumber[u] ← currentPreorder
6:          currentPreorder ← currentPreorder + 1
7:          for each unvisited neighbor v of u do
8:              Visit(v)
9:          end for
10:     end function
11:     Visit(u_0) ..... u_0 is a predetermined starting node
12: end function
```

Using preorder numbering, we can efficiently test ancestry in a DFS tree by saving each node's last descendant after the for-loop on line 7. Let m, n be the preorder numbers of nodes u, v respectively and l the preorder number of u's last descendant. Then u is an ancestor of v if and only if $m \leq n$ and $n \leq l$.

Topological sorting

A very important class of directed graphs is the class of *directed acyclic graphs* (commonly abbreviated as DAGs) which are directed graphs without directed cycles. Their main feature is that they admit a total ordering on nodes commonly called *topological ordering*.

Definition 2 (Topological ordering [11])**.** *If G is a DAG, then* topological ordering *of the nodes of G is a total order $\prec$ such that $uv \in E \implies u \prec v$.*

We can obtain a topological order of our DAG by using a topological sorting algorithm. One of the most standard ones is due to Tarjan [12]:

Algorithm 2 Tarjan's topological sorting algorithm

```
1:  function TopSort(G: a DAG)
2:      S ← an empty stack
3:      function Visit(u: node to be processed)
4:          if u is unvisited then
5:              Mark u as visited
6:              for every uv ∈ E do
7:                  Visit(v)
8:              end for
9:              Push u onto S
10:         end if
11:     end function
12:     for every node u ∈ V do
13:         Visit(u)
14:     end for
15:     return S
16: end function
```

Upon termination, S will contain all the nodes of G in topological order. We can see that this is just a version of DFS alternate to the one we used in preorder numbering.

1.2 Compiler theory

1.2.1 Control-flow graphs and dominators

One of the most fundamental concepts in analysis of programs is that of a *control-flow graph* (commonly abbreviated as *CFG*). In order to define it, we must first introduce the notion of a *basic block*:

Definition 3 (Basic block [13])**.** *A sequence of instructions in a program P forms a* basic block *if the instruction in each position always executes before all those in later positions and no other instruction executes between two instructions in the sequence.*

In particular, we can see that in order to guarantee sequential execution, there can be no control flow jumps in the middle of the basic block. Sometimes

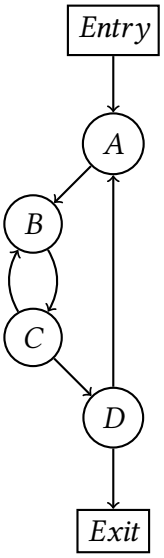

Figure 1.1 An example CFG

basic blocks are assumed to be maximal in the sense that an instruction starts a new basic block only if it is involved in branching of some kind, but this is not necessarily always true in practice. This is best seen in the context of a control-flow graph:

Definition 4 (Control-flow graph [13])**.** *A* control-flow graph *for a program P is a directed graph $G = (N, E)$, where:*

- *N is the set of all basic blocks of P, including two special basic blocks called Entry and Exit,*
- *E represents the control flow jumps between basic blocks.*

In Figure 1.1 we can see an example of a control-flow graph.

It is useful to ask if some basic block is guaranteed to have been reached prior to some other basic block in a given CFG. For example in our CFG, the basic block B is definitely always reached before C as there is no path from *Entry* to C which does not include B. This gives rise to the concept of *dominance*:

Definition 5 (Dominance [13])**.** *We say that basic block A* dominates *basic block B if A is on every path from Entry to B. We denote this as A* dom *B. If $A \neq B$, then we say that A* strictly dominates *B, denoted as A* sdom *B. A is an* immediate dominator *of B if A strictly dominates B and does not strictly dominate any other basic block that strictly dominates B.*

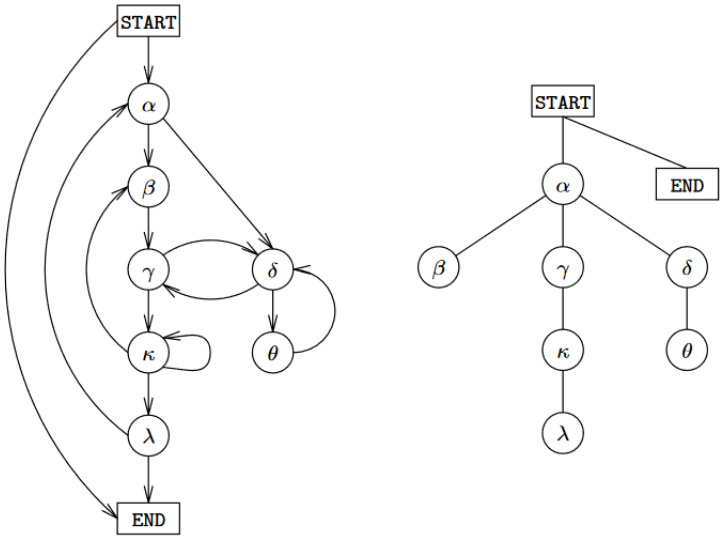

Figure 1.2 An example CFG and its corresponding dominator tree [10]

We can see that *Entry* always dominates all basic blocks and every basic block dominates itself. It can also be easily seen that dominance is a partial order. Hence, we can obtain a *dominator tree* by making the immediate dominator of every basic block its parent. The dominator tree can be built in almost linear time [14].

1.2.2 Loops

A loop is a very common programming construct that allows the execution of a sequence of code multiple times. Using the control-flow graph, we can define a loop independent of the programming language, though looping structures in a CFG can be defined with varying levels of generality. We will firstly define something called a *natural loop* (Definition 7) and then in Definition 8, we provide a fully fledged definition of a loop that we shall use throughout the text. Before providing the definitions, we need to distinguish between two types of backedges:

Definition 6 (Reducible and irreducible backedge [10])**.** *We call uv a* reducible backedge *if v dominates u. If uv is a backedge with respect to a given DFS tree and it is not a reducible backedge, we call uv an* irreducible backedge.

Notice that reducible backedges do not depend on the choice of the DFS tree as opposed to irreducible backedges. We can now provide the definition of a *natural loop*:

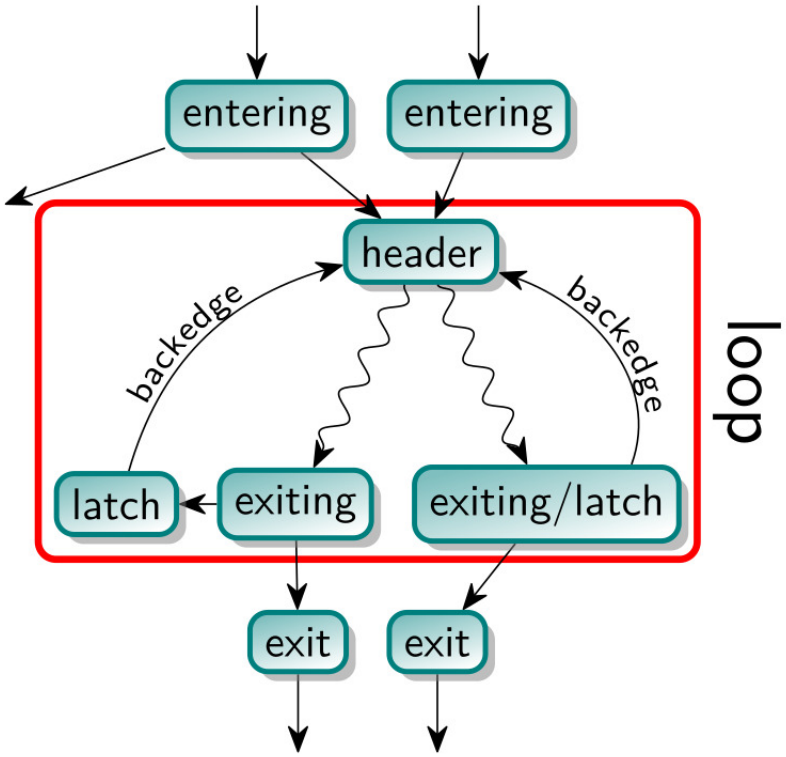

Figure 1.3 Illustration of loop terminology [17]

Definition 7 (Natural loop). *Let $vh \in E$ be a reducible backedge in a control-flow graph $G = (N, E)$. A* natural loop *is a triple $\ell = (h, B, X)$[2] such that $B = \{b \in N \mid h \text{ sdom } b \wedge v \text{ is reachable from } b \text{ without passing through } h\}$ and $X \subseteq N \setminus B$ is the set of* exit blocks *where $x \in N$ is an* exit block *if $x \notin \ell$ but there is a $y \in \ell$ such that $yx \in E$. We call h the* header *of ℓ and B the* body *of ℓ. The node v in backedge vh is called a* latch.

This definition is adapted from Muchnick's definition [15], but the formalization is inspired by de Vos [16]. Nevertheless, while de Vos also defines a loop as a triple $\ell = (h, B, X)$, in his formalism X is the set of *exiting blocks*. A basic block u is an *exiting block* if there is an edge $uv \in E$ such that v is an exit block. See Figure 1.3 for an illustration of the loop terminology.

Natural loop is the looping structure that occurs most often in real-world programs and most of the loops in this text are natural loops. Nevertheless, we will need a more general definition for some purposes. The most general definitions of looping structure in a CFG rely on the concept of strong connectivity [15]:

[2]For ease of notation, we will also denote the set $\{h\} \cup B$ as ℓ.

Definition 8 (Loop [10])**.** *A* loop *in a control-flow graph $G = (N, E)$ is a triple $\ell = (h, B, X)$ such that $G[\ell]$ is a maximal strongly connected subgraph*[3] *of G with at least one edge. For a given DFS traversal of G, h is the first node in ℓ encountered by DFS and it is called the* header *of loop ℓ. B is called the* body *of ℓ and X is the set of exit blocks as defined in Definition 7. If $\ell = (h, B, X)$ is a loop in G, then any loop ℓ' in $G[B]$ is also a loop in G and we say ℓ' is* nested *in ℓ. If a loop is not nested in any other loop, we call it the* outermost loop *and if no loop is nested inside a loop, we call it the* innermost loop.

The definition is a little involved, but the general idea is quite simple. We can see that a loop must also have a backedge (and thus also a latch), though it need not be reducible as is the case with natural loops. Also note that the definition is recursive; we repeatedly remove the header of a loop until we get to the innermost loop.

In Figure 1.1, we have two loops:

- $\ell_1 = (h_1, B_1, X_1)$, where $h_1 = A$, $B_1 = \{B, C, D\}$, $X_1 = \{exit\}$, and D is the latch,
- $\ell_2 = (h_2, B_2, X_2)$, where $h_2 = B$, $B_2 = \{C\}$, $X_2 = \{D\}$, and C is the latch.

The main difference between our definitions of a loop and a natural loop lies in the concept of *reducibility*:

Definition 9 (Reducibility [10])**.** *A basic block in a loop ℓ is an* entry *of ℓ if it can be reached from Entry without passing through any other blocks in ℓ. We say that a loop is* reducible *if it has a single point of entry, otherwise it is* irreducible*. A control-flow graph is* irreducible *if it contains at least one irreducible loop, otherwise it is* reducible.

Reducibility is a very significant property in compiler optimization because irreducible control-flow graphs are hard to analyze. Fortunately, irreducible loops are rare as they require unstructured control flow, such as using `goto`-like constructs.

The provided definition of a natural loop corresponds to a reducible loop as we have just defined it. As we have mentioned already, most of the loops in our text are natural loops and thus reducible, so the reader is free to refer to Definition 7, we will explicitly mention whenever we are dealing with irreducibility. Nevertheless, the algorithms we analyze in Chapter 3 work with this general definition and in Chapter 6, we discuss some repercussions of irreducibility.

[3] *Maximal* in this sense means that no more nodes can be added such that the subgraph will still remain strongly connected

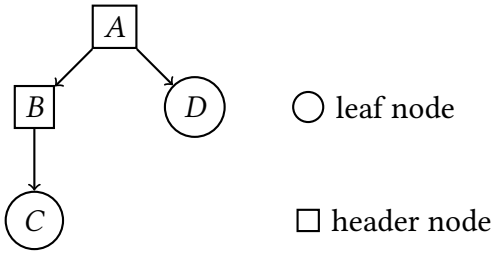

Figure 1.4 The loop-nesting tree of Figure 1.1

If h is the header of only one loop, we also denote this unique loop as $\ell(h)$ for ease of notation. In our example in Figure 1.1 with loops ℓ_1 and ℓ_2 we can see that ℓ_2 is immediately nested in ℓ_1. The nesting relation can be represented by a forest (a collection of trees):

Definition 10 (Loop-nesting forest)**.** *A* loop-nesting forest *for a CFG $G = (N, E)$ is a forest $\mathcal{F} = (\mathcal{L}, F)$, $\mathcal{L} \subseteq N$, consisting of rooted trees such that for every outermost loop $\ell = (h, B, X)$ we have a tree T which we define inductively:*

- *h is the root of T*
- *If u is a loop header, then v is a direct descendant of a node u if and only if:*
  - *v is a loop header and $\ell(v)$ is nested in $\ell(u)$ (we call those nodes* header nodes*), or*
  - *v is contained in the body of $\ell(u)$ and it is not inside any other loop (we call those* leaf nodes*).*
- *If u is not a loop header, it can only be a leaf node*

The loop-nesting forest of our running example consists of only a single tree as seen in Figure 1.4.

1.3 Compiler correctness

Optimizing compilers put a lot of work into transforming code such that the resulting program runs as fast as possible. Nevertheless, these optimizations must also preserve the semantics of the original program. Consider these two programs [18]:

Original

```
int a = x << c;
int b = a / d;
return b;
```

Optimized

```
int t = d / (1 << c);
int b = x / t;
return b;
```

It may not be immediately clear why this would be an optimization but it can be quickly shown why the two programs are *algebraically* equivalent:

$$b_{\text{Optimized}} = x/t = x/(d/2^c) = x \cdot \frac{2^c}{d} = b_{\text{Original}}$$

Nevertheless, in the optimized program there is an additional division by `1 << c`, that is, by 2^c mathematically speaking. Though in mathematics 2^c can never be equal to 0 for any value of $c \in \mathbb{Z}$, in C or C++ the integer value can overflow and become 0 and thus the optimized program introduces undefined behavior that was not present in the original program. C and C++ are standardized languages and anything that is not defined by the standard is called *undefined behavior* which means compilers are free to transform the code in any way they desire[4]. This example is often mentioned because it illustrates how easy it is to get an optimization wrong. While the transformation does indeed seem semantics-preserving, it was actually a bug in LLVM[5].

There are several approaches to catching bugs in software including testing and static analysis. An alternative approach that has become practical only in recent years is *formal verification*. Formal verification is a field of computer science that studies techniques for proving programs formally correct. One of the things that make verification of software challenging is *loops* (other things include recursion, thread creation and dynamic memory allocation). It can be quite easily proved by reduction to the halting problem that the problem of equivalence of two functions is undecidable. There is a sub-area of formal verification that focuses solely on compiler correctness which studies whether or not is the input program equivalent to the program after compilation. Compiler verification is especially hard because a compiler is typically a very complex piece of software and verifying an industrial-strength compiler is currently all but impossible. One approach is to build a compiler from scratch with some form of verification that is integrated from the very beginning. The most famous (and quite recent) example of this is *CompCert* [19], a formally verified optimizing C compiler. CompCert utilizes machine-assisted mathematical proofs using Coq theorem prover [20], it has been experimentally tested to demonstrate it can generate efficient and compact code. The authors of CompCert, led by Xavier Leroy of Collège de France, even received the ACM Software System Award for "development of the first industrial-strength optimizing compiler that has been the subject of a complete, mechanically checked proof of correctness"[6]. Nonetheless, even such a successful project as CompCert has several deficiencies. First of all, it does not support the

[4] `https://raphlinus.github.io/programming/rust/2018/08/17/undefined-behavior.html`

[5] `https://bugs.llvm.org/show_bug.cgi?id=21245`

[6] `https://awards.acm.org/award-recipients/leroy_4273298`

entire language specification of C, so for example writing system-level code may not always be possible. Secondly, CompCert does not provide many optimizations, definitely not enough to compete with LLVM or GCC. And third of all, the theory behind CompCert is not complete yet so CompCert does not guarantee what happens when the program runs out of memory, for example. These concerns show that it is still uncertain how to create a bulletproof specification for a formally correct compiler.

1.3.1 Translation validation

An alternative approach to constructing formally verified compilers is to relax the requirements. It is not strictly necessary to have a fully-verified compiler, it is sufficient to automatically verify every transformation the compiler makes when provided with an input program. This gives rise to the concept of *translation validation*, a technique fully developed by Pnueli et al. [21] but first proposed by Samet [22]. Formal verification requires the program to be correct for all inputs, translation validation on the other hand only requires the program to be correct for the provided input. This immensely expands the area of applicability since it is tenable to retrofit soundness even to industrial-strength compilers. We will describe how this is achieved in Alive2 in the following chapter.

1.3.2 Summary

To summarize, we have two main approaches of formally ensuring compiler correctness:

1. *Formal verification*

   Formal verification strives to prove the compiler correct for *all* input programs, primarily with the use of theorem provers. A prominent example of this is CompCert.

2. *Translation validation*

   With translation validation, every transformation the compiler makes (*translation*) is followed by a verification step (*validation*) which says whether or not this particular transformation on the given input was correct. As opposed to formal verification, we do not require the compiler to be completely correct and so translation validation applies to a wider range of practical problems.

Chapter 2

Overview of LLVM and Alive2

In this chapter, we provide a technical and theoretical introduction to LLVM as well as Alive2.

2.1 LLVM

LLVM[1] [23] is an open-source framework for compiler and toolchain technologies. It is built around *LLVM IR* (Section 2.1.1), an intermediate language for LLVM-based compilers. All middle-end optimizations within LLVM are performed on LLVM IR.

2.1.1 LLVM IR

Some form of intermediate representation (IR) is a necessary part of any modern compiler. In order to understand the importance of IR, it is instructive to see it in the broader context of the compiler. On a high level, a compiler comprises of three parts:

1. Front-end: analyzes the source code written in a concrete programming language and builds the IR
2. Middle-end: expects IR from front-end and performs optimizations on it
3. Back-end: generates platform-specific machine code and performs CPU-specific optimizations

An intermediate representation strives to achieve the "perfect balance" between the programming language and computer code. This means that IR needs

[1] LLVM previously stood for *Low-Level Virtual Machine* but it is no longer used as an acronym

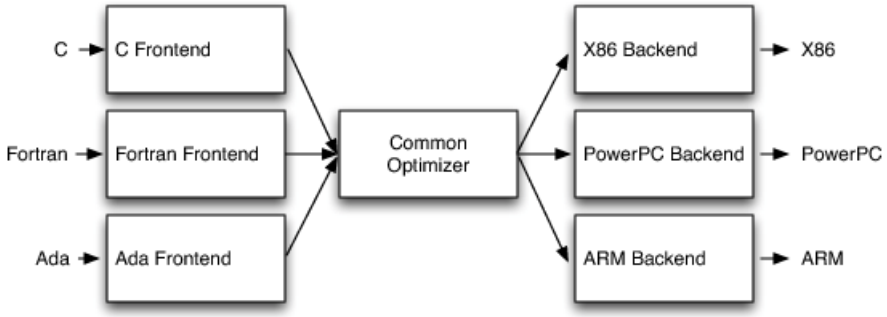

Figure 2.1 A high-level view of a LLVM's design [24]

```
void test(int n) {
    for (int i = 0; i < n; i ++)
            // Loop body
}
```

Figure 2.2 An empty for-loop in C

to be designed to make the transformation from the high-level language to the IR, as well as the transformation from IR to the machine code, reasonably easy to perform.

LLVM Intermediate Representation (LLVM IR) [25] is the crux of LLVM's design [24]. It was designed to support lightweight runtime optimizations as well as aggressive restructuring transformations and interprocedural optimizations. But a particularly important aspect of LLVM IR is that it is itself a language with well-defined semantics. It supports simple instructions like add (`add`), subtract (`sub`), compare (`cmp`), and branch (`br`). These instructions are in three-address form, which means that they take some number of inputs and produce a result in a different register. In contrast to most real instruction sets, LLVM IR is strongly typed (e.g. `i32` is a 32-bit integer) and some machine-specific details are abstracted away (e.g. calling convention). [24]

To give an example, let us consider an empty for-loop in C in Figure 2.2. This would correspond to the LLVM IR code in Figure 2.3.

Every label defines a basic block, in particular, `entry` and `exit` are exactly *Entry* and *Exit* basic blocks in the definition of the control-flow graph (Definition 4). Let us describe every basic block in more detail:

1. `for.header` exactly corresponds to the `int i = 0; i < n;` part of the `for` statement

    - the `phi` instruction corresponds to the ϕ function in SSA (see Section 2.1.1), thus it returns 0 if the preceding block was `entry` or

```
define void @test(i32 %n) {
entry:
  br label %for.header

for.header:
  %i = phi i32 [ 0, %entry ], [ %i.next, %latch ]
  %cond = icmp slt i32 %i, %n
  br i1 %cond, label %body, label %exit

body:
  ; Loop body
  br label %latch

latch:
  %i.next = add nsw i32 %i, 1
  br label %for.header

exit:
  ret void
}
```

Figure 2.3 An empty for-loop in LLVM IR [17]

`i.next` if the preceding block was `latch`

- `icmp slt i32` is a comparison instruction on two, signed 32-bit integers `%i` and `%n`
- `br` takes the condition checked above and if indeed `%i` is less than `%n`, it jumps to `body`, otherwise it goes to `exit`

2. `body` represents the body of the loop but it is empty so it only jumps to the following basic block

3. `latch` simply increments the `%i` variable, saves it to `%i.next` and jumps back to `for.header`

We can see the CFG of the LLVM IR code in Figure 2.4.

The textual representation in Figure 2.3 is not the only format of LLVM IR, it is actually defined in three different forms:

1. textual format (an `.ll` file in Figure 2.3)

2. in-memory data structure (used by the optimizer)

3. binary bitcode format (a `.bc` file readable by tools like `hex`)

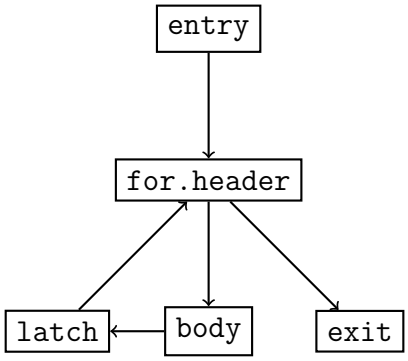

Figure 2.4 CFG of empty loop in LLVM IR

```
if (foo)
    X = 42;
else
    X = 101010;
return X;
```

Figure 2.5 Conditional branching

Static single-assignment form

Like most intermediate representations of today, LLVM IR is written in the so-called *static single-assignment form*, which we shall now define:

Definition 11 (Static single-assignment form)**.** *We say that a program is in* static single-assignment form *(commonly abbreviated as* SSA form*) if every variable is defined before it is used and assigned exactly once.*

SSA form guarantees for instance that the definition of a variable dominates all its uses, which is necessary for many optimizations based on SSA. Nevertheless, real-world programs are rarely in SSA form. Consider the program in Listing 2.5. This program is not in SSA form because the variable `X` is assigned twice (once to 42 in the true branch and to 101010 in the false branch). To convert this program to SSA form, we need to distinguish the uses of `X` by renaming it (so-called *SSA names*), but we also need to know which of the values to return. This is typically achieved by introducing the so-called *ϕ-function*. The ϕ-function takes SSA names as arguments and returns the value depending on which branch the variable came from. The resulting program in SSA is shown in Listing 2.6.

SSA is commonly viewed as a property of intermediate representation and even though it can be applied to any program[2] (and thus can even be inter-

[2]In this sense, it is very akin to functional programming [26]

```
if (foo)
    X1 = 42;
else
    X2 = 101010;
X3 = phi (X1, X2)
return X3;
```

Figure 2.6 Conditional branching after introducing ϕ nodes

preted [27]) it is not designed for direct execution. The message of SSA is that "having distinct names for distinct entities reduces uncertainty and imprecision" [28].

The majority of current commercial and open-source compilers use SSA-based IR, including GCC [29], LLVM [25], the HotSpot Java virtual machine [30], and the V8 JavaScript engine [31].

2.1.2 LLVM Passes

In LLVM, optimizations are implemented as so-called *passes* that traverse a portion of the program to either collect information about the program or transform it in some way. There are three categories of passes in LLVM [32]:

1. *Analysis passes* compute information that other passes can use or for debugging or program visualization purposes,

2. *Transform passes* mutate the program in some way, they can also use or invalidate the analysis passes,

3. *Utility passes* provides some utility but do not otherwise fit categorization

2.2 Alive2

Alive2 [6] is a translation validation framework for LLVM. It is a successor of a tool called Alive[3], which was a verifier for peephole optimizations[4] in LLVM. Alive was a very novel project in its time and it proved to be very useful for LLVM developers. Nonetheless, it had its limitations, particularly performance (it was written in Python), but also it was restricted to only one sort of transformation within LLVM called InstCombine which combines instructions to form fewer,

[3] `https://github.com/nunoplopes/alive`

[4] A *peephole optimization* transforms a small set of instructions to an equivalent set which has better performance

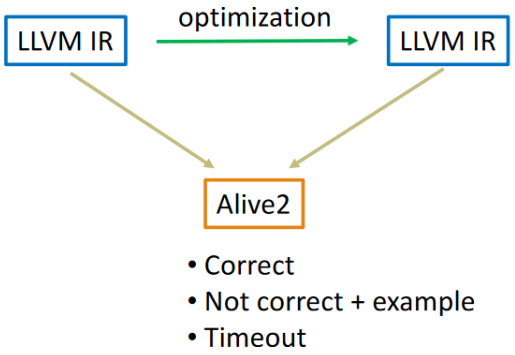

Figure 2.7 High-level illustration of Alive2 [18]

simple instructions but does not modify the control-flow graph. Moreover, Alive's own DSL had to be used to verify the optimizations.

Alive2 is a full re-implementation of Alive in C++, but not only that. Alive2 is a much more sophisticated framework than Alive, it includes a plugin for LLVM's `opt` and `clang` as well as a standalone tool called `alive-tv`. Alive2 checks pairs of functions in LLVM IR for *refinement*:

Definition 12 (Refinement [6])**.** *We say that a target function is a* refinement *of a source function if the target displays a subset of the behaviors of the source for every possible input.*

In the absence of undefined behavior, we could simply check for equivalence. Refinement allows a transformation to remove non-determinism, but not to add it. Alive2 cannot simply check for equivalence because LLVM's optimizations often take advantage of undefined behavior, so any verification tool targeting LLVM (or any modern compiler of a language with undefined behavior) must support it.

2.2.1 Encoding LLVM IR semantics

Alive2 encodes LLVM IR semantics in *SMT expressions* which it then provides to a tool called an *SMT solver*. *Satisfiability Modulo Theories* (SMT) is a field in computer science that involves checking the satisfiability of a logical formula over one or more theories containing more complex structures such as real numbers, integers, and various data structures. It is an extension of the Boolean satisfiability problem (SAT) that only deals with the satisfiability of formulas in propositional logic. An SMT solver is a tool that aims to solve the SMT problem for a formula over a given theory [33]. It is a well-known fact that SAT is NP-complete [34],

thus SMT problems are typically NP-hard (and sometimes even undecidable) but current SMT solvers can solve practical problems very efficiently. One such solver is the *Z3 Theorem Prover* [35], a state-of-the-art SMT solver developed by Microsoft Research, and it is also the one used by Alive2.

Internally, Alive2 uses its own IR (called *Alive IR*) to stay independent from LLVM. Syntactically, it is nearly identical to LLVM IR with only minor differences. Alive2 encodes the *state* of a program into an SMT formula. Program state consists of a register file, memory, and a flag stating whether the program has executed undefined behavior. We will not go into much detail regarding encoding into SMT as it is not relevant to our work, for details we refer the reader to the Alive2 paper [6].

Z3 Theorem Prover does not have a notion of programming constructs such as loops because its input language is in first-order logic. Thus, Alive2 needs to eliminate cycles in our CFG. There are many methods that deal with cycles in the CFG, for example invariants. Alive2 uses a technique called *loop unrolling*, which is a program transformation method that involves replacing the loop with several copies of its body[5]. Thus, Alive2 will find any failure of refinement that is manifested within the specified number of iterations, but miss those that are triggered beyond that.

Alive2 unrolls loops "inside-out" by traversing each loop tree in postorder. Traversing the loop trees in postorder means that the number of unrolls is linear in the number of loops and unroll factor instead of being exponential if done in the reverse order [6]. We will describe the unrolling algorithm in detail in the following chapter.

2.2.2 Showcase

We will now demonstrate how Alive2 works, specifically the tool `alive-tv` (`tv` is short for translation validation) which takes two LLVM IR files as arguments (source and target) and checks if target is a refinement of the source (i.e. if the transformation is correct). The two programs in Figure 2.8 are obviously equivalent since for the given parameter a, `src.ll` returns $a + 1$ and `tgt.ll` returns $1 + a$, and Alive2 correctly recognizes that the transformation is correct.

If we have a source-target pair where the target is not a refinement of the source, Alive2 provides a counterexample as seen in Figure 2.9. Alive2 reports that the transformation does not verify and moreover, it provides us with a counterexample for the variable `%a`. Specifically, it correctly says that setting `%a = 0` will produce the value of -1 in the source program and 1 in the target.

[5]It is also used in compiler optimization for different purposes.

src.ll

```
define i32 @test(i32 %a) {
    %b = add i32 %a, 1
    ret i32 %b
}
```

tgt.ll

```
define i32 @test(i32 %a) {
    %b = add i32 1, %a
    ret i32 %b
}
```

```
$ alive-tv src.ll tgt.ll
----------------------------------------
define i32 @test(i32 %a) {
#0:
  %b = add i32 %a, 1
  ret i32 %b
}
=>
define i32 @test(i32 %a) {
#0:
  %b = add i32 1, %a
  ret i32 %b
}
Transformation seems to be correct!
```

Figure 2.8 Alive2 output for equivalent source and target

src.ll

```
define i32 @test(i32 %a) {
    %b = sub i32 %a, 1
    ret i32 %b
}
```

tgt.ll

```
define i32 @test(i32 %a) {
    %b = sub i32 1, %a
    ret i32 %b
}
```

```
$ alive-tv src.ll tgt.ll
----------------------------------------
define i32 @src(i32 %a) {
#0:
  %b = sub i32 %a, 1
  ret i32 %b
}
=>
define i32 @tgt(i32 %a) {
#0:
  %b = sub i32 1, %a
  ret i32 %b
}
Transformation doesn't verify!

ERROR: Value mismatch

Example:
i32 %a = #x00000000 (0)

Source:
i32 %b = #xffffffff (4294967295, -1)

Target:
i32 %b = #x00000001 (1)
Source value: #xffffffff (4294967295, -1)
Target value: #x00000001 (1)
```

Figure 2.9 Alive2 output for non-equivalent source and target

Chapter 3

Analysis of the problem

In this chapter, we aim to provide a concise but thorough description of Alive2's loop handling algorithms. Analyzing the presented loop algorithms was an important part of our work in order to expose the problem that caused Alive2 to produce false alarms. The resulting pseudocode is our work and it was synthesized from the original source code in Alive2, specifically from `ir/function.cpp`[1] which includes most of Alive2 loop handling code.

Let us first briefly recall the setting of our problem. We are working with Alive2, a translation validation framework that accepts a source-target pair of LLVM IR source codes and attempts to prove or disprove that the target is a *refinement* (see Definition 12) of the source. In order to achieve this, Alive2 translates the given LLVM IR code to an SMT formula which is then passed to the Z3 SMT solver. In loopless code, this translation can be done immediately, but in code that involves loops, the situation is much more complex. Z3 SMT solver accepts formulas in first-order logic which has no notion of loops, so they must somehow be removed from the code. There are several techniques on how to deal with this issue as we describe in the previous chapter. Alive2 uses technique called loop unrolling which involves removing the loop's backedge and duplicating the loop a given number of times (known as *unroll factor*). We call Alive2's output a *false alarm* if the target program is a refinement of the source program but the output says it is not.

3.1 Loop algorithms in Alive2

Recall that a loop in a control-flow graph $G = (N, E)$ must contain a backedge. As we have already described, Alive2 deals with loops by unrolling them. Before the unrolling itself is performed, we must be able to identify loops and their

[1] `https://github.com/AliveToolkit/alive2/blob/master/ir/function.cpp`

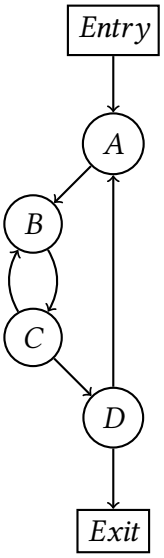

Figure 3.1 Running example

nesting relation so that we are able to unroll in the correct order. We describe this in Section 3.1.1. After the loops are identified, we can unroll them while also patching the instructions that were broken by unrolling (Section 3.1.2). Recall the motivating example from Chapter 1 shown in Figure 3.1.

We will us this as our running example throughout the chapter. We already know that the CFG contains the following two loops:

- $\ell_1 = (h_1, B_1, X_1)$, where $h_1 = A$, $B_1 = \{B, C, D\}$, $X_1 = \{exit\}$, and D is the latch,
- $\ell_2 = (h_2, B_2, X_2)$, where $h_2 = B$, $B_2 = \{C\}$, $X_2 = \{D\}$, and C is the latch,

and that ℓ_2 is nested in ℓ_1.

3.1.1 Loop identification

Recall that $uv \in E$ is a backedge with respect to a given DFS tree if v is an ancestor of u in the DFS tree. Moreover, uv is a reducible backedge if v dominates u. We can observe that any reducible backedge uv found by DFS in a reducible CFG determines a loop. An analogical statement can be made for irreducible loops but it will get a little more complicated as we shall see in the algorithm. But we can see that in order to identify loops, we need to detect backedges. To be able to unroll nested loops correctly, we also need to classify loops according to their nesting relation which means building a loop forest (Definition 10). To identify loops, we also need to decide three things for every node v in the CFG [36]:

1. Is v a loop header?
2. Is v inside a loop body? If yes, what is the header of the innermost loop v is contained in?
3. If v is inside a loop ℓ, is there a $u \notin \ell$ such that $uv \in E$? That is, is ℓ irreducible?

In order to build a loop body, we will need a structure to efficiently decide whether or not two basic blocks belong to the same loop, and if yes, then merge them. With the *Union-Find* data structure such operations can be very fast, shown to be almost linear by Tarjan [37]. The UNION and FIND operations are applied on disjoint sets from a given universe $\mathcal{U}$. We will not go into details about how to implement the operations, we will only describe how we shall use them. Let us index the elements of $\mathcal{U}$ as $m_1, \ldots, m_n$ and initially set $s_i := \{m_i\}$ for all $i \in [n]$.

- UNION(i, j): assigns $s_j := s_i \cup s_j$ and $s_i := \emptyset$
- FIND(k): returns i such that $m_k \in s_i$

Following the identification, we need to decide the nesting relations between the identified loops which involves building a loop forest.

Havlak's algorithm

Alive2 identifies loops and build the loop forest using *Havlak's algorithm* [10] (sometimes referred to as the *Havlak-Tarjan algorithm*). Havlak's algorithm is based on Tarjan's method for testing reducibility [38], but additionally, it can detect irreducible loops while Tarjan's algorithm simply terminates upon encountering an irreducible loop. Havlak's algorithm traverses the CFG twice: first using DFS from *Entry* to detect backedges and forward edges, then in the reverse order based on Union-Find to build the loop body for each header and construct the loop tree.

Let us follow the pseudocode in Algorithm 3. Backedges and forward edges are maintained as sets of predecessors that we call *backPreds* and *nonBackPreds* respectively. They are defined as such for a node $w \in N$: $backPreds[w] := \{v \mid vw \text{ is a backedge}\}$, $nonBackPreds[w] := \{v \mid vw \in E \wedge v \notin backPreds[w]\}$. Both of those structures can be populated using the information we got from DFS as we already know from Section 1.1.1 that ancestry in a DFS tree can be tested very quickly by saving every node's last descendant. And anytime we encounter an edge uv such that v is an ancestor of u, we know that uv is a backedge (lines 5–11).

We also maintain the Union-Find data structure where the representative of each node is either (1) the node itself if it is not inside any loop body, or (2) the

header of the inner-most loop that the node is contained in. This is built during the second traversal of the CFG where we go through the nodes in reverse preorder (lines 14–40). By using reverse preorder, we ensure each inner loop will be processed before the loops it is nested in. Let's say we are processing a node w. If $backPreds[w] = \emptyset$, that is w is not a destination of any backedge and consequently not a loop header, we simply move on to the next node. Otherwise, we start constructing the loop body P for header w by first adding the latches, that is by going through each $v \in backPreds[w]$ and adding $FIND(v)$ (the representative of node v) to P. Then we complete the construction of P in worklist fashion by adding nodes that point to some node in P but are not yet in P themselves. If during this we encounter a node v that is not a descendant of the header w, it means it is another entry to the currently processed loop and thus the loop is irreducible (and in such a case, we don't add v to the body P). Finally, we merge every node $v \in P$ with the header w for the Union-Find structure while also setting $header[v] := w$. Constructing the loop forest is simple, for each node v we only have to set $header[v]$ as the parent of v. Finally, we construct the loop forest by iterating over the nodes in preorder and for every node v, we decide:

- If v is not inside any loop and it is a loop header, we create a new tree in the forest with v as its root

- If v is inside a loop, we attach it under its header in the loop tree

In our running example from Figure 3.1, all nodes have their *backPreds* set empty, except for A and B which contain D and C respectively. For every node v, $nonBackPreds[v]$ contains its direct ancestor in our example, for example $nonBackPreds[C] = \{B\}$. If our CFG had a node v with multiple entry points, the set $nonBackPreds[v]$ would contain those respective entry points.

Then we go through the nodes in reverse preorder, which in our case is D, C, B, A. Let us for example consider the iteration with node B. In this case, $P = \{C\}$, so $W = \{C\}$ and we will only go through one iteration of the while-loop beginning at line 24. This is because C has only one immediate ancestor B and neither condition on lines 28 and 31 will be satisfied. In the union-find step, B becomes the representative of itself and C.

Constructing the loop tree (line 43) will be straightforward; *Entry* (and *Exit*) will not trigger any of the conditions inside the loop, A is the only basic block that will satisfy the first one and thus we set it as the root of our loop tree, B is a header of a loop but is is also inside the loop headed by A and hence we add it as a child of A into the loop tree, C is inside B's loop so likewise, we add it as a child of B, and finally D is inside the loop headed by A and so we add it as a child of A. We can see the resulting loop tree in Figure 3.2.

Algorithm 3 This algorithm identifies loop headers and builds a loop-nesting forest while also recording the types of the loops

```
1:  function LoopAnalysis(G: control-flow graph)
2:      v_1, ..., v_n ← preorder numbering of the basic blocks starting from Entry
3:      backPreds ← an empty map N → 𝒫(N)
4:      nonBackPreds ← an empty map N → 𝒫(N)
5:      for every edge uv ∈ E do
6:          if v is an ancestor of u then
7:              Add u to backPreds[v]
8:          else
9:              Add u to nonBackPreds[v]
10:         end if
11:     end for
12:     type(∗) = nonheader
13:     header(∗) = None
14:     for v ∈ N in reverse preorder do
15:         P ← {FIND(u) | u ∈ backPreds[v], u ≠ v} ..... P is the loop body
16:         if v is a self-loop then
17:             type(v) ← self
18:         end if
19:         if P ≠ ∅ then ..... If the body is empty, then v not a header
20:             ..... If it is not, we first assume v is the header of a reducible loop
21:             type(v) ← reducible
22:         end if
23:         W ← P ..... W is the worklist
24:         while W ≠ ∅ do
25:             x ← pick an arbitrary node from W and remove it
26:             for every y ∈ nonBackPreds[x] do
27:                 z ← FIND(y)
28:                 if v is not an ancestor of z then
29:                     type(v) ← irreducible
30:                     Add z to nonBackPreds[v]
31:                 else if z ∉ P and z ≠ v then
32:                     Add z to P and W
33:                 end if
34:             end for
35:         end while
```

```
36:         for every basic block b ∈ B do
37:             header(b) ← v
38:             UNION(v, b) ..... b will be the representative
39:         end for
40:     end for
41:     ..... Now we construct the loop forest
42:     forest ← empty forest
43:     for every v ∈ V in preorder do
44:         if header(v) = None and type(v) ≠ nonheader then
45:             Add new root v to forest
46:         else if header(v) ≠ None or type(v) ≠ nonheader then
47:             Add v as a child of header(v) to forest
48:         end if
49:     end for
50:     return forest
51: end function
```

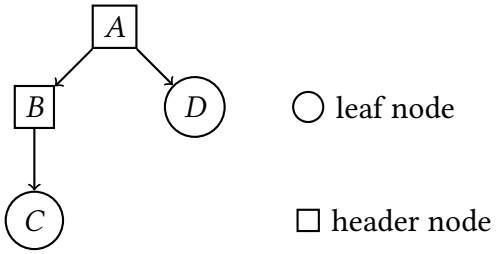

Figure 3.2 Loop tree of the running example

As we can see, Havlak's algorithm has successfully identified all the loops in our CFG. With irreducible CFGs, the situation can get more complicated and which loops are identified depends on the choice of the depth-first spanning tree [10]. We shall not elaborate on this further here as it is not relevant to our problem but we will briefly discuss it in Chapter 6.

3.1.2 Loop unrolling

In the previous chapter, we explained why unrolling loops is necessary in Alive2. Let us now focus on the unrolling algorithm itself. Alive2 unrolls a loop nest "inside-out", which means the inner-most loop is always unrolled first. We shall explain the algorithm with the help of pseudocode in Algorithm 4.

We first need to obtain the loop forest from the LoopAnalysis algorithm and

then traverse each loop tree in postorder. Let's say we are now unrolling a loop $\ell = (h, B, X)$. We first need to determine which basic blocks need to be duplicated. For this we maintain a map *loopNodes* for each header such that eventually $loopNodes[h] = \{h\} \cup B$ once constructed. Because we traverse the loop tree in postorder, all the inner loops of ℓ already have *loopNodes* constructed and thus we can build $loopNodes[h]$ inductively: $loopNodes[h] = \bigcup_{\ell' \text{ nested in } \ell} loopNodes[h'] \cup \{bb \in N \mid bb \text{ is a leaf node with } h \text{ as the parent}\}$. We construct *loopNodes* by iterating over the loop tree in topological order and either adding the node itself if it is a leaf node, or the whole loop if the node is a loop header (lines 10–17).

We then clone the blocks in the order *loopNodes* was constructed (lines 19–33). While cloning a basic block, we also duplicate its instructions by essentially giving them new SSA names. This may seem simple enough, but we also need to update the instruction's operands since they might also have been previously duplicated. For this, we maintain a map *vmap* which has the original instructions as keys and as values an array of pairs (bb, op), where *bb* is always the cloned basic block and *op* is the duplicated operand belonging to the basic block *bb*. Thus, when duplicating an instruction, we simply take the latest duplicate of its operands to create the new duplicated instruction (line 28). Recall that since we duplicate in topological order according to the loop tree, "latest duplicate" means the greatest with respect to topological order of the loop tree. Besides operands, there are two more things that need to be patched after unrolling [6]:

1. Targets of jump instructions: Jump targets are naturally patched by replacing each target with its next duplicate. If there is no such duplicate, this means we are dealing with a backedge in the last unroll and these jumps are redirected to a special basic block called *sink* (see our unrolled example 3.5)

2. Introduce/patch ϕ instructions: Some instructions inside a loop may have their result used outside the loop. There are three cases Alive2 needs to deal with in regard to ϕ instructions:

   (a) Existing ϕ instructions → add more predecessors

   (b) The loop has a single exit to a basic block that dominates the user's basic block → introduce a new ϕ node. A *user* is an instruction using the value.

   (c) Otherwise → introduce a new stack variable to avoid maintaining SSA altogether.

The task of fixing SSA after loop transformations is commonly called *SSA reconstruction* [28].

Let us go through the algorithm with the help of our running example and we will unroll with the factor of $k = 1$. There is only one loop tree in *forest*, we

Algorithm 4 This algorithm unrolls all loops in the program with the unrolling factor of k

```
1: function UNROLL(G = (N, E): control-flow graph, k ≥ 0: unroll factor)
2:     if k = 0 then return
3:     end if
4:     forest ← LOOPANALYSIS(G)
5:     loopNodes ← empty map N → 𝒫(N)
6:     for every loop tree T in forest do
7:         for every header h ∈ T in postorder do
8:             loopNodes[h] ← {h}  ..... The basic blocks we will duplicate
9:             descendants ← descendants of h in forest
10:            for every basic block bb ∈ TOPSORT(descendants) do
11:                if bb is a header of some loop then
12:                    .. We need to include the nested loop's BBs for duplication
13:                    Append loopNodes[bb] to loopNodes[h]
14:                else
15:                    Append bb to loopNodes[h]
16:                end if
17:            end for
18:            for i = 1, ..., k do
19:                for bb in loopNodes[h] do
20:                    ..... Now we clone the basic block bb
21:                    clonedBB ← a new empty basic block
22:                    phis ← ∅
23:                    for every instruction i in bb do
24:                        if i is a φ instruction then
25:                            Add i to phis
26:                        end if
27:                        d ← a copy of i
28:                        Replace all operands of d with their latest duplicates in
   vmap
29:                        Add d to vmap[i] if i is a non-void instruction
30:                        Add d to clonedBB
31:                        Patch φ predecessors
32:                    end for
33:                end for
34:            end for
35:            Patch jump targets, users and φ instructions
36:        end for
37:    end for
38: end function
```

can see it with postorder numbering in Figure 3.3. There are two headers in the loop tree, so we will go through two iterations.

In the first iteration, the header h will be B since it is first in postorder. The variable *descendants* will be a singleton set $\{C\}$ as C is the only descendant of B in the loop tree. Thus, construction of $loopNodes[B]$ will be simple since there is only one possible topological order of one element and C is not a header of any loop so $loopNodes[B]$ will contain B itself and C, in this order.

For clarity, we will only go through the basic block duplication during this iteration. To illustrate basic block duplication, let us say that C contains the instruction `%i.inc = add i32 %i, 1`, that is we add 1 to the instruction `%i` and store it in `%i.inc`. Let us assume that the instruction `%i` comes from the basic block B which has already been cloned and thus contains the duplicate `%i#1` of `%i`. When duplicating `%i.inc = add i32 %i, 1`, we will query the last element of `vmap[%i]` which will be `%i#1` and we will use it as the pertinent operand. The resulting duplicated instruction will be `%i.inc#1 = add i32 %i#1, 1`.

In the second iteration, we are duplicating the basic blocks of the loop with header A. The variable *descendants* is now $\{B, C, D\}$, we can see the subtree induced by those nodes in Figure 3.4. To construct $loopNodes[B]$, we need to topologically sort this subgraph. There are actually three possible topological orders of this graph: $B \prec C \prec D$, $B \prec D \prec C$, and $D \prec B \prec C$. Which of those orders is chosen is dependent on the internal implementation of topological sorting. It may not appear significant, but this uncertainty might be problematic during basic block duplication since patching of operands depends on topological order. With incorrect ordering, wrong values can be observed which can cause a different output in source and target programs. This is quite subtle since it does not affect the structure of the unrolled CFG (which we can see in Figure 3.5) but it can cause problems and even false alarms as we shall learn in Chapter 4, though it may not be obvious at the moment.

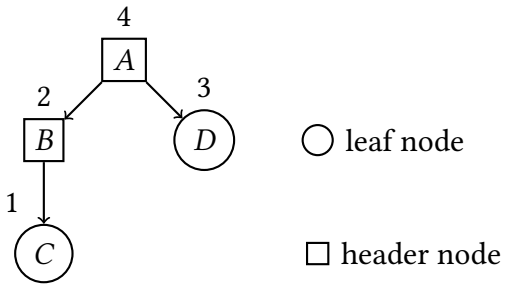

Figure 3.3 Postorder numbering of the loop tree

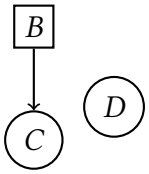

Figure 3.4 Descendants of A in the loop tree

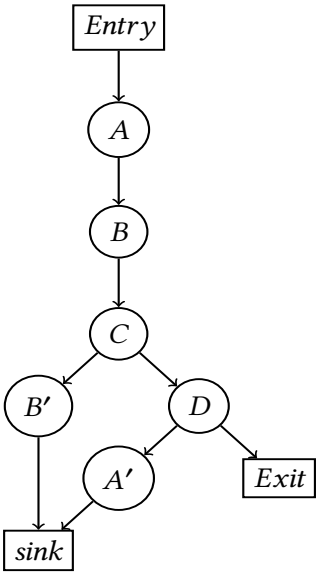

Figure 3.5 Running example unrolled with a factor of 1

3.2 Conclusions from the analysis

We identified several problems in Alive2's loop algorithms some of which we describe in Chapter 6. Nevertheless, the one most likely to cause false alarms in nested loops is constructing the order in which basic blocks should be cloned (line 10). Let us consider the situation in our running example. Recall its looptree in Figure 3.2. Now, if we want to unroll the loop headed by the node A, we need to sort its descendants topologically. The descendants form the disconnected graph in Figure 3.4

And thus, there are three possible topological orders: $B \prec C \prec D$, $B \prec D \prec C$, and $D \prec B \prec C$. Nevertheless, in the original graph, there exists a path from B to D as well as from C to D, so only the first one should be correct. It may not appear that this poses a problem at first but recall that during the unroll, we patch instruction operands by replacing them with its latest duplicate and so an operand may be patched incorrectly.

This is definitely a bug in the algorithm but it does not mean that it can

manifest in practice nor that it could cause false alarms. In the next chapter, we will demonstrate that it can actually happen with a real-world example from LLVM test suite and produce a false alarm.

Chapter 4

Problem in the unrolling algorithm

In the previous chapter, we analyzed Alive2's loop algorithms and identified a problem in the unrolling algorithm that may cause false alarms. In this chapter, we look at the problem in more detail, and in particular, we demonstrate that it can manifest in a real-world LLVM test. We also discuss what the correct behavior should be in that particular case.

4.1 Root cause of false alarms with nested loops

Recall that when Alive2 wants to unroll a loop, it needs to obtain the correct basic blocks to duplicate. If we are unrolling a loop with header h, Alive2 topologically sorts all descendants of h in the loop tree and duplicates them in this order. Since the loop tree does not include all edges that were present in the original graph, this might produce several possible orders that may not be valid in the original graph. Recall the situation in our running example shown in Figure 4.1. We claim that this might produce false alarms because if a basic block is cloned too early, its instructions could observe the wrong values and so the source and target programs can produce different outputs. To demonstrate this explicitly, we examine an LLVM example with this behavior in the following section.

4.1.1 Failing example in LLVM test suite

We will now demonstrate the issue on a real failing example from LLVM test suite. This example failed with the so-called *LCSSA* pass, so let us first explain what LCSSA is. For some loop optimizations, it is desirable to maintain a more restrictive type of SSA which is called the *loop-closed SSA (LCSSA) form*. This type of SSA simplifies updating the SSA form after loop optimizations. It was invented by Zdeněk Dvořák while working on GCC's loop optimizer [39] and

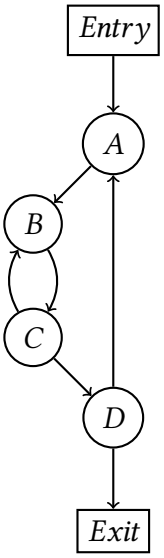

Figure 4.1 Running example

```
c = ...;
for (...) {
  if (c)
    X1 = ...
  else
    X2 = ...
  X3 = phi(X1, X2);  // X3 defined
}
... = X3 + 4;  // X3 used outside the loop
```

Figure 4.2 A program in SSA but not in LCSSA

it has apparently been present in LLVM's own loop optimizer since the very beginning [40].

Definition 13 (Loop-closed SSA form)**.** *A program is in* LCSSA form *if it is in SSA form and all values defined in a loop are only used within that loop.*

In practice, this means adding an extra single-parameter ϕ function in the loop's exit block. We can see an example[1] of a program in SSA but not in LCSSA (Figure 4.2) and an equivalent program in LCSSA (Figure 4.3).

We can see the code is still valid, the extra ϕ node is redundant. In LLVM, this form is ensured by the `lcssa` pass and is added automatically by the LoopPassManager, all loop passes are required to preserve LCSSA. After the loop

[1] `https://llvm.org/docs/LoopTerminology.html#loop-terminology-lcssa`

```
c = ...;
for (...) {
  if (c)
    X1 = ...
  else
    X2 = ...
  X3 = phi(X1, X2);
}
X4 = phi(X3); // redundant phi for LCSSA

... = X4 + 4;
```

Figure 4.3 An equivalent program in LCSSA

optimizations are done, these extra phi nodes will be deleted by the InstCombine pass[2].

By running Alive2 with unroll factor of two over the LLVM test suite, a failing example which demonstrates this issue was found[3] (see Figure 4.4). The LLVM IR code in Figure 4.4 corresponds to the CFG in Figure 4.6. As we can see from the CFG, it is a simple nested loop with `header` being the header of the inner loop, and `outer_header` the header of the outer loop, more formally we have:

- ℓ': $h' :=$ `header`, $B' :=$ {`backedge`}, $X' :=$ {`outer_backedge`}
- ℓ: $h :=$ `outer_header`, $B := \ell' \cup$ {`preheader`, `outer_backedge`}, $X :=$ {`exit`}

Moreover, ℓ' is nested inside ℓ. The pertinent loop tree is shown in Figure 4.7.

When topologically sorting this tree, `outer_header` must always be first, but there is a tiebreak between `preheader` and `outer_backedge`. Alive2 chooses `outer_backedge` to come first and thus produces the following order:

1. `outer_header`
2. `outer_backedge`
3. `preheader`
4. `header`
5. `backedge`

[2] https://llvm.org/docs/Passes.html#instcombine-combine-redundant-instructions

[3] https://github.com/llvm/llvm-project/blob/main/llvm/test/Transforms/LoopSimplifyCFG/constant-fold-branch.ll#L819

```
define i32 @partial_sub_loop_test_branch_loop(i32 %end) {
entry:
  br label %outer_header

outer_header:
  %j = phi i32 [0, %entry], [%j.inc, %outer_backedge]
  br label %preheader

preheader:
  br label %header

header:
  %i = phi i32 [0, %preheader], [%i.inc, %backedge]
  br label %backedge

backedge:
  %i.inc = add i32 %i, 1
  %cmp = icmp slt i32 %i.inc, %end
  br i1 %cmp, label %header, label %outer_backedge

outer_backedge:
  %j.inc = add i32 %j, 1
  %cmp.j = icmp slt i32 %j.inc, %end
  br i1 %cmp.j, label %outer_header, label %exit

exit:
  ret i32 %i.inc
}
```

Figure 4.4 A reduced example from LLVM test suite

```
define i32 @partial_sub_loop_test_branch_loop(i32 %end) {
entry:
  br label %outer_header

outer_header:
  %j = phi i32 [ 0, %entry ], [ %j.inc, %outer_backedge ]
  br label %preheader

preheader:
  br label %header

header:
  %i = phi i32 [ 0, %preheader ], [ %i.inc, %backedge ]
  br label %backedge

backedge:
  %i.inc = add i32 %i, 1
  %cmp = icmp slt i32 %i.inc, %end
  br i1 %cmp, label %header, label %outer_backedge

outer_backedge:
  %i.inc.lcssa = phi i32 [ %i.inc, %backedge ]
  %j.inc = add i32 %j, 1
  %cmp.j = icmp slt i32 %j.inc, %end
  br i1 %cmp.j, label %outer_header, label %exit

exit:
  %i.inc.lcssa.lcssa = phi i32 [ %i.inc.lcssa, %outer_backedge ]
  ret i32 %i.inc.lcssa.lcssa
}
```

Figure 4.5 The example from Figure 4.4 after LCSSA transformation

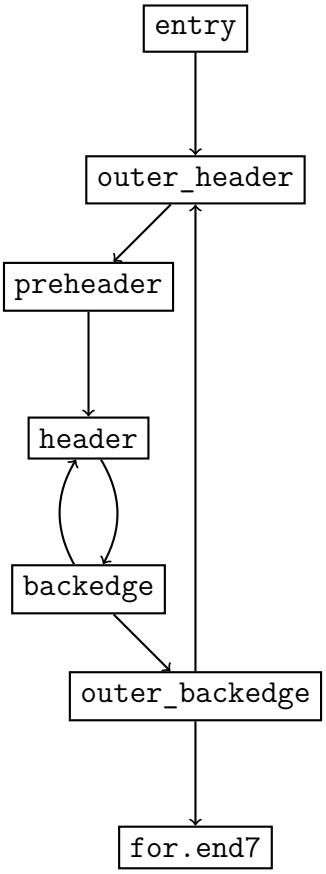

Figure 4.6 A failing example CFG

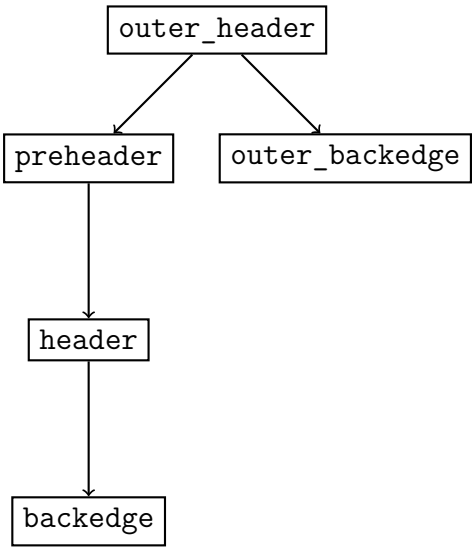

Figure 4.7 Loop tree of failing example

Thus, the ϕ nodes of `outer_backedge` may contain wrong predecessors because it is cloned too early.

The failing transformation is in the `lcssa` pass which simply converts the program into LCSSA form. If we try to verify this with `alive-tv` with unroll of two, that is running `alive-tv bug.ll -src-unroll=2 -tgt-unroll=2 -p=lcssa`, Alive2 says the transformation does not verify. We include the whole output in the following chapter but let us now demonstrate that there are indeed wrong ϕ predecessors present in the output:

Listing 4.1 Wrong predecessors in Alive2 output

```
%outer_backedge#2:
  %i.inc.lcssa#2 = phi i32 [ %i.inc, %backedge ], [ %i.inc#1#2,
      %backedge#1#2 ]
```

The unrolled CFG in Figure 4.8 also clearly shows that the ϕ predecessors are wrong. %outer_backedge#2 contains neither %backedge nor %backedge#1#2 as direct predecessors and thus the ϕ may be observing the wrong values. The right predecessors would be %backedge#2 and %backedge#1#2#2 and thus the correct instruction should look like the following:

```
 %i.inc.lcssa#2 = phi i32 [ %i.inc#2, %backedge#2 ], [ %i.inc
     #1#2#2, %backedge#1#2#2 ]
```

Alive2 also provides us with a counterexample of the %end parameter with a value of 2. It plugs the value of 2 to the unrolled program and produces the following walkthrough of the CFG for the source function:

```
Source:
ptr %i.inc#ptr#2 = pointer(local, block_id=0, offset=0)
    >> Jump to %outer_header
i32 %j = #x00000000 (0)
    >> Jump to %preheader
    >> Jump to %header
i32 %i = #x00000000 (0)
    >> Jump to %backedge
i32 %i.inc = #x00000001 (1)
i1 %cmp = #x1 (1)
    >> Jump to %header#1#2
i32 %i#1#2 = #x00000001 (1)
    >> Jump to %backedge#1#2
i32 %i.inc#1#2 = #x00000002 (2)
i1 %cmp#1#2 = #x0 (0)
    >> Jump to %outer_backedge
i32 %i.inc#phi#0 = #x00000002 (2)
i32 %j.inc = #x00000001 (1)
i1 %cmp.j = #x1 (1)
    >> Jump to %outer_header#2
```

```
i32 %j#2 = #x00000001 (1)
    >> Jump to %preheader#2
    >> Jump to %header#2
i32 %i#2 = #x00000000 (0)
    >> Jump to %backedge#2
i32 %i.inc#2 = #x00000001 (1)
i1 %cmp#2 = #x1 (1)
    >> Jump to %header#1#2#2
i32 %i#1#2#2 = #x00000001 (1)
    >> Jump to %backedge#1#2#2
i32 %i.inc#1#2#2 = #x00000002 (2)
i1 %cmp#1#2#2 = #x0 (0)
    >> Jump to %outer_backedge#2
i32 %i.inc#phi#0#2 = poison
i32 %j.inc#2 = #x00000002 (2)
i1 %cmp.j#2 = #x0 (0)
    >> Jump to %exit
i32 %i.inc#phi#1 = #x00000001 (1)
i32 %i.inc#ptr#2#load = #x00000001 (1)
```

And this is the same walkthrough for the target function:

```
Target:
    >> Jump to %outer_header
i32 %j = #x00000000 (0)
    >> Jump to %preheader
    >> Jump to %header
i32 %i = #x00000000 (0)
    >> Jump to %backedge
i32 %i.inc = #x00000001 (1)
i1 %cmp = #x1 (1)
    >> Jump to %header#1#2
i32 %i#1#2 = #x00000001 (1)
    >> Jump to %backedge#1#2
i32 %i.inc#1#2 = #x00000002 (2)
i1 %cmp#1#2 = #x0 (0)
    >> Jump to %outer_backedge
i32 %i.inc.lcssa = #x00000002 (2)
i32 %j.inc = #x00000001 (1)
i1 %cmp.j = #x1 (1)
    >> Jump to %outer_header#2
i32 %j#2 = #x00000001 (1)
    >> Jump to %preheader#2
    >> Jump to %header#2
i32 %i#2 = #x00000000 (0)
    >> Jump to %backedge#2
i32 %i.inc#2 = #x00000001 (1)
i1 %cmp#2 = #x1 (1)
```

```
    >> Jump to %header#1#2#2
i32 %i#1#2#2 = #x00000001 (1)
    >> Jump to %backedge#1#2#2
i32 %i.inc#1#2#2 = #x00000002 (2)
i1 %cmp#1#2#2 = #x0 (0)
    >> Jump to %outer_backedge#2
i32 %i.inc.lcssa#2 = poison
i32 %j.inc#2 = #x00000002 (2)
i1 %cmp.j#2 = #x0 (0)
    >> Jump to %exit
i32 %i.inc.lcssa.lcssa = poison
Source value: #x00000002 (2)
Target value: poison
```

We can see that because of the wrong predecessors, Alive2 jumps to the wrong basic blocks and the source and target programs produce a different value which leads to a false alarm. Specifically, the output value of the source program is 2 while the value of the target program is *poison*. Poison is a type of value in LLVM IR representing the result of undefined behavior, the other one being *undef* [25].

4.2 Summary

In this chapter, we have demonstrated on a real-world LLVM example that the bug we identified in Chapter 3 produces false alarms in Alive2. We have explained why the false alarm occurs and how to reproduce the issue. In the next chapter, we will look more closely on how to prevent this issue and the solution we have submitted and integrated to Alive2.

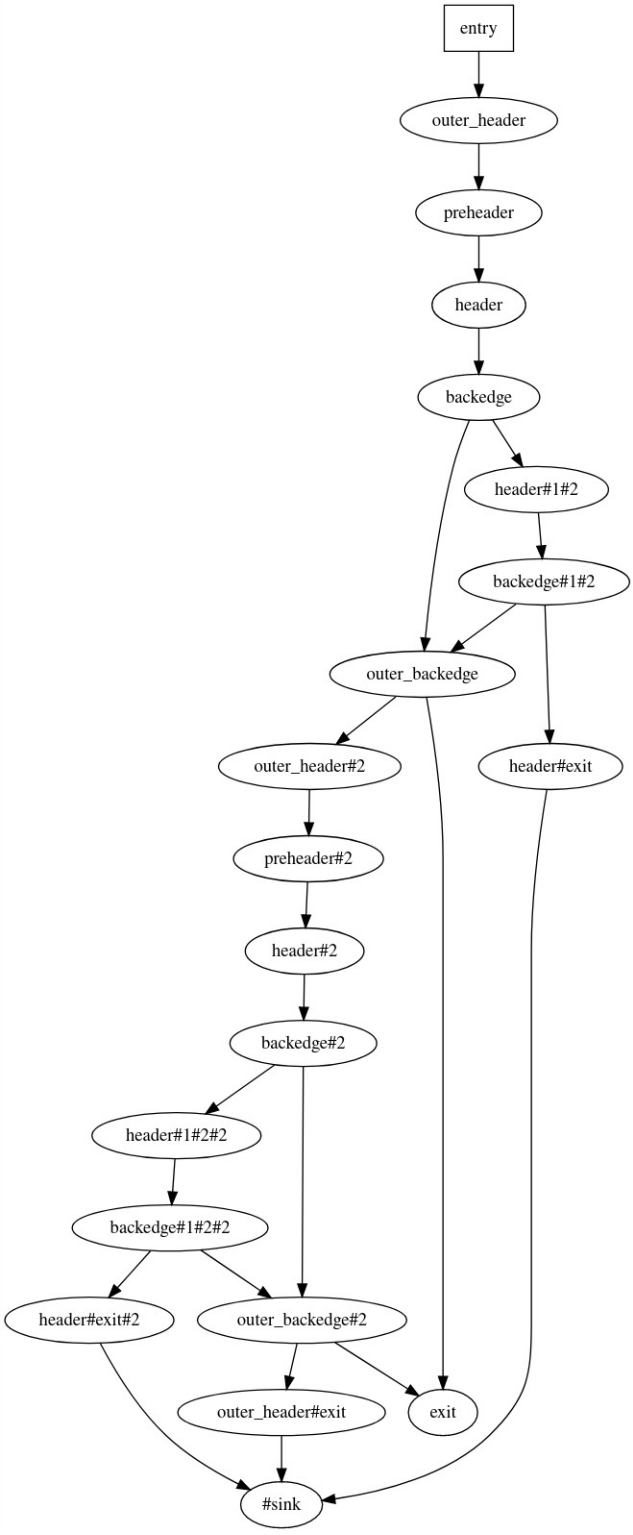

Figure 4.8 Failing example from Figure 4.6 unrolled with a factor of two.

Chapter 5

Implementation and evaluation

In this chapter, we describe our approach to fixing the problem. In the final section, we also present concrete results of our contributions to Alive2.

5.1 Our solution and implementation in Alive2

The essence of the problem is in the order of cloning basic blocks. Specifically, it is the failure to take into account some edges that were present in the original graph but are not present in the loop tree. So to fix this problem, we must take these edges into account which is actually the approach we ended up implementing and we describe it in Section 5.1.1. The reason we decided on this approach is that it added virtually no overhead in terms of performance to Alive2 unrolling algorithm and fixed the problems that needed to be fixed.

5.1.1 Take exit blocks into account

As we have explained in Chapter 4, the current Alive2's unrolling algorithm does not take exit blocks into account when topologically sorting the descendants of a loop header in the loop tree. We have fixed this by adding exit blocks to each loop header before it is unrolled. Here is the relevant part of the unroll function[1]:

```
1 for (auto &dst : bb->targets()) {
2   if (!bbmap.count(&dst)) {
3     exit_edges.emplace(bb, const_cast<BasicBlock*>(&dst));
4     header->addExitBlock(const_cast<BasicBlock*>(&dst));
5   }
6 }
```

[1] https://github.com/AliveToolkit/alive2/blob/8bf86254e2c799526e0f1dc99eefb9c7da646c1d/ir/function.cpp#L566

Here, Alive2 already had code to collect exit edges, so we could use it to get the exit blocks on line 4, thus adding very little overhead. Afterward, we had to communicate this information to top-sort:

```
1  static vector<BasicBlock*> top_sort(const vector<BasicBlock*> &
       bbs) {
2      edgesTy edges(bbs.size());
3      unordered_map<const BasicBlock*, unsigned> bb_map;
4
5      unsigned i = 0;
6      for (auto bb : bbs) {
7          bb_map.emplace(bb, i++);
8      }
9
10     i = 0;
11     for (auto bb : bbs) {
12         for (auto &dst : bb->targets()) {
13         auto dst_I = bb_map.find(&dst);
14         if (dst_I != bb_map.end())
15             edges[i].emplace(dst_I->second);
16         }
17
18         // If `bb` is a loop header, we need to go through its
               exit block
19         // in order to account for some transitive dependencies
               we may have
20         // missed due to compression of its inner loops.
21         // If there are no inner loops, this is redundant and if
               `bb` is not
22         // a loop header, the set of its exit blocks is empty.
23         for (auto &dst : bb->getExitBlocks()) {
24         auto dst_I = bb_map.find(dst);
25         if (dst_I != bb_map.end())
26             edges[i].emplace(dst_I->second);
27         }
28         ++i;
29     }
30
31     vector<BasicBlock*> sorted_bbs;
32     sorted_bbs.reserve(bbs.size());
33     for (auto v : util::top_sort(edges)) {
34         sorted_bbs.emplace_back(bbs[v]);
35     }
36
37     assert(sorted_bbs.size() == bbs.size());
38     return sorted_bbs;
39 }
```

Our added lines 23-27 build the edges that were previously missing and thus

enforcing the correct order of basic blocks. We decided for this solution because it was as minimal as possible while also fixing the underlying issue. Thus it also has no detrimental effect on performance or readability. Our solution works even if there are several basic blocks in an outer loop reachable from the exit block because of the way Alive2 pre-orders the basic blocks but such cases are extremely rare anyway.

Let us look at the output of `partial_sub_loop_test_branch_loop`, one of the tests in `Transforms/LoopSimplifyCFG/constant-fold-branch.ll` that we have already seen in the previous chapter. We include the whole output in the attachment, now let us just verify that the ϕ predecessors are indeed correct. Recall the instruction with wrong predecessors from before:

Listing 5.1 Wrong predecessors in Alive2 output

```
%outer_backedge#2:
    %i.inc.lcssa#2 = phi i32 [ %i.inc, %backedge ], [ %i.inc
        #1#2, %backedge#1#2 ]
```

With our fix, this instruction becomes:

Listing 5.2 Correct predecessors after our fix

```
    %i.inc.lcssa#2 = phi i32 [ %i.inc#2, %backedge#2 ], [ %i.inc
        #1#2#2, %backedge#1#2#2 ]
```

We can see that the previously wrong predecessors are now correct and thus the ϕ function observes the right values.

We submitted our solution as a pull request to Alive2 and it was almost immediately accepted with only minor comments[2]. Our solution is now thus a part of the Alive2 framework.

5.2 Evaluation

To demonstrate that our solution worked, we ran it over the relevant tests in LLVM test suite with unroll factor of two. Specifically, there were six false positives in `Transforms/LoopSimplifyCFG/constant-fold-branch.ll` which we used as our benchmark test set. Furthermore, we wanted to verify that the performance overhead was in fact negligible.

[2] `https://github.com/AliveToolkit/alive2/pull/908/`

5.2.1 Environment

The benchmarks were run on a PC running Linux Mint 20 (Ulyana) with Intel(R) Core(TM) i5-8265U CPU @ 1.60GHz and 8GB of RAM. We used LLVM 15.0.5 and Alive2 was compiled against this version of LLVM as well.

5.2.2 Results

To verify that our solution fixed false alarms present in `constant-fold-branch.ll`, we ran Alive2 translation validation over all the tests in the test file with the LCSSA pass and unroll factor of two. This can be accomplished with the script `opt-alive.sh` which is included in the build of Alive2. In our case, the invocation would look like this:

```
~/alive2/build/opt-alive-test.sh ~/llvm-project/llvm/test/
    Transforms/LoopSimplifyCFG/constant-fold-branch.ll --passes=
    lcssa -tv-src-unroll=2 -tv-tgt-unroll=2
```

We include the output of the run in the attachment. Note that the `-tv-exit-on-error` flag must be removed from the script if we do not want to abort the run after the first failure. We can see from Table 5.1 that the running time is virtually unaffected which we expected. To demonstrate this further, we ran several more passes in the LLVM test suite before[3] (see Table 5.2) and after[4] our commit in Alive2 (Table 5.3) with unroll factors of one, two and four. To accomplish this, we ran `llvm-lit` with Alive2's plugin as follows:

```
./llvm/build/bin/llvm-lit -s -Dopt=~/alive2/build/opt-alive.sh <
    LLVM test>
```

`opt-alive.sh` is a wrapper script for `opt` (LLVM's optimizing tool) and the unroll factor must be specified inside this script at the moment.

We ran the benchmarks a little over a hundred times each, discarding a few warm-up runs. We include the population standard deviation σ in the tables as well.

	Correct	Incorrect	Avg. running time	σ
Before patch	39	6	1.82s	0.052
After patch	45	0	1.81s	0.064

Table 5.1 Running translation validation on Transforms/LoopSimplifyCFG/constant-fold-branch.ll with source and target programs unrolled with a factor of two.

[3] https://github.com/AliveToolkit/alive2/commit/fae975049342fb81940d427d8575e291595733f6
[4] https://github.com/AliveToolkit/alive2/commit/8bf86254e2c799526e0f1dc99eefb9c7da646c1d

LLVM pass	$k = 1$	σ	$k = 2$	σ	$k = 4$	σ
Transforms/LoopSimplifyCFG	1.82s	0.11	3.36s	0.09	11.27s	0.17
Transforms/LoopUnrollAndJam	0.74s	0.03	0.76s	0.03	0.84s	0.03
Transforms/LoopRotate	27.74s	0.62	23.79s	0.36	87.79s	0.91
Transforms/LoopDeletion	1.41s	0.04	1.70s	0.18	1.75s	0.13

Table 5.2 Running llvm-lit over LLVM's test suite with the unroll factor of k before the patch.

LLVM pass	$k = 1$	σ	$k = 2$	σ	$k = 4$	σ
Transforms/LoopSimplifyCFG	1.89s	0.24	3.51s	0.27	11.18s	0.48
Transforms/LoopUnrollAndJam	0.80s	0.03	0.81s	0.10	0.90s	0.04
Transforms/LoopRotate	30.17s	0.10	23.62s	0.09	92.94s	0.89
Transforms/LoopDeletion	1.41s	0.03	1.54s	0.07	1.73s	0.05

Table 5.3 Running llvm-lit over LLVM's test suite with the unroll factor of k after the patch.

It is clear that our solution did not slow down Alive2 on these benchmarks even with higher unroll factors.

5.3 Summary

Our solution was able to fix at least six false positives in the LLVM test suite, there may be others that could have been left unexposed. Furthermore, we accomplished this with near-zero overhead in terms of performance and very little added code to Alive2.

While our solution was successful in solving the problem of false alarms in Alive2, we have discovered some deeper issues related to basic block ordering in Alive2 that we discuss in Chapter 6. Although there was insufficient time to address them as our work is mainly of theoretical character, we believe our analysis provides a solid ground for future work in this area and overall improvement of Alive2's loop algorithms.

Chapter 6

Further results and future work

In this chapter, we present further results we have obtained from our analysis. They were mostly orthogonal to the main line of our work, but some might prove to be promising future directions.

6.1 Correct ordering of basic blocks

Recall that the problem that caused false alarms in Alive2 was an incorrect ordering of basic blocks for duplication in the unrolling algorithm. Hence it is natural to ask whether we can somehow generalize the ordering of the CFG so that basic blocks are already in the correct order. This is still a subject of debate with the authors of Alive2 but through discussions with them, we came up with a few methods. Nevertheless, we eventually found a counterexample for each of them, but we believe it is a good idea to mention them here as they provide solid ground for future research in this area.

6.1.1 Reverse postorder and dominators

A standard ordering for solving dataflow problems in compilers is *reverse postorder (RPO)*. But with loops involved, there are usually a few possible RPOs. Nuno Lopes of Alive2 suggested[1] that we could take inspiration from NewGVN in LLVM[2] where they solve this problem by ordering the dominator tree nodes by RPO which is supposed to break the ties. But consider the example in Figure 6.1.

[1] Personal communication with Nuno Lopes

[2] `https://github.com/llvm/llvm-project/blob/main/llvm/lib/Transforms/Scalar/NewGVN.cpp#L3445-L3467`

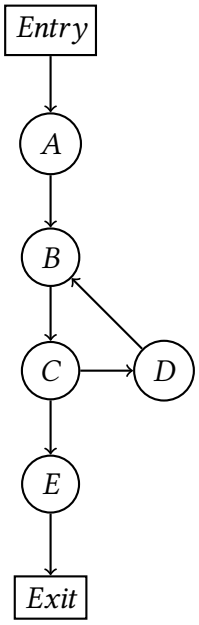

Figure 6.1 Counterexample for RPO with dominators

The nodes E and D are incomparable according to RPO and are independent in the dominator tree as well, but we would like E to come before D. This may also suggest there is a bug in LLVM's NewGVN but we did not have the time to investigate it further.

6.1.2 Reverse post-dominators

Post-dominance in a CFG is a concept closely related to dominance that we defined in the first chapter (Definition 5):

Definition 14 (Post-dominance [15])**.** *We say that basic block B* post-dominates *basic block A if B is on every path from A to Exit. We denote this as B* postdom *A. If $A \neq B$, then we say that B* strictly post-dominates *A. B is an* immediate post-dominator *of A if B strictly dominates A and does not strictly post-dominate any other basic block that strictly post-dominates A.*

We can see that post-dominance has many of the same properties that dominance has, for example, that immediate post-dominators form a tree. We can also immediately observe that post-dominance on a CFG G is essentially dominance with the edges reversed and *Entry* and *Exit* swapped. And thus, if we have an algorithm that computes dominators, we can also easily change it to compute post-dominators as we can see in Algorithm 5.

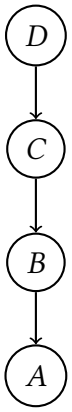

Figure 6.2 Post-dominator tree

Algorithm 5 Algorithm for computing the post-dominator tree

```
1: function ComputePostDominatorTree(G: a DAG)
2:     G′ ← G with reversed edges
3:     T ←ComputeDominatorTree(G′)
4:     return T
5: end function
```

The post-dominator tree of our running example is very simple because it produces a linear order as shown in Figure 6.2.

From the tree, we can construct the post-dominator order. For example in this case, it would be *D postdom C postdom B postdom A* and to obtain the reverse post-dominator ordering we simply reverse this order.

Postdominators are almost by definition very well suited for capturing a certain kind of dependency which is not broken by loops. Nevertheless, the requirement that a node that postdominates a node *A* must be on *all* paths from *A* to *Exit* is too strict for our uses. Consider the example in Figure 6.3. Neither *E* nor *F* postdominate *A* but we would like them to come only after *A*. Thus reverse postdominators by themselves cannot be our answer but they may be a stepping stone.

6.2 Failure to identify nested loops with shared headers

Some optimizations can cause the headers of an inner loop and its outer loop to be merged when performed on nested loops. Our running example in Chapter 1 Figure 1.1 might look like the CFG in Figure 6.4 with merged headers. One such

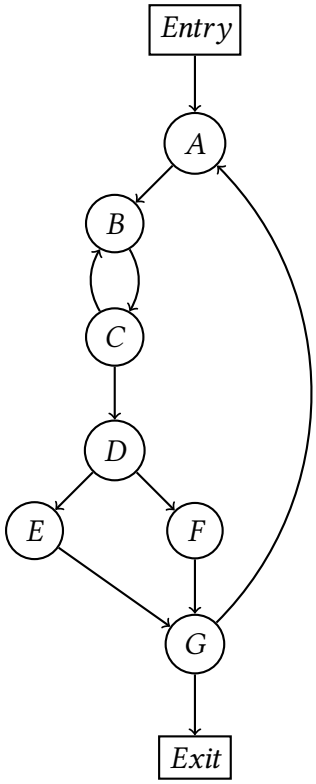

Figure 6.3 Counterexample for reverse post-dominator ordering

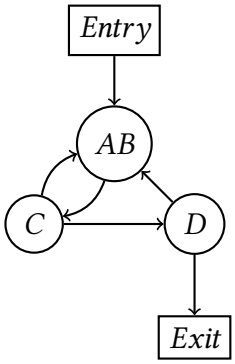

Figure 6.4 Running example with merged headers

Listing 6.1 Before jump threading

```
void foo(int a, int b,
    int c) {
  if (a && b)
    foo ();
  if (b || c)
    bar ();
}
```

Listing 6.2 After jump threading

```
void foo(int a, int b,
    int c) {
  if (a && b) {
    foo ();
    goto skip;
  }
  if (b || c) {
skip:
    bar ();
  }
}
```

Figure 6.5 A program before and after jump-threading

optimization that can cause this is called *jump threading*. Jump threading is a compiler optimization that turns conditional branches into unconditional ones thus speeding up execution but at the price of increased code size. A canonical example where jump threading can be used is two overlapping conditions (i.e. $a \&\& b$ implies $b \| c$) as shown in Figure 6.5.

6.2.1 Possible fixes

1. Havlak presents another algorithm in his paper [10] which is designed to overcome the issue of shared headers by preprocessing the loops as seen in 6.6.

```
procedure fix_loops(G_CF, START)
    number vertices of G_CF using depth-first search from START,
        numbering in preorder from 1 to |N_CF| and saving last[*]
    for w := 1 to |N_CF| do
        redBackIn[w] := otherIn[w] := ∅
        foreach edge (v, w) entering w do
            if (w ⪯ v) then add (v, w) to redBackIn[w]
                       else add (v, w) to otherIn[w]
        if (redBackIn[w] ≠ ∅) and (|otherIn[w]| > 1) then
            insert new node w' into G_CF immediately before w, unnumbered
            create new edge (w', w)
            foreach edge (v, w) ∈ otherIn[w] do
                delete (v, w) and create otherwise identical edge (v, w')
```

Figure 6.6 FixLoops algorithm by Havlak [10]

In brief, it inserts a new intermediate node when an irreducible loop is found which is followed by updating both the forward and backward edge sets as well as preorder numbering of the new CFG. The new intermediate nodes give more choices for headers of irreducible loops and so more loops can be identified in this way.

It is also one of the suggestions by de Vos [16] on how to deal with this problem though he mentions implementing the procedure would require a significant amount of time.

2. Another way to separate shared headers is to implement an algorithm similar to nested loop separation algorithm in LLVM[3].

Initially, we tried to make Alive2 fail by constructing source-target pairs where the source would be a nested loop and the target would contain the same nested loop only with the headers merged. Nonetheless, we were not successful in making Alive2 fail to identify loops with merged headers even after a multitude of attempts either with jump threading or artificially constructed examples, so this issue might not occur in practice.

6.3 Future work

6.3.1 SSA reconstruction

As we describe in Section 3.1.2, Alive2 does not properly reconstruct the SSA form after unrolling, instead opting for introducing a new stack variable. This

[3] `https://llvm.org/doxygen/LoopSimplify_8cpp_source.html#l00220`

might result in a wrong ϕ node placement. There is a failing example in Alive2 that demonstrates this issue[4]:

Listing 6.3 Bug in ϕ node placement

```
; Transforms/LoopFusion/cannot_fuse.ll

define float @test(float* nocapture %a, i32 %n) {
entry:
  %conv = zext i32 %n to i64
  %cmp32 = icmp eq i32 %n, 0
  br i1 %cmp32, label %for.cond.cleanup7, label %for.body

for.body:                                         ; preds = %for
    .body, %entry
  %i.034 = phi i64 [ %inc, %for.body ], [ 0, %entry ]
  %sum1.033 = phi float [ %add, %for.body ], [ 0.000000e+00, %
      entry ]
  %idxprom = trunc i64 %i.034 to i32
  %arrayidx = getelementptr inbounds float, float* %a, i32 %
      idxprom
  %0 = load float, float* %arrayidx, align 4
  %add = fadd float %sum1.033, %0
  %inc = add nuw nsw i64 %i.034, 1
  %cmp = icmp ult i64 %inc, %conv
  br i1 %cmp, label %for.body, label %for.body8

for.body8:                                        ; preds = %for
    .body, %for.body8
  %i2.031 = phi i64 [ %inc14, %for.body8 ], [ 0, %for.body ]
  %idxprom9 = trunc i64 %i2.031 to i32
  %arrayidx10 = getelementptr inbounds float, float* %a, i32 %
      idxprom9
  %1 = load float, float* %arrayidx10, align 4
  %div = fdiv float %1, %add
  store float %div, float* %arrayidx10, align 4
  %inc14 = add nuw nsw i64 %i2.031, 1
  %cmp5 = icmp ult i64 %inc14, %conv
  br i1 %cmp5, label %for.body8, label %for.cond.cleanup7

for.cond.cleanup7:                                ; preds = %for
    .body8, %entry
  %sum1.0.lcssa36 = phi float [ 0.000000e+00, %entry ], [ %add,
      %for.body8 ]
  ret float %sum1.0.lcssa36
  }
```

[4] https://github.com/AliveToolkit/alive2/issues/796

After unrolling in Alive2, we have a wrong `phi` argument in `for.cond.cleanup7`:

Listing 6.4 Wrong ϕ after unroll

```
%for.cond.cleanup7:
  %sum1.0.lcssa36 = phi float [ 0.000000, %entry ], [ %add#phi
      #0, %for.body8 ], [ %add, %for.body8#2 ]
```

But even `%for.body8#2` should point to the ϕ of `%add`, like `%for.body8`.

In the end, this issue seemed to be orthogonal to our work because it does not involve nested loops and we could not replicate a similar issue that would be specific only to nested loops (barring artificially nesting the loop). Nevertheless, this might prove to be an interesting future direction.

The issue of ϕ placement might be solved by implementing a better SSA formation algorithm, e.g. the one by Braun et al. [41]. This would require non-trivial additional effort, it was only recently submitted as a patch to GCC as the result of Filip Kastl's bachelor thesis [42].

6.4 Other minor fixes in Alive2

Alive2 has the ability to generate the CFG of the source and target after unroll by specifying the `-dot-cfg` option. Nevertheless, some examples we were initially testing were crashing in a very non-deterministic way with `-dot-cfg`. We eventually discovered that Alive2 does not allocate enough space for the sink basic block[5], and hence eventually the program wants to access uninitialized memory (which is undefined behavior). We reported this problem and it was fixed[6].

[5] Recall that Alive2 removes backedges by redirecting them to a special basic block called *sink*.
[6] https://github.com/AliveToolkit/alive2/commit/c4e4159addc86c9a38b48ad6c79f43cd59410f40

Conclusion

The goal of this thesis was to analyze loop algorithms in Alive2 with the aim of identifying the root cause of false alarms in Alive2 occurring in the presence of certain types of loops. We have succeeded by pinpointing the concrete bug in Alive2 loop analysis algorithm and moreover, we submitted a pull request which was merged into Alive2 and is currently a part of the framework. While the fix itself may have been simple, it solved a large class of problems in Alive2. Moreover, the issues did not occur deterministically hence many of those may have stayed dormant for a long time. To be specific, our code fixed at least six failing tests in Alive2.

Through our analysis, we discovered how important basic block ordering is in preventing false alarms and we believe that our theoretical work provides a good grounding for future work on problems in this area.

Our work has helped Alive2 move closer to its goal of zero false alarms, which makes it a lot more usable for LLVM developers around the world.

Bibliography

[1] Scott Bauer, Pascal Cuoq, and John Regehr. "Deniable backdoors using compiler bugs". In: *International Journal of PoC|| GTFO, 0x08* (2015), pp. 7–9.

[2] Xuejun Yang et al. "Finding and understanding bugs in C compilers". In: *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. 2011, pp. 283–294.

[3] Eric Eide and John Regehr. "Volatiles are miscompiled, and what to do about it". In: *Proceedings of the 8th ACM international conference on Embedded software*. 2008, pp. 255–264.

[4] Chengnian Sun, Vu Le, and Zhendong Su. "Finding compiler bugs via live code mutation". In: *Proceedings of the 2016 ACM SIGPLAN international conference on object-oriented programming, systems, languages, and applications*. 2016, pp. 849–863.

[5] Jiawei Liu et al. "Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation". In: *Advances in Neural Information Processing Systems* 36 (2024).

[6] Nuno P Lopes et al. "Alive2: bounded translation validation for LLVM". In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 2021, pp. 65–79.

[7] *Bugs Found by Alive2*. `https://github.com/AliveToolkit/alive2/blob/master/BugList.md`. [Accessed 10-12-2023].

[8] *How to contribute to LLVM: Proving the transform correct*. `https://developers.redhat.com/articles/2022/12/20/how-contribute-llvm`. [Accessed 10-12-2023].

[9] Erik Seligman, Tom Schubert, and MV Achutha Kiran Kumar. *Formal verification: an essential toolkit for modern VLSI design*. Elsevier, 2023.

[10] Paul Havlak. "Nesting of reducible and irreducible loops". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 19.4 (1997), pp. 557–567.

[11] Jeff Erickson. *Algorithms*. 2023.

[12] Robert Endre Tarjan. "Edge-disjoint spanning trees and depth-first search". In: *Acta Informatica* 6 (1976), pp. 171–185.

[13] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. eng. Reading: Addison-Wesley, 1986. ISBN: 0-201-10088-6.

[14] Thomas Lengauer and Robert Endre Tarjan. "A fast algorithm for finding dominators in a flowgraph". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 1.1 (1979), pp. 121–141.

[15] Steven Muchnick. *Advanced compiler design implementation*. Morgan kaufmann, 1997.

[16] Kevin Jacobus de Vos. "Translation Validation for the LLVM Compiler". Master's Thesis. Instituto Superior Técnico, Universidade de Lisboa, 2020.

[17] *LLVM Loop Terminology (and Canonical Forms)*. `https://llvm.org/docs/LoopTerminology.html`. [Accessed 10-12-2023].

[18] Nuno P Lopes. *A Decade Verifying LLVM, or How to Retrofit Soundness in Industrial Software, Workshop on Dependable and Secure Software Systems'22*. `https://web.ist.utl.pt/nuno.lopes/pres/a-decade-verifying-llvm.pdf`. [Accessed 10-12-2023]. 2022.

[19] Xavier Leroy et al. "CompCert-a formally verified optimizing compiler". In: *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*. 2016.

[20] *The Coq Proof Assistant: Welcome!* `https://coq.inria.fr/`. [Accessed 10-12-2023].

[21] M Siegel, A Pnueli, and E Singerman. "Translation validation". In: *TACAS*. 1998, pp. 151–166.

[22] Hanan Samet. *Automatically proving the correctness of translations involving optimized code*. Vol. 259. Citeseer, 1975.

[23] Chris Lattner and Vikram Adve. "LLVM: A compilation framework for lifelong program analysis & transformation". In: *International symposium on code generation and optimization, 2004. CGO 2004.* IEEE. 2004, pp. 75–86.

[24] Chris Lattner. *The Architecture of Open Source Applications (Volume 1): LLVM*. `https://aosabook.org/en/v1/llvm.html`. [Accessed 10-12-2023].

[25] *LLVM Language Reference Manual*. `https://llvm.org/docs/LangRef.html`. [Accessed 10-12-2023].

[26] Andrew W Appel. "SSA is functional programming". In: *Acm Sigplan Notices* 33.4 (1998), pp. 17–20.

[27] Jeffery von Ronne, Ning Wang, and Michael Franz. "Interpreting programs in static single assignment form". In: *Proceedings of the 2004 workshop on Interpreters, virtual machines and emulators*. 2004, pp. 23–30.

[28] Fabrice Rastello and Florent Bouchez Tichadou. *SSA-based Compiler Design*. Springer Nature, 2022.

[29] *Gimple*. `https://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html`. [Accessed 10-12-2023].

[30] *The Java HotSpot Performance Engine Architecture*. `https://www.oracle.com/java/technologies/whitepaper.html`. [Accessed 10-12-2023].

[31] *V8 JavaScript Engine*. `https://v8.dev/`. [Accessed 10-12-2023].

[32] *LLVM's Analysis and Transform Passes*. `https://llvm.org/docs/Passes.html`. [Accessed 10-12-2023].

[33] Leonardo De Moura and Nikolaj Bjørner. "Satisfiability modulo theories: An appetizer". In: *Brazilian Symposium on Formal Methods*. Springer. 2009, pp. 23–36.

[34] Stephen A Cook. "The complexity of theorem-proving procedures". In: *Proceedings of the third annual ACM symposium on Theory of computing*. 1971, pp. 151–158.

[35] Leonardo de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340. ISBN: 978-3-540-78800-3.

[36] Tao Wei et al. "A new algorithm for identifying loops in decompilation". In: *International Static Analysis Symposium*. Springer. 2007, pp. 170–183.

[37] Robert Endre Tarjan. *Data structures and network algorithms*. SIAM, 1983.

[38] Robert Tarjan. "Testing flow graph reducibility". In: *Proceedings of the fifth annual ACM symposium on Theory of computing*. 1973, pp. 96–107.

[39] Zdeněk Dvořák. *[lno] Enable unrolling/peeling/unswitching of arbitrary loops*. `https://gcc.gnu.org/legacy-ml/gcc-patches/2004-03/msg02212.html`. [Accessed 10-12-2023]. 2004.

[40] Chris Lattner. *Loop Optimizer Notes*. `https://nondot.org/sabre/LLVMNotes/LoopOptimizerNotes.txt`. [Accessed 10-12-2023]. 2004.

[41] Matthias Braun et al. “Simple and efficient construction of static single assignment form”. In: *Compiler Construction: 22nd International Conference, CC 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings 22*. Springer. 2013, pp. 102–122.

[42] Filip Kastl. “An alternative SSA construction algorithm for GCC”. Bachelor’s Thesis. Univerzita Karlova, Matematicko-fyzikální fakulta, Katedra aplikované matematiky, 2023.

[43] *Alive2 README*. `https://github.com/AliveToolkit/alive2/blob/master/README.md`. [Accessed 10-12-2023].

Appendix A

Compiling Alive2

Here, we provide instructions on how to compile Alive2. More thorough instructions are available in Alive2 readme [43] on which this chapter is based. Let us also mention that it is not necessary to compile Alive2 in order to run translation validation, there is an Alive2 instance hosted here: `https://alive2.llvm.org/ce/`.

A.1 Building Alive2

In order to build Alive2, you need to have the following prerequisites:

- cmake (`https://cmake.org`)
- gcc (`https://gcc.gnu.org`) or clang (`https://clang.llvm.org`)
- re2c (`https://re2c.org/`)
- Z3 (`https://github.com/Z3Prover/z3`)
- LLVM (`https://github.com/llvm/llvm-project`)
- hiredis (`https://github.com/redis/hiredis`)

Afterward, we can clone and build Alive2:

```
git clone git@github.com:AliveToolkit/alive2.git
cd alive2
mkdir build
cd build
cmake -GNinja -DCMAKE_BUILD_TYPE=Release ..
ninja
```

If you wish, you may checkout our commit specifically after cloning Alive2: `git checkout 8bf8625`.

A.2 Running translation validation

Alive2's `opt` and `clang` translation validation requires a build of LLVM with RTTI and exceptions turned on. LLVM can be built targeting X86 in the following way:

```
cd ~/llvm-project/llvm/
mkdir build
cd build
cmake -GNinja -DLLVM_ENABLE_RTTI=ON -DLLVM_ENABLE_EH=ON -
    DBUILD_SHARED_LIBS=ON -DCMAKE_BUILD_TYPE=Release -
    DLLVM_TARGETS_TO_BUILD=X86 -DLLVM_ENABLE_ASSERTIONS=ON -
    DLLVM_ENABLE_PROJECTS="llvm;clang" ../llvm
ninja
```

Alive2 should then be configured and built as follows:

```
cd ~/alive2/build
cmake -GNinja -DCMAKE_PREFIX_PATH=~/llvm-project/llvm/build -
    DBUILD_TV=1 -DCMAKE_BUILD_TYPE=Release ..
ninja
```

In Chapter 2, we described how to use `alive-tv` for a source-target pair of programs and in Chapter 5, we mentioned Alive2 plugin for LLVM lit. For instance, if we want to test a single LLVM testcase using lit, we can do the following:

```
./llvm-project/build/bin/llvm-lit -s -vv -Dopt=~/alive2/build/opt-alive
.sh llvm-project/llvm/test/Transforms/LoopUnrollAndJam/multiple_exit_blocks
.ll
```

The output should be very simple:

```
Testing Time: 0.17s
 Passed: 1
```

If we want to perform translation validation on a whole pass, we can do the following:

```
./llvm-project/build/bin/llvm-lit -s -vv -Dopt=~/alive2/build/opt-alive
.sh llvm-project/llvm/test/Transforms/LoopUnrollAndJam.
```