IMPROVING COMPILER CONSTRUCTION USING FORMAL METHODS

by

Jubi Taneja

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

School of Computing

The University of Utah

December 2022

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of **Jubi Taneja**

has been approved by the following supervisory committee members:

John Regehr,	Chair(s)	**14 May 2021**	Date Approved
Mary Hall,	Member	**14 May 2021**	Date Approved
Matthew Flatt,	Member	**14 May 2021**	Date Approved
Eric Eide,	Member	**14 May 2021**	Date Approved
Nuno Lopes,	Member	**14 May 2021**	Date Approved

by **Mary Hall**, Chair/Dean of

the Department/College/School of **Computing**

and by **David B. Kieda**, Dean of The Graduate School.

ABSTRACT

Programming languages are becoming more high-level and processors are moving towards more specialization. This trend has increased the gap between high-level programming languages and low-level processors. It is the responsibility of a compiler to bridge the gap between the two. The key contribution of this dissertation is to use formal techniques to build parts of the compiler to make them more effective and more correct.

Static analyses compute properties of programs that are true in all executions. Compilers use these properties to justify optimizations such as dead code elimination. Each static analysis in a compiler should be as precise as possible while remaining sound and being sufficiently fast. Unsound static analyses typically lead to miscompilations, whereas imprecisions typically lead to missed optimizations. Neither kind of bug is easy to track down. My contributions include a collection of algorithms that use an Satisfiability Modulo Theories (SMT) solver to compute sound and maximally precise static analysis results for fragments of code in the LLVM intermediate representation, enabling automated testing of the corresponding static analysis code in LLVM. Some of the abstract domains that we target have convenient properties enabling efficient search for the most precise result, but one of them (integer ranges) appears to lack these properties; in this case we compute the most precise result using a CEGIS (Counterexample Guided Inductive Synthesis) loop. Our test cases include all of the LLVM IR generated while compiling SPEC CPU 2017. I found many imprecisions, some of which have resulted in patches being made to the LLVM compiler. My work is important because the correctness and efficiency of a large amount of real-world code in a number of programming languages depends on the soundness and precision of static analyses in LLVM.

Peephole optimizations are hard to get right as they involve many corner cases. They are hard to extend because developers are not using domain specific language to define the rewrite rules, automatically generate and verify them for soundness. My contributions include designing an automatic peephole optimizer that is driven by a superoptimizer. It makes use of a superoptimizer's intermediate representation to define the peepholes and transform them to another IR and it guarantees correctness as these optimizations are verified by a SMT solver.

For my Lalaji, who has always encouraged me to dream big.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

This dissertation would not have been possible without the help, support, and guidance of many people. To begin with, I am grateful to my advisor, John Regehr, for being very supportive and kind throughout my PhD journey. John has spent countless hours with me in answering my questions, having research discussions, reviewing paper drafts, proposal, dissertation, attending practice talks, and a lot more until I felt comfortable and confident about my own work. Thank you for teaching me to deal with the rejections with a positive attitude. He has patiently read all my drafts much more carefully than I could do because of a rare vision problem. He has taught me all about correctness both in the Software Systems and English language. He has ensured that I feel safe everywhere, be it at an institute level or outside. Thank you is not enough for a person who has taught me all about and beyond research.

Thanks to my committee members–Eric Eide, Matthew Flatt, Mary Hall, and Nuno Lopes–for their useful advice, constructive feedback, and interesting questions they raised at my proposal defense and final defense. I would like to thank Nuno Lopes for spending long hours teaching me formal methods, LLVM's dataflow analyses, and making me learn how to pick the right questions to write a good paper.

My research exploration on the Cranelift compiler and Wasmtime runtime system would not have been possible without my internship mentor, Dan Gohman, and later a longer collaboration with Nick Fitzgerald. Thank you for helping me to learn Rust programming language and supporting my work and ideas. Dan, I am grateful to you for long discussion hours that we had during my internship at Mozilla. With each and every feedback that I received from you, I could see my own progress not just technically, but personally as well.

Thank you, Nick, for collaborating with me on the automatic peephole optimizer project and helping me till the last moment of my dissertation submission. I honestly cannot express in words how much I appreciate your strong support, kindness, and time through the toughest year–2020. Thanks for listening to my crazy ideas in our weekly meetings, and

helping me bypass all the hurdles–programming, designing, benchmarking and writing–and extending me all the moral support. Thank you for in-depth discussions particularly on benchmarking, introducing me to all kinds of WebAssembly related tools, and reviewing my dissertation chapter.

Thank you to my external mentor, Cristina Cifuentes, for the encouragement and valuable feedback on the introduction chapter of my dissertation.

My entry into the world of Compilers would not have been possible without my former advisor, Uday Khedker. Thank you, Sir, for giving an opportunity to work with you during my undergrad when I had no prior knowledge on this subject. Thank you for all the valuable lessons and time that you have given me to help me grow professionally and personally both.

My formal learning of Compilers continued at the University of Utah when I enrolled in a couple of graduate-level courses taught by Prof. Matt Might. Thank you for widening my perspectives about research and career choices. I was too narrowly focused on one topic when I started graduate school, but I learned to think beyond my research with every class I attended. I learned to find the value and purpose of my research and personal goals. Thank you, Matt, for inspiring me with your contributions and consideration toward humanity.

I shared my dream to pursue a graduate degree with my Grandma, my *Badi Mummy*, but she passed away a couple of months before I left for US. She did not understand much but whenever she used to see an airplane passing over our home, she used to say that I am in that airplane. I wish she was here to see this day because her prayers and blessings have made this day possible.

I am dedicating my dissertation to my Grandfather, my *Lalaji*, who has been most excited and encouraging in our entire family for my dissertation and doctorate degree from the United States of America. I am writing my acknowledgment for my Lalaji in Hindi so he can at least read this part by himself.

आदरणीय लालाजी, शब्दों में ब्यान करना मुश्किल है जो भरोसा और विश्वास आपने मुझपे दिखाया है| मैं तह दिल से आपका धन्यवाद करती हूँ| मुझे आज भी याद है जब मैंने आपको अमेरिका में पढ़ने का सपना बताया था| आपने बेझिझक और बिना कोई सवाल जवाब किये मुझे प्रोत्साहित किया और कहा कि आपको मुझपे बहुत विश्वास है कि मैं आगे बडूगी और अपने सब सपने पूरे करुँगी| लालजी, मैं बहुत ज़्यादा किस्मत वाली हूँ जो मुझे आप और बड़ी मम्मी के जैसे दादा-दादी मिले| अपनी हर खवाइश आपको बताना फिर एक आदत ही बन गई क्योंकि हमेशा ये लगता था कि चाहे मेरा यक़ीन खुद पर कम हो जाए, लेकिन आप हमेशा मुझे प्रेरित करेंगे| मेरा ये सोचना आपने

कभी गलत साबित नहीं होने दिया| पूरे परिवार में केवल आप ही हो जिसने हमेशा मुझसे इतने उत्साहित होकर पुछा की मेरी किताब कब छपेगी| आपका हर परिस्थिति को सरलता से देखने का नज़रिया बहुत विचित्र है| काफ़ी बार जब मुझे लगा की मुझसे ये काम पूरा नहीं होगा तो सरलता से सोचने की कोशिश की मैंने और सब सच में आसान लगने लगा| काश आज बड़ी मम्मी भी होते ये दिन देखने के लिए हम सबके साथ लेकिन मैं जानती हूँ कि उनका आशीर्वाद मेरे साथ है| मैं ये शोध निबंध (जिसे आप मेरी किताब कहते हैं) आपको समर्पित करती हूँ|

It would be a very small thing to say thank you to my Mom (Mrs. Shakuntla Taneja), Dad (Mr. Jagdish Taneja), and brother (Rashin Taneja) who stood by my side through thick and thin of my professional and personal life. Your unconditional love, support, and blessings made this day possible. I can never forget the sacrifices you have made in your entire life to educate me and to allow me to be a free bird. Mummy, thank you for always reminding me that I am not meant to quit anything in-between. Daddy, thank you for believing in me and inculcating patience in me that I needed the most in my PhD journey. Rashin, thank you for adjusting your entire childhood and teenage life with my geeky and nerdy habits. I wish I had given a lot more time to you. I know you have missed me so much and how we both have grown up living so far away in the past one decade. You have taken care of me as my elder brother and kept me sane with your bad jokes and words of wisdom both. Thanks to my entire family for their love and blessings.

I would like to thank my husband, Gaurav Gupta, for his endless support and encouragement throughout my PhD. Thank you for burning the midnight oil together with me on every small to big deadlines that I had to meet. I will never forget your zillion round trips from San Francisco to Salt Lake City to help me fight the isolation struggle. Thank you for reading all the papers, blogs, articles, review comments, and drafts to me when my eyes were tired, and for listening to an infinite amount of the practice talks to prepare me for the best. Thank you for being so patient with me as this PhD life continued to stretch for longer, and for believing in me whenever I failed.

Thanks to my co-author, Zhengyang Liu, who helped me with testing static analyses work. Thanks to my lab mates who brought happiness to our lab and assisted me. I truly appreciate all instructors, seniors and staff at the University of Utah for answering all my technical and non-technical questions, and friends for helping and cheering for me. Thank you to the wonderful women in the Women in Compilers and Tools working group for igniting that fire in me to accomplish my goals. Thanks to all my high-school and undergrad teachers for pushing my boundaries and building a strong foundation.

I also want to acknowledge the PL/Systems Twitter community for sharing awesome ideas, cute pet pictures, and mostly nerdy and bad jokes. Finally, I want to thank my friends working at the coffee shops and book stores in Salt Lake City, San Francisco and San Mateo for creating such a perfect ambiance that allowed me to work for long hours, and for keeping a check on me when I didn't show up in between.

This would not have been possible if you all had not been there for me. Thank you, everyone.

CHAPTER 1

INTRODUCTION

The correctness and efficiency of a large amount of real-world code in a number of programming languages depends on the quality of code generated by the production-grade compilers. While compiler developers are putting in a lot of effort in constructing compilers, especially towards writing compiler optimizations in order to generate efficient code, many optimizations are either missed or have bugs in them. Compiler developers may choose to spend more time to find the missing optimizations and track down the bugs manually. However, they may not always win. To meet the requirements of generating correct yet effective code, developers need to shift their focus towards using formal methods-based techniques, such as search techniques and SMT solver-based tools for discovering the missing optimizations, verifying their correctness, and implementing them manually at compiler-compile time.

1.1 Motivation

A compiler is a program that translates the code written in a high-level programming language to low-level instructions that can be directly executed on a machine. Earlier programmers used to code in low-level language to perform any task on a machine. It was feasible to follow this strategy for simpler programs, but as users' requirements increased, it became complicated to program in low-level programming languages. Many new programming languages were designed that were more human-readable and had abstractions in them so that generic programs could be written more efficiently. As users' requirements kept growing, software became more extensive, and thus the programming languages became more high-level. On the other side, hardware also started improving as more and more functionality was added to it and the architecture changed, but users could not leverage the benefits of growing hardware because there was a gap between high-level programming languages and the underlying hardware chips on which programs were executed. Thus, compilers were designed to bridge the semantic gap between high-level programming languages and machines. The goal of writing compilers at first started merely as an interpreter that only translated one statement of the program to machine code without being able to look at the whole program. However, with growth in both software and hardware industry, the goal of compilers shifted towards generating efficient code; reducing compile time and run time; and producing less buggy code.

In early days compiler design was affected by the complexity of source programming language and resource availability. Lack of memory led to the need to pass through source code more than once that influenced a multi-pass design of compiler construction. On the other side, compilers written for less complicated programming languages were single-pass compilers. Single-pass compiler design got extended by splitting compiler construction into different phases so that it was easy to implement and make changes to each stage individually as the complexity of programming languages grew. Modern compilers have three main phases: frontend, middle-end, backend.

The frontend verifies syntax and semantics according to the source language, and transforms it into an intermediate representation (IR). The different aspects of frontend involve lexical analysis, syntax analysis, and semantic analysis (for instance, type checking, scope resolution). The middle-end is a crucial phase in compiler design that performs optimizations on the IR that are independent of the target machine architecture. These optimizations improve code quality, and they are shared across different source languages. Static analysis in this phase gathers information at each program point that drives optimizations, like dead code elimination, constant propagation, constant folding, and loop transformation. Thus, the middle-end generates a functionally equivalent IR, i.e., faster or smaller than an input IR. The backend performs machine-dependent optimizations that vary from one architecture to another. The most common optimizations in this phase are register allocation and instruction selection/scheduling. This phase eventually generates native machine code that can be executed on the target CPU.

Compiler developers spend a lot of time implementing optimizations in the middle-end of a compiler, but many optimizations are still missing, and some are found to be buggy. Some of the reasons that cause these problems are:

- developers cannot intuitively think of all possible optimizations;

- developers cannot implement all optimizations manually because of lack of time;
- developers do not want to pay the compile time cost of running optimizations that might not pay off;
- optimizations are hard to reason about manually as they involve a lot of corner cases;
- static analysis that triggers these optimizations have soundness and precision issues which either lead to bugs in optimizations or not trigger them at all;
- peephole optimizations are hard to extend because of the naive matching strategies as implemented in LLVM's instruction combiner, and several other compilers do not make much use of a domain-specific language or an intermediate representation to define the peephole optimizations so that they can be auto-generated and auto-verified for soundness.

This dissertation is an effort towards advancing the state of the art in compiler construction to solve some problems discussed above by assisting compiler developers with search techniques and SMT solver-based tools for discovering missing optimizations, verifying their correctness, and implementing them automatically at compiler-compile time. In other words, this dissertation aims to improve the static analysis and peephole optimization pass of compiler development by reducing the manual effort and time a developer has to put towards the implementation otherwise, and by making analyses sound and more precise. This dissertation is an effort towards convincing developers to define the semantics of instructions, dataflow analyses they trust to perform optimizations so that any compiler phase can be tested for soundness.

1.2 Dissertation Overview

Peephole optimizations work on small portions of code. They are based on replacement rules where a slow or more extensive set of instructions is replaced with a fast or smaller set. This optimization is applied repeatedly until a steady state is reached, where no more modification is needed.

Developers and researchers have put effort into improving peephole optimizers in existing compilers. The first step towards ease of implementation is to rewrite rules using a domain-specific language (DSL). The idea of abstraction from the high-level language in which the compiler is usually implemented to a DSL allows developers to add new optimizations easily. For instance, GCC is implemented in C, but some of its peephole optimizations are specified in a separate DSL-based file (match.pd). This match-and-simplify file in GCC is consumed by a different tool to generate the corresponding C implementation of the optimizer automatically. However, the entire peephole optimizer in this compiler is not implemented using the rewrite rules. Furthermore, it leaves the problem of verifying these optimizations unsolved. On the other side, LLVM does not follow the rule rewriting approach and implements the entire peephole optimizer in 36,000 lines of C++.

A different approach is to define a language to enable automatic verification of transformations written in an intermediate representation. This technique promises correctness because it describes the formal semantics of the IR and uses an SMT solver for verification. However, this approach still involves manually feeding and implementing new transformations.

The last question that remains is to figure out an approach that can help us discover missing optimizations automatically, and ensure their correctness. Superoptimization uses a search procedure, a cost function, and an equivalence checker to find better(or even optimal) code sequences in a given program. The search procedure can be enumerative, stochastic, or counter-example guided inductive synthesis. The cost function can be defined in the context of reducing the code size or execution time. A superoptimizer's scope is limited to a smaller number of instructions, but we want to extend the amount of code it can effectively interact with so that we can discover more optimizations. One solution is to integrate compilers' dataflow analyses with a superoptimizer. We further want to ensure that we learn the precise and sound dataflow facts from the compiler and discover the correct transformations. The final problem to ease the implementation of the learned new optimizations is solved by automatic code generation strategy at the compiler-compile time. We implement a compiler-compiler to implement a peephole optimizer at the compile time automatically.

The thesis statement that this dissertation supports is that *Techniques based on formal methods can be used to make production-grade compilers more correct and more effective.*

1.3 An Overview of Souper

Souper is a synthesizing superoptimizer that automatically derives novel middle-end optimizations for LLVM. Souper was initially designed to implement an extension of Gulwani et al.'s counter-example guided inductive synthesis (CEGIS) algorithm [1], allowing it to synthesize LLVM's instructions. Later, Souper's synthesis design was extended to use enumerative synthesis.

Souper's fundamental abstraction is a directed acyclic graph. It works within a function but looks across basic blocks, taking phi nodes into account, and representing branches by adding an explicit path condition corresponding to each branch it passes while extracting LLVM into Souper IR. It looks for instructions that return an integer-typed value and constructs a root of Souper's left-hand side (LHS). The remaining LHS is formed by following the backward dataflow edges from the root instruction. The main limitations are that it cannot see through memory references, function calls, or multiple loop iterations. It does not extract cycles or calls, so there are no termination issues on our side.

Souper represents its synthesized result in a right-hand side (RHS). The right-hand side is a DAG that may refer to values in its corresponding left-hand side. Souper's operations closely follow those in LLVM IR. Its intermediate representation is purely functional, and the order of statements matters only if a value is referenced before its definition.

1.3.1 Superoptimizer and Static Analysis

Souper tends to be slow and not consider very much code at a time as its search space increases rapidly with code size. The first part of my research extends Souper to interact with some of the LLVM's dataflow analyses. Souper's interaction with the dataflow facts increases the amount of code that it can effectively interact with. Thus, the optimization power of Souper increases as it exploits LLVM's dataflow facts. As a consequence, Souper discovers more missing optimizations in a given program. The static analysis not only allows the flow of information to reach farther program points, but it is also a pivotal technique in compiler development that enables optimizations. If the dataflow information is unsound, it may lead to miscompilations. On the other side, an imprecise fact can cause missing optimizations and performance degradation. Therefore, it is significant to test static analyses of a compiler. The existing testing approach relies on writing a unit test and performing end-to-end testing. This approach is manual and cannot scale very well for production compilers. We thus need a systematic and automated method for testing static analyses of a compiler.

This dissertation contributes to solving this problem by designing a collection of SMT solver-based algorithms to compute sound and maximally precise dataflow facts. The automated technique works by comparing the facts derived by the compiler with algorithms designed in Souper.

1.3.2 An Automatic Peephole Optimizer for Cranelift

Peephole optimizations are applied on a small sequence of instructions with the goal to simplify it. These optimizations are applied everywhere and are expected to reduce the code size and execution time of the compiled binary. Existing compilers like LLVM have hundreds of rewrite rules implemented in *InstCombine*, *InstSimplification*, and other similar passes to remove local inefficiencies. The size of LLVM's instruction combiner has increased from 2,000 LOC to 36,000 LOC in the past 20 years. Compiler developers have invested their time and effort to manually implement new optimizations and the complexity is continuously growing.

In the spectrum of other commercial compilers, GCC and Go adopt a DSL-based approach to define rewrite rules in a simpler way and use a pattern matcher to generate the optimizations. Unfortunately, many compilers do not adopt this approach to lift their optimizations from the rest of the compiler yet. GCC also specifies only a few of its simplifications in a DSL style, and most of its optimizations are implemented in traditional way as LLVM, MSVC compilers implement their optimizations individually in each pass. The next challenge that remains unsolved is to automatically find the missing optimizations in a given program and express them in an easy to read and easy to use DSL.

Besides the increasing implementation complexity in writing the peephole optimizations, these optimizations are prone to bugs. It is hard to reason about the correctness of these optimizations as there are so many corner-cases involved. A random testing tool, Csmith [4], and a verification tool, Alive [3], identified bugs in the peephole pass of LLVM. It is important to get these optimizations implemented correctly as unsound optimizations can lead to miscompilation. This motivates my goal of verifying the optimizations for soundness while we discover the missing ones.

Cranelift, a retargetable code generator integrated in Wasmtime [2], a runtime system for WebAssembly, did not implement a peephole optimization pass until December 2020. peepmatic, a DSL-based automatic peephole optimization generator got integrated into Cranelift with the goal of specifying the transformations easily. It implemented an automaton-based matcher to generate the optimization pass for Cranelift. However, this compiler does not support verifying the correctness of the rewrite rules and requires developers and users to specify transformations instead of automatically generating them and verifying their correctness.

This dissertation uses formal-methods-based tools to fulfill these requirements. We use a synthesizing superoptimizer, Souper, that uses an SMT solver to find missing optimizations and generating them in Souper's IR. The details of the IR are discussed in Chapter 2. The optimizations synthesized by Souper are verified for correctness as well. This dissertation presents a fast prefix-tree pattern matcher to automatically implement the peephole optimizer for a WebAssembly JIT compiler, Cranelift.

1.4 Contributions and Outline

This dissertation makes the following contributions:

(1) collection of novel solver-based algorithms for computing maximally precise dataflow analysis results, demonstrated in Chapter 4.

(2) the solver-based algorithms have found numerous imprecisions in LLVM's static analyses, many of which have been fixed, presented in Section 4.3 of Chapter 4.

(3) improving superoptimization power of Souper using LLVM's dataflow analyses to find more missing optimizations in a program, discussed in Chapter 3.

(4) an automatic peephole optimizer based on superoptimization technique for the Cranelift compiler, illustrated in Chapter 5.

1.5 References

[1] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan, *Synthesis of loop-free programs*, in Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '11, 2011, pp. 62–73.

[2] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, *Bringing the web up to speed with webassembly*, in Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2017, pp. 185–200.

[3] N. P. Lopes, D. Menendez, S. Nagarakatte, and J. Regehr, *Provably correct peephole optimizations with Alive*, in Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2015, pp. 22–32.

[4] X. Yang, Y. Chen, E. Eide, and J. Regehr, *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.

CHAPTER 2

BACKGROUND

This chapter gives an overview of the tools, compiler frameworks, and techniques that build the basic background for the rest of this dissertation. Section 2.1 provides an overview of LLVM, its goal, and its uses. The overall design and intermediate representations of LLVM, Cranelift, and Souper are discussed in Sections 2.1, 2.2, and 2.3, respectively.

2.1 LLVM

The LLVM compiler infrastructure is a collection of modular and reusable compiler and toolchain technologies. The LLVM project is open source and supports a large number of modular libraries. The C/C++ frontend, Clang, makes use of the LLVM libraries for supporting its backend. The uniqueness of this compiler is that it makes it so easy to extend the compiler toolchain to support a new programming language, or new backend.

2.1.1 Design

LLVM adopts the classical three-phase compiler design with a front, middle, and back end. It makes the best use of this design by supporting multiple source languages in the frontend, a common intermediate representation in the middle-end, and multiple target architectures in the backend, shown in Figure 2.1. The intermediate representation in the middle-end is called the LLVM IR. LLVM was initially designed for C and C++. However, it now supports a broader set of languages like Ada, C#, Objective-C, CUDA, Haskell, Ruby, Java bytecode, Rust, Swift, Scala, and many more. Programming languages like Rust and Swift have their frontends built upon LLVM, and they depend on LLVM's optimizer and backend for further functionality. The most commonly used frontend in LLVM is *Clang* that supports C, Objective-C, and C++. The middle-end performs various transformations on the LLVM IR and generates optimized LLVM IR. The backend is also easily extended for a new target architecture as it is only responsible for compiling the LLVM IR to target machine code.

2.1.2 LLVM IR

The LLVM code representation can be used in three different functionally equivalent forms: a human-readable assembly language, an in-memory internal representation, and bitcode format [2]. It is the only common intermediate representation between several frontends and backends, unlike various intermediate representations in GCC. GCC lowers a given program written in C programming language to GIMPLE, a three-address representation, and further compiles it to a register transfer language (RTL) representation, and finally to a machine description to emit the assembly code.

The LLVM IR is a strongly typed assembly language which abstracts away details of the target. For instance, the IR supports an unlimited number of virtual or temporary registers to store a value.

A program consists of modules and each module has one or more functions. Each function is divided into basic blocks. Each basic block has a single entry and an exit point. The basic block comprises instructions, and an instruction specifies an opcode and a list of operands to operate on. The instructions are explicitly typed and may produce a single or an aggregate value. The values returned by an instruction are stored in temporaries and can be used in the following instructions. LLVM IR is in static single assignment (SSA) form, which means each virtual register can be assigned a value only once and should be defined before use. If LLVM IR did not have a SSA form, it would require implementing a static analysis to trace the def and use chains to perform any optimizations. The SSA form simplifies the optimization transformations to be performed. For control flow, LLVM IR has a branch, phi, switch, return, invoke instructions. The branch instruction occurs at the end of a basic block that diverges its control flow to a true and false path. The phi instruction occurs at the start of a basic block, where multiple paths of incoming values converge.

Consider a C function and its equivalent LLVM IR in Figure 2.2. The function declaration closely follows a C style syntax. The function *foo* takes two arguments of $i32$ type and return an $i32$ value. Identifiers start with a *%*. The entry block labeled *entry* compares (*icmp eq*) the value of $\%a$ with zero and returns an $i1$ value. The branch instruction, *br*, takes a single $i1$ value i.e., a conditionm and two target labels. If the condition is true, it branches to a basic block with the first label and otherwise to the second label. The branch instruction in this example is conditional. Another important instruction is *phi*, which represents a phi node in SSA form. The first argument to a phi instruction specifies the type of incoming values. The remaining list denotes the pairs of arguments. The first value in the pair indicates the incoming value, and the second part is the label of a predecessor block from which the value is incoming.

2.2 Cranelift

Cranelift is a compiler framework implemented in Rust that translates a target independent IR to a machine code executable. It is designed to be used in just-in-time compilation and as a code generator for the runtime system, Wasmtime. My research focuses on using it as a standalone compiler or a code generator that compiles WebAssembly to X86_64 code. Cranelift's use-case has recently been extended towards replacing the LLVM backend of Rustc compiler for debug build mode.

Cranelift is known for its fast code generation as it generates the machine code that runs at a reasonable speed. Unlike LLVM, it does not implement a lot of expensive optimizations and rather sticks with light-weight optimizations. The compilation model in Cranelift thus compiles each function separately, allowing concurrent compilation leading to an overall fast compilation. In LLVM, the largest entity, module, is a collection of functions and global variables that may reference the external symbols as well.

2.2.1 Design

Crabelift's frontend compiles WebAssembly programs to Cranelift IR, which are further optimized by light-weight optimization passes to produce optimized Cranelift IR, and finally lowering and register allocation is done to generate machine code for the X86 or AArch64 architecture. The existing Cranelift compiler has optimization passes like, *preopt* that involve a few early-stage optimizations which look for patterns involving divide or remainder by a constant. The *postopt* is a post-legalization rewriting pass. It optimizes branch or compare instructions to use flags. A dead code elimination pass removes instructions that have no side-effects, and whose result values are never used. The detailed design of Cranelift compiler framework is shown in Figure 2.3.

2.2.2 Cranelift IR

Compilers are usually implemented as a sequence of passes. These passes communicate with each other by transferring the facts they learn about a program. Compilers need a representation to compactly and correctly represent the fact at each step and it is called an intermediate representation (IR). Compilers may have a single or multiple IRs. For instance, GCC that has GIMPLE, RTL, other low-level IRs to represent the information as GCC passes transitions to emit the final machine code. LLVM has a primary LLVM IR, but it also has graph based IRs to represent the code that is used by an instruction selector. It has a linear IR to represent ISA-specific information in the instruction scheduling and register allocation phases of code generation. Another difference in LLVM IR as compared to the Cranelift IR is that Cranelift's IR has more low-level abstraction that can be used throughout the code generation phase.

Cranelift has two different kinds of intermediate representation:

- a high-level text format based IR called **CLIF** (Cranelift IR Format),
- a low-level in-memory data structure based IR called **VCode** (Virtual-registerized Code).

My research work on Cranelift's peephole optimizations mainly focuses on the CLIF IR. This IR represents all kinds of instructions' operations, the types supported by them, and the operands in a specific order as the semantics of each operation are defined. It is in Static Single Assignment (SSA) form which means that each variable is defined exactly once, and every variable must be defined before it is used. CLIF IR represents phi-nodes as the block parameter instead of defining a phi instruction at the start of basic block. For instance, optimization passes like constant folding, dead code elimination in Cranelift operate on the CLIF IR. We discussed earlier that Cranelift compiles each function individually. But this fact does not mean that the .clif file cannot have multiple functions. The only restriction is that functions cannot share any data or reference each other directly. An example of manually written .clif file following a peephole optimization found by Souper is shown in Figure 2.4.

However, there is obviously a way to generate CLIF IR from WASM using the tool 'clif-util' with subcommand 'wasm'. We can also use the same utility to verify the CLIF IR.

The instructions in SSA form (target-independent) define zero, one, or more result values that must be mapped later to emit the target code correctly.

The type-system in Cranelift is less abstract and closely follows the types supported by the ISA registers. There are numerous types of SSA values but the major ones that my research work deals with are:

- **Boolean Types** with 1, 8, 16, 32, 64-bits and these are denoted by B1, B8, B16, B32, and B64 respectively. All of them encode 'true' or 'false'. For larger bit-widths, the bits are redundant.

- **Integer Types** such as I8, I16, I32, and I64. These can be interpreted as both signed and unsigned integer values depending on the type of instruction and the type of operand the instruction expects. My work mainly uses I32 and I64 for bitwise and arithmetic operations.

- **Immediate Operand Type** is a not a SSA value, rather it is a constant. Some arithmetic and bitwise instructions that apply operations on a pair of SSA value and an actual constant or literal, they expect their operand to be an immediate value. The two common types that my work has used are: Imm64 for a 64-bit signed integer and Uimm64 for a 64-bit unsigned integer operand.

The dataflow graph (DFG) defines all instruction, basic blocks and values in a function. It tracks the dataflow dependencies among them and also checks if the values are block parameters or result of an instruction.

The CLIF IR is lowered to build a VCode IR. The main purpose of the lowering is to gather more target-dependent information in the VCode IR from the target-independent facts in CLIF IR. The VCode is no longer in the SSA form and backend phases like, instruction selection, register allocation, and so on start working on this intermediate representation to finally emit the machine code.

2.2.3 Undefined Behavior

Cranelift models undefined behavior in different way than LLVM. Arithmetic instructions in Cranelift either produce a value or a trap. The trap instruction terminates execution unconditionally. As an example, if the shift operand (op[0]) is shifted left or right by an amount that is equal or greater than the number of bits of a shift operand (op[0]), LLVM returns a poison value [3]. Cranelift, however, avoids undefined behavior by masking the shift amount (i.e., op[1]) of shift instruction with a value one less than the number of bits of shift operand (op[0]). The idea is to set all high-order bits to zero that would have caused an overflow or undefined behavior otherwise. In other words, the shift amount is wrapped around when it reaches the maximum value of number of bits in an operand. The semantics of shift-left instruction in Cranelift are:

```
Result = ShlExpr::create(op[0], AndExpr::create(op[1], Width - 1))
```

For example, if a 32-bit operand %0 is shifted left by a constant 33 (value larger than the width), LLVM returns a poison value.

```
%1:i32 = shl %0, 33:i32
```

For the same example, let's see how Cranelift avoids returning a poison value.

```
// Cranelift masks the shift amount 33 with Width - 1
// 100001 && 011111 = 000001

%1:i32 = and 33:i32, 32:i32

// Cranelift shifts the operand %0 by %1 (result from mask above)
// %2 = %0 << %1

%2:i32 = shl %0, %1
```

Cranelift masks the shift amount (if it is larger than the width of an operand to be shifted) with a constant value one less than the width. In this case, it applies the mask on constants 33 and 31 that returns a constant value 1 in %1. It later shifts the operand %0 by mask result, %1.

2.3 Souper

Peephole optimizations work on small portions of code to improve code size and performance. Peephole optimizations in the LLVM compiler have a significant implementation complexity. as the size of it's major pass, *InstCombine*, an instruction combiner, has increased from 2,000 lines of C++ code to 36,000 LOC. These optimizations were primarily written for C/C++, but as LLVM continued to extend its support for other programming languages, it became harder for compiler developers to recognize missing transformations and hence, some important optimizations were not implemented in the compiler. Compiler developers spend years writing these optimizations and this traditional approach of writing an optimizer for a compiler is turning out to be very effort-intensive.

An alternative approach, superoptimization, is an effort to automatically discover the missing peephole optimizations in a given program using a search procedure, a cost function, and an equivalence checker. Souper, an open-source enumerative synthesis-based superoptimizer for LLVM IR was developed to find missing peepholes and verify them for correctness. Souper has two intended use cases. First, it can be used in an offline mode to make suggestions for missing optimizations to compiler developers. Production compilers such as LLVM, Microsoft Visual C++ Compiler, Mono [1], Binaryen [5] have used Souper for the this purpose. Souper IR is simple to read, understand and has so much similarity with different IRs in compilers that it has made it easier for other compiler developers to use Souper. Second, it can be used as an optimization pass in LLVM in an online mode to perform optimizations discovered by Souper. When building LLVM+Clang-5.0 using Souper as an optimization pass in LLVM, it makes the Clang binary 1.6 MB smaller.

2.3.1 Design

Souper's basic abstraction is a directed acyclic dataflow graph. The harvester extracts every integer SSA value from a function in the LLVM IR to Souper IR using backward slicing along dataflow edges to construct the candidates. These harvested candidates are called left-hand sides (LHSes). Extraction of a single LHS is terminated by a load from memory, a store to memory, a loop back edge, a value coming from a different function, or any other value type that is not supported by Souper.

For each harvested left-hand side, Souper uses an enumerative synthesis algorithm that takes a cost function to synthesize a refinement, called right-hand side (RHS). The goal of synthesis is to solve an exists-for-all query to find the RHS and Souper uses an SMT solver like, Z3 to solve the queries. The query searches for a solution, $rhs(x)$, such that $lhs(x) = rhs(x)$ and $cost(rhs(x) < cost(lhs(x))$, for all input values of x. Enumerative synthesis approach computes all solutions with a cost that is less than the cost of the given left-hand side.

When Souper successfully finds the solution, i.e., RHS with a minimum cost, it transforms the RHS in Souper IR to LLVM IR. The mapping for a pair of each left-hand side and right-hand side is stored in a cache to save the time taken by synthesis in case there is a repeated LHS.

The overall design is illustrated in Figure 2.5. Souper's extension with dataflow analyses to improve its capability to find more missing optimizations is discussed in Chapter 3. The latest extension in Souper by Mukherjee et al. [4] improves the runtime of an enumerative synthesis approach by pruning the search space of harvested candidates, using dataflow analysis.

2.3.2 Souper IR

The Souper intermediate representation has two primary forms: an in-memory data structure, *(Inst)*, which is generated by an instruction harvester, and it is the core data structure on which the synthesis engine works; and a text format which can be manually written by the users and is parsed to build an *Inst* data structure. The textual format is aligned closely to the LLVM IR in order to make it accessible to LLVM developers. The IR is purely functional; the definition of a variable dominates every use, and each variable is assigned exactly once. The operations also closely follow those in the LLVM IR. The operands can either be a constant or a value. Souper currently supports 59 instructions that are a subset of scalar integer set of instructions of LLVM. The different types of instructions and value types are:

1. **Var**: In LLVM IR, the input parameters are passed as an argument to a function. Souper specifies the input arguments with a *var* that must have a width and a label to which it is assigned. For example, $\%0 : i32 = var$ is an unconstrained value with 32-bits and can be used later by any other instructions by referencing its label %0.

2. **Bitwise and Arithmetic Instructions**: Souper supports several instructions in this category and follows the semantics of corresponding instruction from LLVM IR.

   (a) Basic Binary Operations like Bitwise-And (and), Bitwise-Or (or), Bitwise-Xor

(xor), Addition (add), Subtraction (sub), Multiplication (mul), Division (div), Shift-left (shl), Logical Shift-right (lshr), Arithmetic Sight-right (ashr) have straightforward semantics. They require two operands of same type, either a value or a constant. The result value has the same type as its operand.

(b) Some of the basic arithmetic operations also allow no signed wrap, or no unsigned wrap, or both. Instructions such as: addnsw, addnuw, addnw, shlnsw, shlnuw, shlnw, and so on trigger UB if the result of the operation overflows and it tries to wrap around the result.

3. **Comparison Instructions**: The operands of compare instruction must have same number of bits and the instruction returns a 1-bit result. Different kinds of compare instructions supported in Souper are:

   (a) Equal checks if the operands are equal. The syntax is as:

   $\%1 : i1 = eq\ \%var1, \%var2.$

   (b) Not equal checks if the operands are not equal. The syntax is as:

   $\%1 : i1 = ne\ \%var1, \%var2.$

   (c) Unsigned less than interprets the operands as unsigned and checks if first operand, %var1, is less than second operand, %var2. The syntax is:

   $\%1 : i1 = ult\ \%var1, \%var2.$

   (d) Signed less than interprets the operands as signed and checks if first operand, %var1, is less than second operand, %var2. The syntax is as:

   $\%1 : i1 = slt\ \%var1, \%var2.$

   (e) Unsigned less than or equal to interprets the operands as unsigned and checks if first operand, %var1, is less than or equal to second operand, %var2. The syntax is:

   $\%1 : i1 = ule\ \%var1, \%var2.$

   (f) Signed less than or equal to interprets the operands as signed and checks if first operand, %var1, is less than or equal to second operand, %var2. The syntax is:

   $\%1 : i1 = sle\ \%var1, \%var2.$

4. **Conversion Instructions**: The instructions in this category take one operand and perform bit conversion on it.

   (a) Truncate instruction, $\%1 : i32 = trunc\ \%var : i64$, truncates a 64-bit operand $\%var$ to 32-bit. The only constraint for this operation is that the bit size of input operand must be larger than the destination type.

   (b) Zero-extend instruction, $\%1 : i32 = zext\ \%var : i8$, fills zero in the high order bits of the operand $\%var$ until it reaches the width of the result %1. In the example, *zext* fills 24 high order bits with zero.

   (c) sign-extend instruction, $\%1 : i32 = sext\ \%var : i8$, copies sign bit to the high order bits of the operand $\%var$ until it reaches the width of the result %1.

   Souper does not have any specific instructions for greater than or greater than equal to operations because greater than operations are canonicalized to less than operations in LLVM. Hence, all such comparisons are expressed using less than operation.

5. **Intrinsics**: Souper supports four bit manipulation intrinsics from LLVM that allow efficient code generation.

   (a) Popcount counts the number of set bits in a given value. $\%0 : i32 = ctpop\ \%var : i32$ in Souper is implemented by iterating on each bit and counting if the value is set to one.

   (b) Byteswap swaps the low and high byte of the input operand. It operates on an operand with even number of bytes starting at 16-bits. $\%1 : i32 = bswap\ \%var : i32$ swaps the first byte with last byte, and second with a third byte of operand %var. This intrinsic in Souper is implemented by extracting the bytes and concatenating them in an order.

   (c) Count leading zeros (ctlz) and count trailing zeros (cttz) is also implemented by iterating on each bit of the input operand. The syntax of the intrinsics are: $\%1 : i32 = ctlz\ \%var : i32$ and $\%1 : i32 = cttz\ \%var : i32$.

   Souper also supports arithmetic with overflow intrinsics from LLVM that allow fast checking of overflow result. The intrinsics in this category return an aggregate type structure in LLVM that has more than one result values. However, Souper represents the aggregated result in a larger bit-width value. For example, if the result of add operation on two 32-bit operands overflow, then the result of *add.with.overflow* is 33-bits wide that includes both 32-bit result of an add operation, and 1-bit to represent an overflow bit. The six different overflow intrinsics supported by Souper with it's syntax are as:

(a) Signed add with overflow:

$\%1 : i33 = sadd.with.overflow\ \%var1 : i32, \%var2 : i32.$

(b) Unsigned add with overflow:

$\%1 : i33 = uadd.with.overflow\ \%var1 : i32, \%var2 : i32.$

(c) Signed subtract with overflow:

$\%1 : i33 = ssub.with.overflow\ \%var1 : i32, \%var2 : i32.$

(d) Unsigned subtract with overflow:

$\%1 : i33 = usub.with.overflow\ \%var1 : i32, \%var2 : i32.$

(e) Signed multiply with overflow:

$\%1 : i33 = smul.with.overflow\ \%var1 : i32, \%var2 : i32.$

(f) Unsigned multiply with overflow:

$\%1 : i33 = umul.with.overflow\ \%var1 : i32, \%var2 : i32.$

6. **Extract Instruction**: This instruction extracts the result value from an operand that contains multiple results all aggregated into one value with a larger number of bits.

7. **Select Instruction**: The select instruction requires three operands; first operand is a 1-bit condition, the other two operands must be of the same type. This instruction is used to choose one value between two operands based on a condition without branching.

In $\%1 : i32 = select\ \%cond : i1, \%var1 : i32, \%var2 : i32$, if the condition %cond is true, the result value of select instruction is assigned the value %var1, otherwise %var2.

8. **Control flow Instructions**: Souper does not have control flow, but it provides two ways to represent dataflow facts learned from control flow of a program being optimized. These constructs are known as path conditions (to represent diverging control flow) and block path conditions (to represent the converging control flow). The path condition specifies that the relation $var1 == var2$ must hold true in the instructions following after $pc\%var1 : i1, \%var2 : i1$ in a program.

   The second construct, *blockpc*, represents that the relation $\%var1 == \$var2$ must hold true on the ith incoming path of the converging block.

9. **Phi and Block Instruction**: The convergent control flow is represented using a phi node and a block. The phi instruction, $\%1 : i32 = phi\ \%b, \%var1 : i32, \%var2 : i32$, requires at least three operands; the first specifies the block name %b, and the remaining operands specify the values to be assigned to the result value %1. The number of remaining operands is equal to the number of paths incoming to the converging block. The type of these operands must match the type of result, %1.

10. **Infer, Result, and Candidate Instructions**: The *infer* instruction, $infer\%x$, indicates the root node of a program that Souper is being asked to optimize the computation of. This is specified at the end of left-hand side of an optimization in Souper IR. Souper instructions return a single-value result. On synthesizing a better way to write a given left-hand side, result instruction, $result\%var1$, specifies the root node of synthesized right-hand side of an optimization. The *cand* instruction, $cand\%X\%Y$, simply specifies that X can be rewritten as Y. This is used to verify an optimization (including both left and right side) in Souper.

2.4 References

[1] *Mono c# compiler.* http://mono-project.com/docs/about-mono/languages/csharp/.

[2] C. Lattner and V. Adve, *LLVM: A compilation framework for lifelong program analysis & transformation*, in Proceedings of the International Symposium on Code

Generation and Optimization: Feedback-directed and Runtime Optimization, CGO '04, 2004, pp. 75–88.

[3] J. Lee, Y. Kim, Y. Song, C.-K. Hur, S. Das, D. Majnemer, J. Regehr, and N. P. Lopes, *Taming undefined behavior in llvm*, ACM SIGPLAN Notices, 52 (2017), pp. 633–647.

[4] M. Mukherjee, P. Kant, Z. Liu, and J. Regehr, *Dataflow-based pruning for speeding up superoptimization*, Proceedings of the ACM on Programming Languages, 4 (2020), pp. 1–24.

[5] A. Zakai, *Binaryen: Compiler infrastructure and toolchain library for webassembly*, 2015. http://github.com/WebAssembly/binarye.

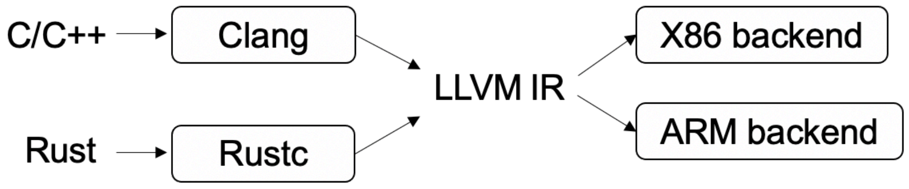


Figure 2.1. Design of LLVM.

```
int foo(int a, int b) {
  if (a == 0)
    a = a + 1;
  else if (a > 0)
    a = a + 2;
  else
    a = a - 1;
  return a + b;
}
```

(a)

```
define i32 @foo(i32 %a, i32 %b) {
entry:
  %cmp = icmp eq i32 %a, 0
  br i1 %cmp, label %if.true, label %else.if

else.if:          ; preds = %entry
  %cmp1 = icmp sgt i32 %a, 0
  br i1 %cmp1, label %else.if.true, label %else

else.if.true:     ; preds = %else.if
  %add2 = add nsw i32 %a, 2
  br label %if.true

else:             ; preds = %else.if
  %sub1 = add nsw i32 %a, -1
  br label %if.true

if.true:          ; preds = %else, %else.if.true, %entry
  %a.addr.0 = phi i32 [ %add2, %else.if.true ], \
                      [ %sub1, %else ], \
                      [ 1, %entry ]
  %add6 = add nsw i32 %a.addr.0, %b
  ret i32 %add6
}
```

(b)

Figure 2.2. A function written in C and compiled to LLVM IR

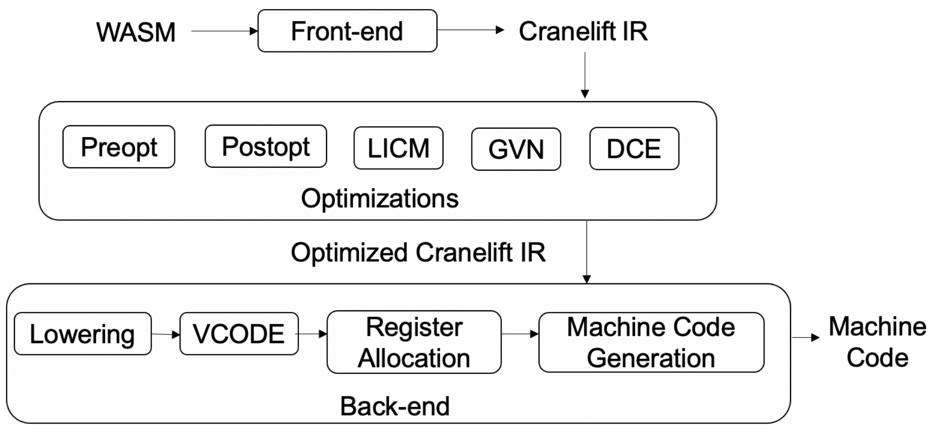


Figure 2.3. Design of Cranelift.

```
%0:i32 = var                ; a 32-bit input variable %0
%1:i32 = add %0, -1:i32     ; %1 = %0 + (-1)
%2:i32 = sub 0:i32, %1      ; %2 = 0 - %1
infer %2                    ; return %2
%3:i32 = sub 1:i32, %0      ; %3 = 1 - %0
result %3                   ; return %3
```

(a) An example peephole optimization synthesized by Souper in Souper IR.

```
function %souper1(i32) -> i32 {
block0(v0: i32):
    v1 = iadd_imm v0, -1   ; v0 + (-1)
    v2 = irsub_imm v1, 0   ; 0 - v1
    return v2
}
```

(b) The LHS of a same peephole optimization in CLIF IR.

```
function %souper1(i32) -> i32 {
block0(v0: i32):
    v1 = iadd_imm v0, -1   ; v0 + (-1)
    v2 = irsub_imm v0, 1   ; 1 - v0
    return v2
}
```

(c) The optimized CLIF IR generated by a peephole pass that replaces v2 in (b) by v2 in (c).

Figure 2.4. An example peephole optimization in both Souper and CLIF IR.

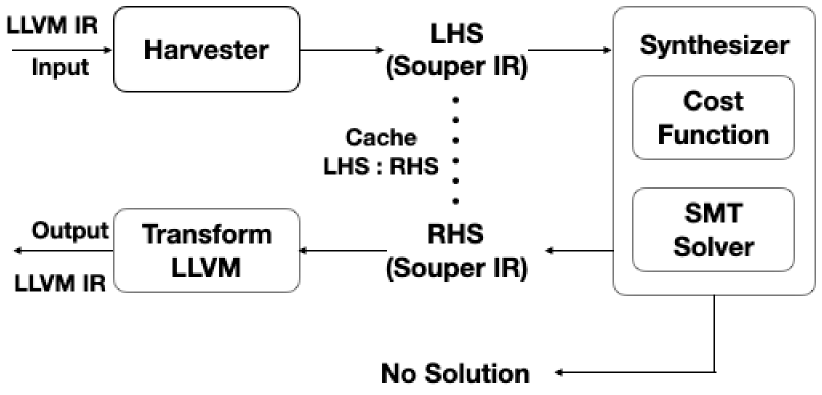


Figure 2.5. Souper: An enumerative synthesis-based superoptimizer for LLVM IR.

CHAPTER 3

DATAFLOW ANALYSIS

This chapter presents an in-depth introduction to the dataflow analyses. It emphasizes on why these analyses are required to perform compiler optimizations in Section 3.1. While defining any dataflow analysis, it is important to understand the requirements because they guarantee correctness (safety) and precision. The detailed requirements are discussed in Section 3.2. This chapter continues to formally define all the dataflow analyses in Section 3.5 and define its representation in Souper in Section 3.6. Overall, this chapter sets a background for testing of dataflow analyses work that we will study in detail in the next chapter.

3.1 Motivating Example

We will start with a motivating example to understand a simple compiler optimization. In this example, the compiler recognizes that the first assignment to a at point $P1$ is useless because a is defined here but never used later. Hence, the compiler can optimize these two lines of code by deleting the statement $P1$.

```
P1: a = b + c
P2: a = 3
```

In another example as shown below, the compiler observes that the expression $a * b$ is computed again at point $P2$, and thus extracts the redundant or common multiplication expression and later uses it. With this observation, compiler can optimize the code fragment as follows.

```
                                   t1 = a * b;
P1: x = a * b + 1;        ⇒        x = t1 + 1;
P2: y = a * b;                     y = t1;
```

Compiler must locate the points in a given program where changing the code can lead to improvement. To gather the information and to make such observations, the compiler performs an analysis at compile-time to make predictions about the runtime flow of values. This is called **dataflow analysis**.

Every optimization operates in a particular region that defines the scope of an optimization [3]. The scope can be limited to a basic block, i.e., branch-free code fragment; different basic blocks to include branches or cycles in a control flow graph (CFG); intraprocedural (within a function); or interprocedural (across multiple functions). In the first motivating example, we need a dataflow analysis that can gather information about variables `a,b,c` in the context of their use and definitions at each line of code to track the definitions that become useless, or which variables are not used at all and thus are marked as dead code.

For a complex example that involves a branch condition or a loop, the dataflow analysis problem becomes more complicated as we need to identify how to combine information coming from different paths (i.e., from multiple predecessors or successors) and how many times we need to iterate to compute the dataflow facts for a control flow graph (CFG) that involves cycles. Dataflow analysis or static analysis is used to perform code optimizations as we discussed in this section earlier. However, this technique is also useful in identifying the buffer overflow, null pointer dereference, divide by zero kind of errors. Thus, it is used in verification and validation of programs to ensure code safety.

3.2 Requirements of Dataflow Analysis

Dataflow analysis makes a prediction for the runtime flow of values at compile time. The prediction made statically must satisfy the following requirements.

- **Sound:** Static analysis is sound if properties or facts it can prove are indeed true in all executions, and there are no false-positives. If the analysis computes an unsound fact that drives an optimization, it leads to a miscompilation. For example, an unsound analysis might result in eliminating the bounds check that could fail.

- **Precise:** The termination problem does not always allow dataflow analysis to see if a property or fact holds and this causes imprecision. We rather want the analysis to work well in practice so that it can compute precise dataflow facts. In an urge to not spend too much time reasoning about the dataflow information, production compilers choose a maximum iteration factor for static analysis and thus result in imprecise dataflow facts sometimes. In such cases, the lack of precision leads to missed optimizations. There is another possibility of imprecision where a dataflow analysis gathers the information from all predecessors or successors at any program point, even though any of the incoming paths is not reachable. In this situation, the extra information from unreachable paths sometimes results in loss of precision. For example, an imprecise interval analysis over approximates the range to a larger set of values than the original smaller set of values. It may result in missing an opportunity to optimize an if-condition that checks the upper bound of the value of a variable. However, the goal of over-approximation is to ensure that the information computed is correct.

- **Fast:** Reaching a fixpoint can be very fast or very slow, depending on the structure of the abstract domain, the structure of the program being analyzed, the number of iterations, order of traversal of nodes in a control flow graph, and whether the analysis is flow, path, or context sensitive. The important aspect of implementing a static analysis in a production compiler is that it should not spend too much time computing the dataflow information leading to an increase in compile time.

3.3 Properties of Dataflow Analyses

A Dataflow Analysis provides facts/information about the runtime flow of data in a program being analyzed at the compile time. program. The following list of properties classify the dataflow analysis into a different category.

- **Scope:** A dataflow analysis may be performed at different scopes in a program. For instance, when an analysis is performed at a sequence of instructions restricted to a single basic block that does not involve any control flow transfer, it is called Local Dataflow Analysis. When an analysis is performed at a basic block level across different basic blocks in a given function or program that involves control flow transfer, it is called Global or Intraprocedural Dataflow Analysis. The last category is when analysis is performed at a function level across different functions or procedures, it is called Interprocedural Dataflow Analysis.

- **Direction:** To reach a fixpoint in an iterative dataflow analysis, it is important to identify how the flow of data propagates in a given function or program. In forward dataflow analysis, the information at a program point summarizes what can happen on paths from "entry" to that program point. In backward analysis, the information at a program point summarizes what can happen on a path starting from the current program point to the exit.

- **Sensitivity of Analysis:** To perform static analysis, a collection of abstract operations executes repeatedly on an initial program state, terminating when a fixpoint is reached.

  A **flow sensitive** analysis respects control-flow relationships within functions of the program being analyzed; a **path sensitive** analysis avoids imprecisions that would otherwise occur when control flows merge together after branches; and, a **context sensitive** analysis respects the fact that functions behave differently when reached via different call chains.

  All of the analyses that we consider in this chapter are flow sensitive but path and context insensitive. The information is computed at each instruction in the program.

3.4 Dataflow Analysis in LLVM

Dataflow analyses in LLVM are implemented using an iterative algorithm. These analyses look at N instructions deep, backward or forward to compute the dataflow facts. They can guarantee more precision if they iterate in an unbounded iteration space or if they can look at any depth of instructions without limit. However, the low efficiency leads to the high compilation time as this information is computed at the compile time. To make a compromise, the compiler developers generally set the bound on the number of iterations. The important question to address is that the information computed is safe and sound, even though precision is affected to some extent. Chapter 4 discusses these aspects in detail.

All these analyses are implemented at the intermediate representation level. LLVM's intermediate representation (IR) is a good substrate for many static analyses. First, rules that are tricky and implicit at the programming-language level, such as type conversions and ordering of side-effects in expressions, are made explicit. Second, the SSA [2] form makes it easy and efficient (in the absence of pointers, at least) to follow the flow of data through a function. Third, since LLVM contains competent implementations of a variety of IR-level optimizations such as constant propagation, dead code elimination, function inlining, arithmetic simplifications, and global value numbering, many small obstacles to static analysis are optimized away before they need to be dealt with. LLVM supports many different static analyses; the ones that we have focused on are extensively used by the LLVM's optimizations, instruction simplification, instruction combine, and other transformation passes. These analyses are not only used in middle-end optimizations, but also in code generation.

3.5 Formal Definitions

Programs are made of concrete operations such as addition that operate on concrete values. Dataflow analyses employ abstract operations that operate on abstract values; each abstract value represents a set of concrete values. For example, the integer range $[6..10]$ represents the set $\{6, 7, 8, 9, 10\}$. A *concretization function* γ maps an abstract value to the set of concrete values it represents, and an *abstraction function* α maps any set of concrete values to the most precise abstract value whose concretization is a superset of the given set. For any abstract value e, it should always be the case that $\alpha(\gamma(e)) = e$. However, it is not the case that, for an arbitrary set of concrete values S, $\gamma(\alpha(S)) = S$. For example, in the abstract domain of integer ranges, $\gamma(\alpha(\{5, 8\})) = \{5, 6, 7, 8\}$. Abstract values form a lattice or semilattice, with the lattice order corresponding to the subset relation among the concretization sets of the values in the lattice.

Let $\hat{f}$ be an abstract operation, such as addition over integer ranges. If $\hat{f}([6..10], [1..2]) = [7..12]$, then in this case $\hat{f}$ is returning the most precise possible result. It is always possible to create a sound and maximally precise abstract operation by concretizing its arguments, applying the corresponding concrete operation to the cross product of the concretization sets, and then applying the abstraction function to the set containing the resulting concrete values. Obviously this implementation is too slow to use in practice unless concretization sets are small. The challenge in creating good abstract operations is to make them sound and as precise as possible, while making them fast enough so that static analyses built upon them do not slow down the overall compilation too much. Addition for integer ranges is an easy case: the lower bound of the output interval is just the sum of the lower bounds of the input intervals, etc. Other cases are much more difficult.

3.5.1 Known Bits

This analysis determines which bits of a value are zero or one in all executions. Known bits are used in the preconditions of peephole optimizations. For example, a signed division-by-constant can be implemented more efficiently if the dividend provably has a zero in its high-order bit. These precondition checks are ubiquitous in LLVM's instruction combiner, a 36,000 line collection of peephole-like optimizations.

Let f be a function composed of LLVM instructions, W be the width in bits of f's output, and $\mathbb{K}$ be the result of the known bits analysis for the output of f. Let $f(x)_i$ and $\mathbb{K}_i$ represent the values of f and the analysis result at bit position i. Each bit of the analysis result is known to be zero, known to be one, or else unknown.

- The known bits analysis is sound if:

$$\begin{aligned}\forall i \in 0 \ldots W-1,\ \forall x : \mathbb{K}_i = 0 &\implies f(x)_i = 0 \land \\ \mathbb{K}_i = 1 &\implies f(x)_i = 1\end{aligned}$$

- The known bits analysis is maximally precise if it always concludes that a bit is known when it is sound to do so:

$$\begin{aligned}\forall i \in 0 \ldots W-1,\ (\forall x : f(x)_i = 0) &\implies \mathbb{K}_i = 0 \land \\ (\forall x : f(x)_i = 1) &\implies \mathbb{K}_i = 1\end{aligned}$$

3.5.2 Number of Sign Bits

The number of sign bits is the number of high-order bits that provably all hold the same value. Every value trivially has at least one sign bit, and values can accrue additional sign bits by, for example, being sign extended or arithmetically right shifted. The sign bits analysis can be used to reduce the number of bits allocated to a variable.

Let $\mathbb{S}$ be the result of the number of sign bits analysis for the output of f, an input function composed of LLVM instructions as defined in the context of previous analysis. Let $f(x)_i$ represent the values of f at bit position i.

- The sign bits analysis is sound if:

$$\forall x,\ \mathbb{S} = n \implies (\forall i \in 1 \ldots n-1 :\ f(x)_i = f(x)_{i-1}),\ 1 \le n \le W$$

- The sign bits analysis is maximally precise if:

$$\forall x,\ \forall i \in 1 \ldots n-1 :\ f(x)_i = f(x)_{i-1} \implies \mathbb{S} = n,\ 1 \le n \le W$$

3.5.3 Single-bit Analyses

LLVM provides a number of analyses that provide a single bit of information; these are:

- Value is provably non-zero: Let $\mathbb{Z}$ be the result of the non-zero analysis for the output of function f defined in previous analyses. Let $f(x)$ represent the values of f for all inputs x. The non-zero analysis is sound if:
$$\forall x, \mathbb{Z} = 1 \implies f(x) \neq 0 \land$$
$$\mathbb{Z} = 0 \implies f(x) = 0$$
The non-zero analysis is precise if:
$$\forall x, f(x) \neq 0 \implies \mathbb{Z} = 1 \land$$
$$f(x) = 0 \implies \mathbb{Z} = 0$$

- Value is provably negative: Let $\mathbb{N}$ be the result of the negative analysis for the output of function f defined in previous analyses. Let $f(x)_0$ represent the values of f for all inputs x at bit position 0 , which is the most significant bit (MSB). $\mathbb{N} = 1$ represents that the value is provably negative, and vice-versa. The negative analysis is sound if:
$$\forall x, \mathbb{N} = 1 \implies f(x)_0 = 1 \land$$
$$\mathbb{N} = 0 \implies f(x)_0 = 0$$
The negative analysis is precise if:
$$\forall x, f(x)_0 = 1 \implies \mathbb{N} = 1 \land$$
$$f(x)_0 = 0 \implies \mathbb{N} = 0$$

- Value is provably non-negative: Let $\mathfrak{N}$ be the result of the non-negative analysis for the output of function f defined in previous analyses. Let $f(x)_0$ represent the values of f for all inputs x at bit position 0 , which is the most significant bit (MSB). $\mathfrak{N} = 1$ represents that the value is provably non-negative, and vice-versa. The non-negative analysis is sound if:
$$\forall x, \mathfrak{N} = 1 \implies f(x)_0 = 0 \land$$
$$\mathfrak{N} = 0 \implies f(x)_0 = 1$$
The non-negative analysis is precise if:
$$\forall x, f(x)_0 = 0 \implies \mathfrak{N} = 1 \land$$
$$f(x)_0 = 1 \implies \mathfrak{N} = 0$$

- Value is provably a power of two: Let $\mathbb{P}$ be the result of the power of two analysis for the output of function f defined in previous analyses. Let $f(x)$ represent the values of f for all inputs x. $\mathfrak{P} = 1$ represents that the value is provably a power of two. The power of two analysis is sound if:

$$\forall x,\ \mathbb{P} = 1 \implies (f(x) \wedge f(x-1) = 0)\ \wedge (x \neq 0)$$

The power of two analysis is precise if:

$$\forall x,\ (f(x) \wedge f(x-1) = 0)\ \wedge (x \neq 0) \implies \mathbb{P} = 1$$

3.5.4 Integer Range Analysis

Integer ranges are used to optimize away comparisons, for example $[0, 100) < [200, 205)$ can be simplified to "true." LLVM's Correlated Value Propagation pass attempts to optimize every comparison in a program in this fashion.

Lazy Value Info (LVI) is LLVM's version of the classic integer range analysis; it computes a *constant range* that has one of four forms:

1. Empty set: concretization set is empty
2. Full set: concretization set is all values of the integer type
3. Regular range $[a, b)$ with $a <_u b$: concretization set contains all values $\geq a$ and $< b$
4. Wrapped range: $[a, b)$ with $a >_u b$: concretization set contains all values either $\geq a$ or $< b$

Here the $>_u$ and $<_u$ operators indicate unsigned integer comparison. So, for example, the concretization set of the wrapped range $[11, 10)$ would include every value except 10. Constant ranges where $a = b$ are invalid unless $a = b = 0$ or $a = b =$ UINT_MAX, which respectively represent the empty and full sets.

This analysis is sound if the concretization set of the analysis result $\mathbb{R}$ is a superset of the union of the results of applying the concrete operation to all possible inputs:

$$\gamma(\mathbb{R}) \supseteq \bigcup_{\forall a} f(a)$$

Unfortunately, the corresponding comparison in the abstract domain—ensuring that the analysis result is high enough in the lattice—does not work because this abstract domain does not actually form a lattice. To see this, notice that the abstraction function is not forced to return a unique best result, but rather must decide between a regular and wrapped constant range. We sidestep this problem by dictating that an integer range is maximally precise if it always returns a result whose concretization set is as small as possible.

3.5.5 Demanded Bits

So far, the analyses from LLVM that we have considered are *forward analyses*: they track data flow facts through the program in the same direction that data flows during execution. The *demanded bits* analysis is a *backwards program analysis* that pushes facts in the opposite direction; it looks for bits whose value does not matter. For example, if a 32-bit value is truncated to 8 bits (and has no other uses in the program), then the demanded bits analysis can return false for its top 24 bits, indicating that they are not demanded.

A demanded bits analysis for f is sound if every not-demanded bit of the input can be set to either zero or one without changing the result of the computation:

$$\forall i \in 0 \ldots W-1, \mathbb{D}_i = 0 \implies$$
$$\forall x : (f(x) = f(\text{setBit}_i(x))) \wedge (f(x) = f(\text{clearBit}_i(x)))$$

Here $\mathbb{D}$ is the result of the demanded bits analysis and $\text{setBit}_i()$ and $\text{clearBit}_i()$ are functions that respectively force bit i of their inputs to one and zero.

This analysis is maximally precise if, for every demanded bit, there exists an input value causing f to return a different result than it produces when that bit of input is forced to either zero or one:

$$\forall i \in 0 \ldots W-1, \mathbb{D}_i = 1 \implies$$
$$\exists x : (f(x) \neq f(\text{setBit}_i(x))) \vee (f(x) \neq f(\text{clearBit}_i(x)))$$

When some bits are not demanded, it is sometimes the case that a computation can be replaced by a simpler one that only produces the desired result for the demanded bits. When no bits are demanded, a value is dead and the instructions producing it can be removed (assuming their values have no other uses).

3.6 Dataflow Analysis in Souper

This section introduces the representation of all dataflow analyses under study in Souper's intermediare representation and its encoding in the form of a solver query. As the search space continues to grow, it gets harder for the superoptimizer, Souper, to synthesize the transformations because of a limited scope. This section will show how dataflow analyses help in increasing the scope and capability of Souper.

3.6.1 Representation in Souper IR

This section describes how we extended Souper to support dataflow analysis facts. Since LLVM developers are an important kind of client for Souper, we designed the syntax of dataflow information in Souper to mirror LLVM's syntax as closely as possible. For example, since LLVM has a utility function 'computeKnownBits()' for performing the known bits analysis, we use the string `knownBits` to label these facts in Souper.

- **Known Bits:** The value `%0` represents a 16-bit variable with most significant bit known to be zero, the least significant bit known to be one, and the remaining bits cannot be proven to hold either value.

```
%0:i16 = var (knownBits=0xxxxxxxxxxxxxx1)
```

- **Nonnegative, Negative, Nonzero, Power of two:** The value **%0** represents a 32-bit nonnegative variable, **%1** is a negative variable, **%2** is a nonzero variable, **%3** is a power of two variable.

```
%0:i32 = var (nonNegative)
%1:i32 = var (negative)
%2:i32 = var (nonZero)
%3:i32 = var (powerOfTwo)
```

- **Number of Sign Bits:** The variable `%0` has at least three high-order bits that all have the same value.

```
%0:i64 = var (signBits=3)
```

- **Integer Range:** The 32-bit variable **%0** has a value in the range that starts at -1 and ends at 255. In the integer range, we follow the same specification as LLVM where the lower bound is included but the upper bound is excluded. The variable **%1** has the value in the range [30, 2). This contains all the values starting at 30 and goes all the way to 32-bit INT_MAX value and wraps around to a 32-bit INT_MIN value until the value 1. Thus, for the integer range where lower bound value is bigger than the upper bound value, the range wraps around.

```
%0:i32 = var (range=[-1, 255))
%1:i32 = var (range=[30, 2))
```

- **Demanded Bits:** While the previous dataflow facts decorate inputs since they are the results of forward analyses, demanded bits decorate outputs since they are the results of a backwards analysis. In this example, an `and` instruction is unnecessary and can be replaced with its input `y` because the only output bits that matter can be shown to come from `y`:

```
%x:i8 = var (knownBits=1111xxxx)
%y:i8 = var
%out = and %x, %y
infer %out (demandedBits=11110000)
result %y
```

3.6.2 Encoding in Solver Query

To verify existing optimizations and as part of the algorithm for synthesizing new ones, Souper translates its IR into SMTLIB [1] and sends it to an SMT solver. Incorporating constraints that come from dataflow facts into solver queries is relatively straightforward. For example, the known bits constraint is encoded by or-ing the input variable with a mask of bits known to be one and then and-ing the variable with a mask of bits that is the complement of the bits known to be zero.

The power-of-two constraint is specified by masking the two conditions. The first is encoded by checking that an input variable is not equal to zero. Second is encoded by checking the result of masking an input variable with a value one less than the input variable is equal to zero.

The constant range constraint that contains a lower bound and an upper bound involves some complexity depending on whether the range is wrapped or not. If the range is not wrapped, it is encoded by masking two conditions. The first specifies that lower bound is unsigned less than or equal to the input variable, and the second criterion specifies that the same input variable is unsigned less than the upper bound of the range. However, if the range is wrapped, it is encoded by or-ing the two conditions that are mentioned above.

3.6.3 Example

Integer overflows, a common kind of bug, can be detected by a programming language implementation. The Swift language, which compiles to LLVM IR, traps all integer overflows; the Clang front end that compiles C and C++ to LLVM IR optionally traps integer overflows.

The accepted LLVM-level idiom for overflow detection is to use intrinsic functions that provide an overflow flag that is similar to the one provided by common processor architectures. For example, the first instruction in this LLVM code multiplies two 32-bit numbers using an intrinsic that returns a structure containing both the result and an overflow flag; the second instruction extracts the overflow flag:

```
%mul = call { i32, i1 } \
  @llvm.smul.with.overflow.i32(i32 %x, i32 %y)
%overflow = extractvalue { i32, i1 } %mul, 1
```

This optimization, which assumes that LLVM was able to conclude that the inputs respectively have 13 and 21 sign bits, shows Souper proving that the overflow bit is zero:

```
%x:i32 = var (signBits=13)
%y:i32 = var (signBits=21)
%0 = smul.with.overflow %x, %y
%1 = extractvalue %0, 1
infer %1
result 0:i1
```

In other words, the multiplication cannot overflow. A subsequent LLVM optimization pass will notice that intrinsic function's overflow result is not used, and will lower the intrinsic call to a regular multiplication instruction. This optimization works any time the total number of sign bits across the inputs is $\geq (\text{bitwidth} + 2)$.

3.6.4 Effect of Dataflow Analysis on Superoptimization

We will discuss the effects of using dataflow facts from LLVM towards improving the superoptimization. Giving Souper access to dataflow information computed by LLVM did not make a huge difference in Souper's effectiveness. In retrospect this is not too surprising. Our hypothesis is that the program features that stop Souper from extracting an LLVM instruction (a load from memory, a function call, etc.) also, in many cases, stop LLVM's dataflow analyses. Thus, LLVM is seldom able to supply Souper with facts that Souper was not already able to derive on its own. However, in the future we plan to artificially stop Souper's instruction extraction early, in order to derive the kind of optimizable code that is seen by a traditional peephole optimizer. In this case we expect LLVM's dataflow analyses to contribute significant additional information.

When extracting dataflow information from LLVM, Souper attempts to optimize about 423,000 distinct code patterns, an increase of 3.2% compared to no dataflow support. The number of distinct optimizations was 7,513, an increase of 3.1%. These optimization fired about 265,000 times while compiling LLVM, an increase of about 20%. However, the code size of the resulting binary decreased by only about 6.5 KB due to dataflow information. These results are summarized in Table 3.1. The "new distinct optimizations" mentioned in the table are optimizations that Souper finds, that are not performable by LLVM. We detect these by converting Souper IR back into LLVM IR and seeing if it optimizes. The reason that Souper finds optimizations that LLVM can also perform is that when the Souper optimization pass is invoked (which happens five times during an `-O2` compilation), the LLVM optimizer has not reached a fixpoint. In fact, LLVM does not reach a fixpoint even after all of its passes have run.

Table 3.2 shows the number of optimizations (and distinct optimizations) occurring in Souper optimizations. As mentioned earlier, we believe the relatively low numbers are due to Souper's extraction strategy. Supporting this view, the demanded bits fact occurs a relatively large number of times—this is because Souper extracts in a backwards direction whereas demanded bits information comes from instructions that are forward of the instruction being analyzed. It appears that demanded bits information is relatively ineffective in giving Souper additional optimization power; we are still investigating why this is the case. These experiments are performed for LLVM+Clang 5.0 when we did not support constant range dataflow analysis in Souper. However, performing this same experiment for an updated version of LLVM and Souper should not be hard.

These are some optimizations that are enabled by dataflow information. The examples that we chose here are ones that occur the most often while building LLVM+Clang using Souper. The first example shows that if all bits except the bottom one are zero, and if we pass a branch indicating the overall value is not equal to zero, then the value must be one.

```
%0:i8 = var (knownBits=0000000x) ; input is 8-bit integer, all bits known
                                 ; to be zero except lowest bit
%1:i1 = eq 0:i8, %0              ; %1 is true if %0 = 0
pc %1 0:i1                       ; path condition: %1 is false
%2:i64 = zext %0                 ; zero-extend %0 to 64-bit
infer %2
```

⇒

```
result 1:i64
```

The second example shows that an input value with 30 sign bits have thirty higher-order bits set to zero or one. When it is added to constant 6, the result will at most have five unknown bottom bits leaving 27 sign bits. Adding -1 to zero-extended result doesn't change the upper bound on number of unknown bits. This makes it obvious that the result must be less than 64.

```
%0:i32 = var (signBits=30) ; input is 32-bit integer that has 30 higher
                           ; bits with same value
%1:i32 = addnsw 6:i32, %0  ; add no signed wrap
%2:i64 = zext %1           ; zero-extend 32-bit %1 to 64-bit
%3:i64 = addnsw -1:i64, %2 ; addition -1 + %2
%4:i1 = ult %3, 64:i64     ; %4 is true if %3 < 64
infer %4
```

⇒

```
result 1:i1
```

Another example shows that if the bottom three bits of input value are zero, and if it passes a branch indicating the overall value is unsigned less than 8, then the value must be 0. Adding 8 to this input value gives 8.

```
%0:i64 = var (knownBits=x...x000) ; input is 64-bit integer with
                                  ; 3 bottom bits clear
%1:i1 = ult %0, 8:i64             ; %1 is true if %0 < 8
pc %1 1:i1                        ; path condition: %1 is true
%2:i64 = add 8:i64, %0            ; addition 8 + %0
infer %2
```

⇒

```
result 8:i64
```

The last example here shows that if we pass the branch indicating that a nonzero value (%0) is less than zero-extended value (%2), then %0 must have at least 48 zeros in higher bits. When -1 is added to this input value, it is obviously less than %2.

```
%0:i64 = var (nonZero)       ; input is 64-bit nonzero integer
%1:i16 = var                 ; input is 16-bit integer
%2:i64 = zext %1             ; zero-extend 16-bit %1 to 64-bit
%3:i1 = ult %0, %2           ; %3 is true if %0 < %2
pc %3 1:i1                   ; path condition: %3 is true
%4:i64 = addnsw -1:i64, %0   ; addition -1 + %0
%5:i1 = ult %4, %2           ; %5 is true if %4 < %2
infer %5
```

⇒

```
result 1:i1
```

3.7 References

[1] C. Barrett, A. Stump, and C. Tinelli, *The SMT-LIB Standard: Version 2.0*, in Proceedings of the 8th International Workshop on Satisfiability Modulo Theories, A. Gupta and D. Kroening, eds., SMT 2010, 2010.

[2] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, *Efficiently computing static single assignment form and the control dependence graph*, ACM Transactions on Programming Languages and Systems (TOPLAS), 13 (1991), pp. 451–490.

[3] L. Torczon and K. Cooper, *Engineering A Compiler*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd ed., 2007.

Table 3.1. Results of compiling LLVM 5.0 using Souper with and without dataflow information

Factor	No dataflow	With dataflow
Code size (bytes)	79,798,452	79,791,868
Size improvement	1.58 MB	1.59 MB
Distinct optimizations	7284	7513
New distinct optimizations	7090	7325
Total optimizations	221,880	264,892

Table 3.2. Occurrences of dataflow facts in optimizations performed by Souper while building LLVM+Clang 5.0

Dataflow fact	Distinct optimizations	Total optimizations
Nonzero	675	2697
Known bits	320	4717
Sign bits	169	4334
Nonnegative	156	4229
Negative	8	101
Power of two	0	0
Demanded bits	276	96804

CHAPTER 4

TESTING STATIC ANALYSES FOR SOUNDNESS AND PRECISION

This chapter presents the work on testing static analyses for soundness and precision. Section 4.1 motivates the idea of testing static analyses and gives an overview of the problem we addressed in this chapter. The detailed design and implementation of maximally precise algorithms are presented in Section 4.2. Section 4.3 evalutes our work and discusses the imprecisions and soundness errors in LLVM's dataflow analyses.

4.1 Introduction

Static analysis is one of the basic technologies that enables compilers to generate optimized executables for code in high-level languages. For example, an integer overflow check can be removed when the operation provably does not overflow, and an array bounds check can be removed when the access is provably in-bounds. In each case, the necessary proof is implied by the results of a static analysis.

Production-quality optimizing compilers have typically had significant engineering effort put into their static analyses. For example, LLVM currently has about 80,000 lines of C++ in its lib/Analysis directory, which contains much, but not all, of its static analysis code. GCC does not have an analogous directory structure that makes it easy to find static analyses, but for example its integer range analysis alone (tree-vrp.c) is 6,929 lines of C. This code is tricky to get right and, so far, developers are implementing static analyses without any help from formal-methods-based tools.

We have developed algorithms that use an SMT solver to compute sound and maximally precise results for several important dataflow analyses used by LLVM:

- Known bits: a forward analysis attempting to prove that individual bits of a value are always either zero or one
- Demanded bits: a backward analysis attempting to prove that individual bits of a value are "not demanded": their value is irrelevant
- Integer ranges: a forward analysis attempting to prove that an integer-typed value lies within a sub-range, such as [5..10]
- A collection of forward analyses determining Boolean properties of a value, such as whether it is provably non-zero or a power of two

Since 2010, 55 soundness bugs in LLVM's forward bit-level analyses have been fixed. Similarly, three soundness bugs have been fixed in its demanded bits analysis, and 23 soundness errors have been fixed in its integer range analysis. On the other side, static analyses that are insufficiently precise can impede optimization; in one case, increasing the precision of a static analysis in LLVM improved the performance of a Python benchmark by 4.6% [10]. The goal of our work is to develop formal-methods-based algorithms that can be used to find both imprecisions and unsoundnesses in static analyses implemented in production-quality compilers.

Our algorithms work on concrete code fragments, which we gather by compiling real applications. For each such fragment, we compute a sound and maximally precise dataflow fact and then compare this against dataflow information computed by LLVM. The interesting outcomes are where LLVM's result is "more precise" than ours (indicating a soundness error in LLVM) or LLVM's result is less precise than ours (indicating an imprecision in LLVM that may be worth trying to fix). We have reported several precision errors to the LLVM developers, and some of these have been fixed. We did not find any soundness errors in the latest version of LLVM, but we have verified that our method can find previously-fixed bugs. Finally, we have created a version of LLVM that uses our dataflow results instead of its own. This compiler is very slow, but it demonstrates the value of increased precision by sometimes creating binaries that are faster than those produced by the default version of LLVM.

4.2 Design and Implementation

This section describes our work, which must solve several sub-problems.

1. Finding representative LLVM IR on which to test the compiler's dataflow analyses (Section 4.2.1)

2. Ensuring that LLVM's static analyses and our algorithms for computing dataflow results both see the same code, so that their results are directly comparable (Section 4.2.2)

3. Computing precise dataflow results using an SMT solver (Section 4.2.3)

The high-level structure of our method is shown in Figure 4.1.

4.2.1 Finding Test Inputs

We require concrete code fragments to use as bases for comparison between LLVM's dataflow implementations and ours. We got these by compiling all of the C and C++ benchmarks in SPEC CPU 2017 [1, 6] with Souper plugged into LLVM as a middle-end optimization pass, and with Souper's synthesis disabled. This has the effect of loading a large number of Souper expressions into a cache that we subsequently used as a source of test inputs.

After compiling SPEC CPU 2017, we ended up with 269,113 unique Souper expressions. 71.6% of these were encountered more than once during compilation, 11.4% more than 10 times, and 1.6% more than 100 times.

4.2.2 Enabling Comparable Results

A problem we had to solve is that we cannot easily control what LLVM is willing to do to get precise dataflow results, making it hard to put LLVM's analyses and ours on an even playing field. As an example, LLVM's integer range analysis is sometimes interprocedural, because some passes will mark function calls with "range metadata" gathered from callees. This kind of feature puts us in the awkward position of potentially having to either modify LLVM or else emulate all of LLVM's precision-increasing mechanisms in order to get comparable results. We developed a different solution: instead of running LLVM's static analyses on its original IR, we convert Souper's IR back into LLVM IR and then analyze that. This is accomplished using the `souper2llvm` tool shown in Figure 4.1. This strategy ensures that LLVM's analyses and ours are computing dataflow facts over exactly the same code.

4.2.3 Algorithms for Maximally Precise Dataflow Analyses

Like other synthesis problems, computing precise dataflow results using an SMT solver is a search problem that in practice requires multiple solver calls to arrive at the best answer. Our algorithms achieve maximal precision and perform reasonably well by exploiting the structure of the abstract domains that we target. In some cases these algorithms are trivial. For example, all of the single-bit analyses described in Section 3.5.3 can be computed in a maximally precise fashion using a single solver call which simply checks whether the property implied by the fact holds or not. Another analysis, the one computing the number of sign bits, is nearly as easy, since there are only $n-1$ non-trivial possibilities for an n-bit value. We simply try all alternatives, one after the other, starting with the most precise result (n sign bits). In contrast, an exhaustive search strategy is intractable for:

- Demanded bits: 2^n alternatives
- Known bits: 3^n alternatives
- Integer ranges: $4^n - 2^n + 2$ alternatives

The rest of this section is about how to compute these dataflow facts more efficiently and without giving up maximal precision. Our arguments for maximal precision are all contingent on the SMT solver returning a definite answer (SAT or UNSAT) for every query, as opposed to timing out. It is easy to construct an adversarial example (involving 64-bit divisions, for example) that cannot be analyzed in practice, using our methods, because it results in SMT queries that cannot be solved in a reasonable amount of time by a state-of-the-art solver.

4.2.3.1 Known Bits

Algorithm 1 computes known bits efficiently and maximally precisely. The algorithm is not difficult: it simply guesses that each bit is always zero and then always one. In this pseudocode (and also in the pseudocode for the next two algorithms) we elide error-checking code; it should be assumed that the algorithm terminates with a sound (but likely imprecise) result whenever the solver runs out of time or memory. The input, F, in this algorithm is a function computed by some Souper expression. So then, "getInputs(F)" returns the inputs to the Souper expression F, "F.setBit(input, i)" pins bit 1 of one of the inputs to F, etc.

Algorithm 1 maximally precise known bits

```
1: procedure COMPUTEKNOWNBITS(F)
2:     for i ← 0 ... Width($F$) − 1 do
3:         if askSolver(F = F.clearBit(i)) then
4:             $\mathrm{Result_i} \leftarrow 0$
5:         else if askSolver(F = F.setBit(i)) then
6:             $\mathrm{Result_i} \leftarrow 1$
7:         else
8:             $\mathrm{Result_i} \leftarrow$ Unknown
9:     return Result
```

It is a little trickier to prove that bit-by-bit search, which requires only $2n$ solver queries, is maximally precise. The argument is based on *separability*: a property held by a dataflow analysis if the aggregate dataflow function can be viewed as a product of simpler functions on individual data items [11]. If known bits is separable at the bit level, then our bit-by-bit algorithm must always return the most precise result. The abstract domain for known bits forms a *join semilattice* [20] ; every subset of its elements has a join, but not a meet. (Throughout this discussion, we'll refer to join semi-lattices as lattices when this does not seem to risk ambiguity.) The lattice for one bit contains three abstract values as shown in Figure 4.2(a), where elements 0 and 1 are more precise than the top element ⊤ and the join of elements 0 and 1 is ⊤:

$$0 \sqsubseteq \top, \quad 1 \sqsubseteq \top, \quad 0 \sqcup 1 = \top$$

The lattice for two bits, as shown in Figure 4.2(b), is created by taking a cross-product of two one-bit lattices. The elements of the cross-product lattice, $L_1 \times L_2$, are tuples of the form: $<X_1,\ X_2>$ such that every $X_i \in L_i$ lattice. To prove separability of known bits analysis, we need to prove the following three properties as defined by Khedker et al. [11].

Property 4.2.3.1. For all elements in a lattice, the dataflow property is independent, and thus can be ordered element-wise.

$\forall X, Y \in L$: $X \sqsubseteq Y \equiv <X_1 \sqsubseteq Y_1,\ X_2 \sqsubseteq Y_2,\ \ldots,\ X_n \sqsubseteq Y_n>$

Property 4.2.3.2. For each element X in a cross-product lattice, where X is a tuple of component elements, there exists a set of functions f in a bigger set of transfer functions $\mathfrak{F}$ for a dataflow framework, that when applied over aggregate element X is equivalent to a tuple of smaller abstract functions that are applied on component elements of X. In short, the separable function space can be factored into a product of the functions' spaces for individual data items.

$\forall f_i\colon L_i \rightarrow L_i$, $1 \leq i \leq n$, such that $\forall f \in \mathfrak{F}$, $f(X) \equiv <f_1(X_1), f_2(X_2), \ldots, f_n(X_n)>$.

Both of these properties are clearly held by a lattice like known bits that is created by concatenating items corresponding to individual bits: there is no interaction across bits in this abstract domain.

Property 4.2.3.3. The height of each L_i is bounded by a constant.

The height of the known bits lattice is one larger than the number of bits being analyzed.

Separability establishes that a transfer function applied to an element of a cross-product lattice gives the same solution as the single-bit transfer functions applied individually. This indicates that both approaches compute the same fixed point. Thus, our bit-by-bit algorithm is maximally precise. (Our reasoning here is empirically supported by the fact that over a large number of expressions harvested from LLVM IR, our known bits result was never less precise than the one computed by LLVM.)

4.2.3.2 Demanded Bits

Algorithm 2 attempts to prove each bit is non-demanded by forcing its value to zero and then using the solver to check if the resulting Souper expression is equivalent to the original expression. If so, we force its value to one and do another equivalence check.

If both checks succeed, we have proved that the bit's value does not affect the computation, and therefore that bit is not demanded. The demanded bits abstract domain is separable following a similar argument to the one we used for known bits, and so this algorithm is also maximally precise.

Algorithm 2 maximally precise demanded bits

```
1:  procedure computeDemandedBits($F$)
2:      InputList ← getInputs($F$)
3:      for each input in InputList do
4:          for i ← $0 \ldots$ Width(input) $- 1$ do
5:              if askSolver(F.setBit(input, i) = F) ∧
6:                askSolver(F.clearBit(input, i) = F) then
7:                  Result.input$_i$ ← 0
8:              else
9:                  Result.input$_i$ ← 1
10:     return Result
```

4.2.3.3 Integer Ranges

To find the most precise integer range, we are searching for a range $[X, X + C)$ that excludes more integer values than any other dataflow result. In other words, $[X, X + C)$ is a sound dataflow fact but no sound result exists for any $[X_2, X_2 + C_2)$ where $C_2 < C$. Unfortunately, the integer range abstract domain is not made of conveniently separable smaller parts, and so finding the most precise result is a more difficult problem that the ones we have faced so far.

The smallest feasible value of C can be found using binary search. Since we are not aware of a direct method for efficiently finding an X corresponding to each choice of C, we find X using synthesis. The synthesizeBase() function in Algorithm 3 attempts to find an X that, combined with a choice of C, results in a sound integer range. It uses a CEGIS-style loop to synthesize the constant, using *generalization by substitution* [9] to rule out classes of choices of X that do not work in each iteration of this loop. This function guides the binary search for the smallest C that is implemented in computeIntegerRange().

The proof of maximal precision for Algorithm 3 follows from the fact that—assuming synthesis succeeds—it always finds an integer range excluding the most possible values from the concretization set.

4.3 Results

This section presents our findings on imprecisions (Section 4.3.1, 4.3.2) and soundness errors (Section 4.3.4) in LLVM's dataflow analyses, and the concrete imporcements (Section 4.3.5) our maximally precise algorithms made towards improving LLVM. Section 4.3.3 studies the impact of maximal precision of dataflow analyses on code generation.

Algorithm 3 maximally precise integer ranges

```
1: procedure SYNTHESIZEBASE(F, $C_0$)
2:     return constantSynthesis(
3:         if $X + C_0 \leq$ UINT_MAX then                    ▷ Regular range
4:             $F \in [X, X + C_0)$
5:         else                                               ▷ Wrapped range
6:             $F \in [X,$ UINT_MAX$) \cup [0, X + C_0 -$ UINT_MAX$)$
7:     )
8: procedure COMPUTEINTEGERRANGE(F)
9:     $L \leftarrow 1$
10:    $R \leftarrow$ UINT_MAX
11:    while $L \leq R$ do                                     ▷ Binary Search
12:        $M \leftarrow L + (R - L) / 2$
13:        temp $\leftarrow$ synthesizeBase(F, M)
14:        if temp.success then
15:            $X_1 \leftarrow$ temp
16:            $C_1 \leftarrow M$
17:            $R \leftarrow M - 1$
18:        else
19:            $L \leftarrow M + 1$
20:    return $[X_1, X_1 + C_1)$
```

4.3.1 Precision Comparison

We compared the precision of our algorithms against LLVM's dataflow analyses for every Souper expression encountered while compiling the C and C++ programs in version 1.0.1 of the SPEC CPU 2017 benchmark suite: 269,113 in all. Each Souper expression has a single output (where forward analysis results end up), but may have many inputs (where the backwards dataflow information ends up). Hence, the total number of comparisons for demanded bits is over 2.1 million variables. The average Souper expression in our experiments contains 98 instructions (Souper instructions are mostly isomorphic to LLVM instructions). The largest expression is 3,665 Souper instructions. Our implementation is based on the latest version of Souper as of May 28, 2019 and LLVM 8.0. SMT queries were answered using Z3 [8] ; individual solver queries were timed out after 30 seconds. We also timed out any dataflow computation for a Souper expression that required more than five minutes of total execution time. Because the evaluation was time-consuming, we ran it across several machines of generally comparable (per core) power: a six-core Intel i7, a 32-core Threadripper 2, and a machine with two 28-core Xeon processors.

Table 4.1 summarizes the results of our precision experiment. In many cases, LLVM's analyses and ours have the same precision. This reflects the fact that LLVM's static analyses are quite good: their precision has been gradually improved and tweaked over a number of years. In a substantial minority of cases, our result is more precise than LLVM's. Each such example indicates an opportunity to improve the precision of an analysis in LLVM; we present some examples in the next section. We found no new soundness bugs in LLVM 8.0; this kind of bug would be signaled by LLVM computing a dataflow result that is "more precise" than our maximally precise result.

The right-most column of Table 4.1 confirms that our precise dataflow implementations are too slow to use inside a compiler. They are suitable only as an offline test oracle. Additionally, we ran into many timeout issues while computing integer ranges and demanded bits: both of these algorithms failed to compute a result around half of the time. We spent some time investigating these failures and it appears that the queries being posed are simply difficult for Z3 to handle.

4.3.2 Examples of LLVM Imprecisions

This section shows some examples where LLVM's dataflow analyses return imprecise results. All of these code fragments originated in SPEC CPU 2017, but in some cases we have reduced bitwidths to make the examples easier to understand. For each example, we present a fragment of LLVM-like code, followed by the maximally precise dataflow result and also LLVM's dataflow result.

4.3.2.1 Known Bits

Here, the value 32, represented as an 8-bit integer, is shifted left by a variable amount %x:

```
%0 = shl i8 32, %x

Precise %0: xxx00000
LLVM    %0: xxxxxxxx
```

Our known bits algorithm recognizes that any trailing zeroes in the original number cannot be eliminated by a left-shift operation, but LLVM returns a completely unknown result (indicated by the "x" in each bit position).

Here a four-bit value `%x` is zero-extended to eight bits and then right-shifted by a variable amount `%y`:

```
%0 = zext i4 %x to i8
%1 = lshr i8 %0, %y

Precise %1: 0000xxxx
LLVM    %1: xxxxxxxx
```

Our algorithm recognizes that high zeros cannot be destroyed by a logical right shift, whereas LLVM's analysis returns an all-unknown result.

Here, LLVM misses a known bit at the low end of a word:

```
%0 = and i8 1, %x
%1 = add i8 %x, %0

Precise %1: xxxxxxx0
LLVM    %1: xxxxxxxx
```

Its transfer function for adding known bits fails to recognize that the low bits of `%x` and `%0` are either both cleared or both set, forcing the low bit of their sum `%1` to be cleared.

When a multiplication by 10 is not allowed to overflow (the `nsw` qualifier makes signed overflow undefined) its result must be evenly divisible by 10, allowing an analysis to determine that all bits in the resulting value are zero:

```
%0 = mul nsw i8 10, %x
%1 = srem i8 %0, 10

Precise %1: 00000000
LLVM    %1: xxxxxxxx
```

LLVM's known bits computation takes advantage of integer range information, if available, but misses the fact that adding one to an integer in the range 0..4 cannot affect any bit beyond the third:

```
%x = range [0,5)
```

```
%0 = add i8 1, %x

Precise %0: 00000xxx
LLVM    %0: 0000xxxx
```

There are more examples along these patterns where LLVM returns imprecise known bits for a given input expression.

4.3.2.2 Power of Two Analysis

This section shows three LLVM fragments whose result is a power of two, but where LLVM's power-of-two dataflow analysis fails to derive that fact.

A value that is constrained to be either one or two is clearly a power of two:

```
%x = range [1,3)
```

The idiom $x \,\&\, -x$ is a commonly-used way to rapidly isolate the right-most set bit in a word. LLVM's power of two analysis recognizes this idiom, but does not connect it with range information specifying that the value cannot be zero:

```
%x = range [1,0)
%0 = sub i64 0, %x
%1 = and i64 %x, %0
```

Here a value that LLVM can easily prove to be a power of two, `%1`, is truncated; LLVM conservatively drops the power-of-two fact on the assumption that the one bit may be chopped off, failing to recognize that the constrained shift amount makes that impossible:

```
%0 = and i32 7, %x
%1 = shl i32 1, %0
%2 = trunc i32 %1 to i8
```

These examples are among the most commonly occuring expressions for which LLVM cannot prove power of two dataflow fact.

4.3.2.3 Demanded Bits

Recall that demanded bits is a backwards analysis; in these examples dataflow facts will be presented for the inputs, not the result, of an LLVM fragment. If the demanded bits analysis returns zero, then the value in that bit position provably does not matter.

We found many variations on this theme, where some bits of a value feeding a comparison were not demanded:

```
%0 = icmp slt i8 %x, 0

Precise demanded bits for %x: 10000000
LLVM demanded bits for    %x: 11111111
```

We also found many variations of this theme, where some bits of a value feeding an unsigned division were not demanded:

```
%0 = udiv i16 %x, 1000

Precise demanded bits for %x: 1111111111111000
LLVM demanded bits for    %x: 1111111111111111
```

Overall, LLVM's demanded bits showed a lot of missed opportunities for improving the precision.

4.3.2.4 Integer Range Analysis

Recall LLVM's integer range representation from Section 3.5.4, which is closed with respect to the lower bound and open with respect to the upper bound, and which supports wrapped ranges.

The `select` instruction is LLVM's ternary conditional expression, analogous to the `?:` construct in C and C++. The expression below returns one if `%x` is zero, and returns `%x` otherwise. Thus, the value zero is excluded from the result set:

```
%0 = icmp eq i32 0, %x
%1 = select i1 %0, i32 1, %x

Precise range for %1: [1,0)
LLVM range for    %1: full set
```

In this example, it is clear that any value in the range 1..6 will be unchanged by anding with $0xFFFFFFFF$, but LLVM pessimizes the resulting integer range slightly anyway:

```
%x = range [1,7)
%0 = and i32 4294967295, %x

Precise range for %0: [1,7)
LLVM range for    %0: [0,7)
```

The signed remainder of a 32-bit integer with eight is always in the range $-7..7$. LLVM's result $-8..7$ is pessimistic:

```
%0 = srem i32 %x, 8

Precise range for %0: [-7,8)
LLVM range for    %0: [-8,8)
```

The unsigned division operator is also analyzed imprecisely:

```
%0 = udiv i64 128, %x

Precise range for %0: [0,129)
LLVM range for    %0: full set
```

LLVM computed imprecise range dataflow results for many expressions along the common themes presented in this section.

4.3.3 Impact of Maximal Precision on Code Generation

One might ask: What effect would increasing the precision of LLVM's dataflow analyses have on the quality of its generated code? To investigate this question, we created a version of LLVM 8.0 that has maximally precise forward bit-level dataflow analyses by invoking our algorithms. (We did not modify LLVM to call our demanded bits or integer range implementations.)

We compiled several smallish applications—gzip, bzip2, SQLite, and Stockfish —using LLVM 8.0 and also our modified LLVM 8.0, and compared their performance. We used the compiler flags that each application intended to use for the release (or default, if there was only one) build mode. We used "-O3" to compile gzip, bzip2, SQLite, and "-O3 -fno-exceptions -msse -msse3 -mpopcnt" to compile Stockfish. Compilation time ranged from about an hour for bzip2 to more than 70 hours for SQLite. (We would have liked to test SPEC CPU 2017 in this fashion, but compile times for its larger applications such as GCC were exceedingly long.)

To test the performance of gzip and bzip2, we compressed the 2.9 GB ISO image for SPEC CPU 2017, and decompressed the resulting compressed file. To test Stockfish (a chess engine) we ran a single-threaded benchmark computing the next move in 42 chess games that are part of its test suite, using a depth of 26 and hash table size 1024. To test SQLite (an embedded database) we ran a workload suggested in its documentation: we create a SQL database with 2,500,000 insertions using a transaction and then run 100 selects on it.

We measured performance on two machines with diverse microarchitectures: an AMD Threadripper 2990WX and an Intel Core i7-5820K. Both machines were idle except for a single core running our benchmarks. We ran each test five times and report the average execution time.

Table 4.2 summarizes the results of our experiment. The most interesting data point is the speedup of the compression side of bzip2. We looked at the differences between the baseline and precise versions of this program. At the level of LLVM IR, there were only a few minor differences, indicating that for this program, the middle-end optimizers did not benefit much from increased dataflow analysis precision. However, the executables were different in hundreds of locations because the SelectionDAG pass in the x86-64 LLVM backend makes heavy use of known bits information. The baseline and precise versions of bzip2 both executed roughly the same number of instructions during compression, but the instructions per cycle was considerably higher for the precise version.

4.3.4 Finding Soundness Errors

Since we did not find any new soundness errors in the dataflow analyses from LLVM 8.0 that we looked at, we wanted to make sure that our method can in fact detect soundness bugs. To do this, we looked for soundness bugs that had been previously fixed in LLVM, and selected three of them that had reasonably isolated patches, allowing us to port them forward to LLVM 8.0. We then used the regression tests attached to the three patches fixing these bugs to ensure that our artificially broken LLVM properly exhibits each soundness bug.

- **Soundness Bug 1:** An LLVM patch [2] introduced many dataflow improvements including an unsound one which leads LLVM to believe that a value is provably non-zero if it is the sum of two provably non-negative values. Of course that is wrong: both non-negative values may be zero. This bug was first fixed partially and then completely in a pair of commits [3, 4].

  When we ran the Souper expressions harvested from SPEC CPU 2017 through our precision checking tool, it detected this soundness bug by producing this output:

  ```
  %0:i32 = add 0:i32, 0:i32
  infer %0

  non-zero from our tool: false
  non-zero from llvm: true
  llvm is stronger
  ```

  For this trivial example, the buggy LLVM infers that the sum is non-zero. “llvm is stronger” means that LLVM was able to compute a stronger dataflow result than the one our algorithm computed, which is maximally precise. This is how our tool points out a soundness bug.

- **Soundness Bug 2:** A miscompilation bug [12] in LLVM had a dataflow unsoundness bug as its root cause. The bug was fixed in patch [7]. The problem is in analysis of the signed remainder operation where the divisor is constant. This code was also triggered using one of the Souper expressions harvested by compiling SPEC:

  ```
  %0:i32 = var
  %1:i32 = srem %0, 3:i32
  infer %1

  known sign bits from our tool: 30
  known sign bits from llvm: 31
  llvm is stronger
  ```

Here, LLVM incorrectly believes that the signed remainder of an arbitrary 32-bit value and 3 has 31 sign bits, which is one more than it actually has.

- **Soundness Bug 3:** This wrong code bug [18] was again in the code handling the signed remainder operation, but this time in the "known zero" analysis. It was fixed in a patch [5]. The example that triggered this bug was originally in 64 bits, but to improve readability we present an 8-bit version of it here:

```
%0:i8 = var
%1:i8 = srem 4:i8, %0
infer %1

known bits from our tool: 00000x0x
known bits from llvm: 00000x00
llvm is stronger
```

  This bug was not triggered by any Souper expression found when compiling SPEC CPU 2017. However, we did trigger it using a collection of Souper expressions harvested by compiling a number of programs randomly generated by Csmith [21] and Yarpgen [13]. Given these facts, it is perhaps unsurprising that the miscompilation error in LLVM that this bug caused had been originally discovered by Csmith.

4.3.5 Concrete Improvements to LLVM

Some time ago (prior to the LLVM 5.0 release) we performed an early solver-based study of just the known bits analysis in LLVM, and worked with LLVM developers to get some of the worst imprecisions fixed. Some of the patches discussed below were written by us, others were written by LLVM developers.

(1) Evaluating $x \wedge (x-1)$ results in a value that always has the bottom bit cleared. The patch added to LLVM handles a slightly generalized pattern $x \wedge (x-y)$ where y must be odd. (Note: In this expression, ($\wedge$) is used for bitwise conjunction operation, and ($\vee$) is used for bitwise disjunction operation.) [17].

(2) The LLVM byte-swap intrinsic function, used to change the endianness of a value, was not previously handled by the known bits analysis. This is fixed now [15].

(3) The additive inverse of a zero-extended value, $0 - \text{zext}(x)$, is always negative [14].

(4) The result of the Hamming weight intrinsic in LLVM (`@llvm.ctpop`) had room for improvement [16].

(5) Tests for equality and inequality can sometimes be resolved at compile time if, for example, $x = y$ is being evaluated and at some bit position, x is known to have a zero and y is known to have a one [19].

Manually identifying imprecisions in dataflow analyses is difficult. Given some code to compile, we can identify imprecisions automatically. In this section we showed that five such issues in LLVM have already been fixed based on the results of our work.

4.4 References

[1] *Standard performance execution corporation, spec cpu 2017*, 2017. http://spec.org/cpu2017.

[2] Baldrick, *Revision 124183*, 2011. http://llvm.org/viewvc/llvm-project?view=revision&revision=124183.

[3] ——, *Revision 124184*, 2011. http://llvm.org/viewvc/llvm-project?view=revision&revision=124184.

[4] ——, *Revision 124188*, 2011. http://llvm.org/viewvc/llvm-project?view=revision&revision=124188.

[5] ——, *Revision 155818*, 2012. http://llvm.org/viewvc/llvm-project?view=revision&revision=155818.

[6] J. Bucek, K.-D. Lange, and J. v. Kistowski, *Spec cpu2017: Next-generation compute benchmark*, in Companion of the 2018 ACM/SPEC International Conference on Performance Engineering, 2018, pp. 41–42.

[7] S. Das, *Revision 233225*, 2015. http://llvm.org/viewvc/llvm-project?view=revision&revision=233225.

[8] L. De Moura and N. Bjrner, *Z3: An efficient smt solver*, in International conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer, 2008, pp. 337–340.

[9] B. Dutertre, *Solving exists/forall problems with yices*, in Workshop on Satisfiability Modulo Theories, 2015.

[10] Y. Hiroshi, *[lvi] constant-propagate a zero extension of the switch condition value through case edges*, 2017. http://reviews.llvm.org/rL309415.

[11] U. P. Khedker and D. M. Dhamdhere, *A generalized theory of bit vector data flow analysis*, ACM Transactions on Programming Languages and Systems (TOPLAS), 16 (1994), pp. 1472–1511.

[12] N. Lewycky, *Miscompile of % in loop*, 2015. http://bugs.llvm.org/show_bug.cgi? id=23011.

[13] V. Livinskii, D. Babokin, and J. Regehr, *Random testing for c and c++ compilers with yarpgen*, Proceedings of the ACM on Programming Languages, 4 (2020), pp. 1–25.

[14] D. Majnemer, *Instsimplify: Optimize signed icmp of -(zext v)*, 2014. https://reviews.llvm.org/D3754.

[15] P. Reames, *Extend known bits to understand @llvm.bswap*, 2015. https://reviews.llvm.org/D13250.

[16] ——, *Tighten known bits for ctpop based on zero input bits*, 2015. https://reviews.llvm.org/D13253.

[17] ——, *[valuetracking] recognize that and(x, add (x, -1)) clears the low bit*, 2015. https://reviews.llvm.org/rL252629.

[18] J. Regehr, *Wrong code bug*, 2012. http://bugs.llvm.org/show_bug.cgi? id=12541.

[19] Suyog, *Instcombinecompare with constant return false if we know it is never going to be equal*, 2014. https://reviews.llvm.org/D3868.

[20] Wikipedia-Semilattice, *Semilattice — wikipedia, the free encyclopedia*, 2019. [Online; accessed 01-09-2019].

[21] X. Yang, Y. Chen, E. Eide, and J. Regehr, *Finding and understanding bugs in C compilers*, in Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '11, 2011, pp. 283–294.

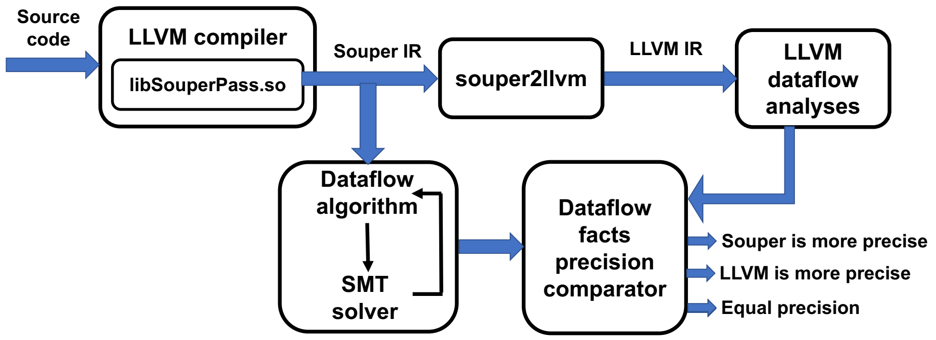


Figure 4.1. Overview of our method for finding imprecisions or unsoundnesses in LLVM's dataflow analyses. libSouperPass.so is dynamically loaded into LLVM to allow Souper to run as an optimization pass. souper2llvm is a utility we created that translates Souper IR into LLVM IR.

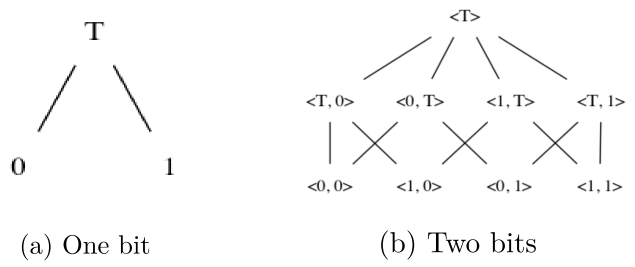


Figure 4.2. The known bits abstract domain

Table 4.1. Comparing the precision of LLVM's dataflow analyses (DFA) and our dataflow algorithms. The "resource exhaustion" category includes cases where the solver timed out, where the solver used too much RAM, and where Souper's constant synthesis procedure fails. CPU time per expr is an average CPU time for each expression.

DFA	Same precision	Souper more precise	LLVM more precise	Resource exhaustion	CPU Time per expr
Known bits	227,728 (84.6%)	17,722 (6.6%)	0	23,663 (8.8%)	9.0s
Sign bits	227,709 (84.6%)	20,920 (7.8%)	0	20,484 (7.6%)	3.2s
Non-zero	234,833 (87.3%)	13,594 (5.1%)	0	20,686 (7.7%)	2.9s
Negative	248,172 (92.2%)	1,364 (0.5%)	0	19,577 (7.3%)	2.8s
Non-negative	234,699 (87.2%)	13,749 (5.1%)	0	20,665 (7.7%)	3.3s
Power of two	247,209 (91.8%)	1,278 (0.5%)	0	20,615 (7.7%)	3.1s
Integer range	131,081 (48.7%)	22,300 (8.3%)	0	115,575 (42.9%)	16.0s
Demanded bits	741,940 (35.0%)	179,485 (8.4%)	0	1,195,368 (56.5%)	26.6s

Table 4.2. Evaluation of the impact of maximally precise dataflow facts on generated code. The baseline compiler is LLVM 8.0 and the precise compiler is LLVM 8.0 modified to use our dataflow algorithms instead of its own.

Benchmark	AMD machine			Intel machine		
	Baseline	Precise	Speedup	Baseline	Precise	Speedup
bzip2 compress time	333.61s	260.40s	+21.94%	384.12s	353.00s	+8.10%
bzip2 decompress time	148.82s	150.94s	-1.42%	163.81s	164.02s	-0.13%
gzip compress time	75.41s	75.33s	+0.11%	82.22s	84.57s	-2.85%
gzip decompress time	14.09s	14.25s	-1.13%	14.20s	14.32s	-0.84%
Stockfish total time	257.28s	254.68s	+1.01%	273.99s	272.54s	+0.53%
SQLite	37.78s	36.78s	+2.65%	40.01s	39.88s	+0.32%

CHAPTER 5

AUTOMATIC GENERATION OF A PEEPHOLE OPTIMIZER

This chapter presents Cranelift-PeepGen, an automatic peephole optimizer for Cranelift. Section 5.1 motivates the idea of why compiler developers must adopt the technique to automatically geenrate a peephole optimizer instead of manually implementing tons of rewrite rules in a compiler. Section 5.2 discusses the detailed design and implementation of Cranelift-PeepGen, and Section 5.3 studies its performance impact.

5.1 Motivation

Peephole optimizations replace locally inefficient code with equivalent but better code. These optimizations are applied systematically and are expected to reduce the size and execution time of the compiled binary. LLVM's source code is complicated and has peephole optimizations are implemented in different passes. However, the majority of them are in *InstCombine*, a pass that combines redundant instructions and simplifies algebraic expressions. The size of LLVM's instruction combiner has increased from 2,000 LOC to 36,000 LOC over the past 20 years. An optimization [14] to transform several operations (an OR, two ANDs, two NOTs) to an XOR:

```
(A & ~B) | (~A & B) --> A ^ B
```

is implemented in LLVM as:

```
static Instruction *foldOrToXor(BinaryOperator &I,
                                InstCombiner::BuilderTy &Builder) {
  if (match(Op0, m_c_And(m_Value(A), m_Not(m_Value(B)))) &&
      match(Op1, m_c_And(m_Not(m_Specific(A)), m_Specific(B))))
    return BinaryOperator::CreateXor(A, B);

}
```

Similar to the above optimization with an OR operation at the root, two more optimizations with a root node at XOR and ADD operations respectively, fold them to a single XOR operation as:

```
(A & ~B) ^ (~A & B) --> A ^ B
(A & ~B) + (~A & B) --> A ^ B
```

Both of these optimizations save four operations. LLVM implements each of them individually in a different function adding to the growing size of the optimizer.

In contrast, GCC adopts a DSL-based approach to specify these optimizations [13].

```
/* Simplify (A & ~B) |^+ (~A & B) -> A ^ B.  */

(for op (bit_ior bit_xor plus)
 (simplify
  (op (bit_and:c @0 (bit_not @1)) (bit_and:c (bit_not @0) @1))
   (bit_xor @0 @1)))
```

The Go compiler also adopts the DSL-based approach to specify peephole optimizations [3]. In this DSL, these two optimizations would be:

```
(Xor64 (And64 a (Not b)) (And64 (Not a) b)) --> (Xor64 a b)
(Or64 (And64 a (Not b)) (And64 (Not a) b)) --> (Xor64 a b)
```

It seems obvious that specifying these optimizations in a DSL and auto-generating their implementation is a cleaner and more extensible approach. Unfortunately, many compilers do not adopt this approach to lift their optimizations from the rest of the compiler yet. Even though GCC puts an effort to specify some of the simplifications in a DSL format in the *match.pd* file [4], a significant number of its peephole optimizations are implemented in code. MSVC implements its optimizations individually as well. The next challenge that remains unsolved is to automatically find the missing optimizations in a given program and express them in an easy to read and easy to use DSL.

Besides the increasing implementation complexity in writing peephole optimizations, these optimizations are prone to bugs. It is hard to reason about the correctness of these optimizations as there are so many corner-cases involved. A random testing tool, Csmith [16], and a verification tool, Alive [15], found numerous bugs in the peephole pass of LLVM. It is important to get these optimizations implemented correctly as unsound optimizations can lead to miscompilation. This motivates another goal of verifying the optimizations for soundness while we discover the missing ones.

Cranelift, a retargetable code generator integrated in Wasmtime, a runtime system for WebAssembly, did not have a peephole optimization pass until December 2020. Peepmatic, a DSL-based automatic peephole optimization generator was integrated into Cranelift with the goal of specifying transformations more easily. It implements an automaton-based matcher to generate the optimization pass for Cranelift. However, this compiler does not focus on verifying the correctness of the rewrite rules and requires developers to specify transformations instead of automatically generating them and verifying their correctness.

We use formal methods based tools to fulfill these requirements. We use a synthesizing superoptimizer, Souper, that uses an SMT solver to find missing optimizations and generate them in Souper's DSL. The details of the DSL are discussed in Chapter 2. Optimizations synthesized by Souper are verified for correctness as well. We implemented a prefix-tree pattern matching approach to automatically generate an implementation of a peephole optimization pass.

In the rest of this chapter, we will discuss what are the components of Cranelift-PeepGen, an automatic peephole optimizer for Cranelift, their implementation and performance impact.

5.2 Design and Implementation

To automatically generate a peephole optimizer for the Cranelift compiler, we first find the peepholes automatically instead of handwriting them in an intermediate representation or a domain-specific language. Later, we implement a compiler to transform these peepholes in a DSL to code using a prefix-tree pattern matching technique to generate an optimizer.

The next problem is to understand how we can find the peepholes. Searching for the transformations is done in two steps. First, we harvest candidates that can be optimized; these are called left-hand sides (LHSes). Second, we search for an optimized equivalent of a given LHS using an enumerative synthesis technique. The synthesized solution is called a right-hand side (RHS). The left and right-hand sides together form a complete peephole optimization.

The overview of Cranelift-PeepGen design is shown in Figure 5.1. We further discuss the details of our approach along with the design choices we explored.

5.2.1 Extracting the Candidates

Cranelift is a code generator that compiles a WebAssembly (WASM) source program to native machine code by first lowering the WASM module to Cranelift IR, leaving an opportunity for optimizations to be applied on Cranelift IR before it emits the code for target architectures, like x86_64 or ARM. WebAssembly [7] is designed as a portable target for compilation of high-level programming languages like C, C++, Rust, etc. which uses a pipeline such as:

$$\text{C++/Rust} \rightarrow \text{LLVM IR} \rightarrow \text{WASM} \rightarrow \text{Cranelift IR} \rightarrow \text{X86\_64/ARM}$$

The search for transformations begins with harvesting candidates that are extracted at the Cranelift IR level. Given a Cranelift function, the harvester in Cranelift collects all integer sub-expressions and translates them into Souper IR, so they can be given to Souper for superoptimization. As the harvester follows the post-order traversal of the dataflow graph, it continues to translate Cranelift values to Souper values. The traversal is stopped when a value is produced by a memory load or store operation that cannot be mapped to Souper IR as Souper does not support these instructions, the harvester creates a *var* which is an input parameter/variable to the optimization. For any other instruction which can be directly mapped to an equivalent Souper IR, the operands of that Cranelift instruction are also translated to Souper IR and in this way each Cranelift instruction is translated to a Souper instruction.

5.2.2 Synthesizing the Candidates

Souper searches for a right-hand side that refines and is less expensive than a given left-hand side. The refined solution can either be a constant, or any other sequence of instructions. A cost function guides the search as it measures the cost of LHS and RHS. We use Souper's synthesis which is based on enumeration as discussed in Chapter 2.

5.2.2.1 Extending Souper for Cranelift IR

It is not so straight-forward to use Souper directly to superoptimize the candidates because Souper was originally designed for LLVM IR; it encodes each instruction to satisfy the semantics of LLVM IR. However, in our case, we want to use Souper for the candidates that are extracted from Cranelift functions. Thus, the encoding of Souper IR must match the semantics of Cranelift IR.

Cranelift has a slightly different model for handling potentially erroneous operations. In LLVM, an instruction such as shift left, if the shift amount exceeds the number of bits of a given value then it would normally lead to an UB. However, Cranelift avoids the UB here by masking the shift amount with a value one less than the number of bits (i.e., width) to set the higher bits to zero and avoid overflow. The logical shift right and arithmetic shift right use the same idea.

We also extended Souper IR to support some new bitwise instructions that exist in Cranelift. For instance, the *not* operation, *and_not*, *or_not*, and *xor_not*. The fused operation *and_not* encodes the semantics as: $x \ \& \sim y$. We do not support conditional operations such as *select*.

5.2.3 Auto-Generating the Peephole Optimizer

We design a compiler, Cranelift-PeepGen, for generating a peephole optimizer from the set of peepholes that we found using a superoptimizer. The input peepholes generated by Souper are in Souper IR and we want to generate an optimizer that implements these peepholes as functions in Rust, and finally integrate it as a Cranelift module. Cranelift-PeepGen is implemented in Rust and has the following phases.

- **Parsing**: The parser transforms the input Souper instructions on both left and right-hand side of the optimization to Cranelift instructions. The parser along with an instruction builder pass usually performs a 1:1 mapping between Souper and Cranelift IR. However, there are a lot of different instruction opcodes and canonicalization rules in Cranelift IR because of which one Souper instruction is sometimes mapped to multiple Cranelift instructions, or the operands are canonicalized so that the constant value operand is always placed as the last operand of an instruction. For instance, transformation of an add instruction which adds an SSA value to an immediate constant from Souper IR to Cranelift IR is as shown below.

```
add X, c    ⇒   iadd_imm X, c
```

Cranelift supports various fused instructions such as *and_not*, *or_not*, *xor_not* and we discussed an extension to Souper's synthesis to support them in Section 5.2.2.1. Parsing a variant of these fused instructions in Souper IR, where second operand is a constant value, to Cranelift IR, is a little tricky. Souper does not define constant values as an individual instruction. In contrast, the Cranelift IR handles it in two different ways. First, constants are 64-bit immediate integer values. Second, Cranelift has a specific instruction *iconst* to create a scalar integer SSA value with an immediate constant value. The semantics of Cranelift's *and_not* instruction require both operands to be SSA values. Thus, for a Souper *andNot* instruction with an immediate integer constant, the parser in Cranelift-PeepGen first transforms the immediate integer constant to an *iconst* instruction, followed by *band_not* operation that uses the scalar integer SSA value from *iconst*.

```
                        Y = iconst c
Z = andNot c, X    ⇒    Z = band_not Y, X
```

- **Linearization of LHS**: This pass operates on the left-hand side of each peephole optimization because matching is performed for the LHSes, and actions to replace a left-hand side are taken on the right-hand side. The goal of this phase is to transform the sequence of parser-generated Cranelift instructions into a list of linear tree nodes. This phase starts its traversal at the root of the LHS which is the operand of an *infer* instruction and continues in preorder traversal. At the end, it generates a linear preorder traversed tree nodes list which makes matching easier for subsequent passes in Cranelift-PeepGen. For reference, the linear tree nodes list for a peephole in Figure 5.2 is shown in Figure 5.3. During parsing, the operands of each instruction only capture the index number or labels/variable names that were bound to it. This pass generates argument names and appends this information to tree nodes. A mapping from index number to argument names is stored in a map so that it can be used to bind correct argument names on the right-hand side for all such operands or sub-expressions that are reused from the left-hand side. Besides this, the linearization pass also maintains a mapping from ID to name for input variables of an optimization to differentiate between them.

- **Updating RHS and Mapping**: In this phase, the Cranelift instruction on the right-hand side of an optimization is updated with argument names that were generated for the instructions' operands on the left-hand side.

  In addition, the mapping for left and right-hand sides of an optimization is stored in a map that can be accessed once pattern matching on LHS finishes. Later, the replacement by RHSes begins.

- **Building a Merged Prefix Tree for LHS**: This pass generates a merged prefix tree from linear tree nodes of all LHSes. It starts by building a parent root node; we call it a *MAIN* node. It's the node to which all other linear trees will be appended later on. The idea is to visit each left-hand side's linear tree, and determine where the root node of individual tree can be appended. There can be different scenarios:

  (1) the root node of individual linear tree does not exist as the child node of *MAIN*, and in this case we can set root node of this LHS as a child node of *MAIN*.

  (2) the root node of LHS already exists. In this case, we continue to iterate on each node of the linear tree to determine at what node it starts to differ from an existing merged tree structure. The node where we identify the difference between an incoming tree's node and an existing path of nodes in the merged tree, we fork at that node and build a connection between the existing merged tree and an incoming tree.

  (3) there might also be a case in which we already have a path in merged prefix tree that matches exactly with a complete incoming linear tree nodes; we don't add any new nodes.

Figure 5.4 shows merged prefix tree for two left-hand sides of peephole optimizations shown in Figure 5.5. These linear tree nodes contain a lot more information, such as ID number (global value number), attributes of Cranelift IR: InstructionData type, Opcode Type, Width, Argument names bound to an operation, condition code, and so on; but these are skipped in the figure to keep the tree node structure simpler to visualize and understand. In this example, both of the trees share the nodes starting from ‘sub’, followed by an first operand of subtraction operation, i.e., a constant with a value zero node. It forks at the opcode types: ‘add’ and ‘shl’, and thus generates arms of match constructs on Cranelift IR’s *Opcode* shown in Lines 12 to 25 in Figure 5.6.

- **Pattern Matching and Code Generation**: In the prefix tree pattern matching approach, this pass traverses each node of the merged prefix tree starting from the *MAIN* node in a depth first order. As the nodes continue to be visited, this pass generates the Rust implementation for left-hand side of the peepholes. When the traversal reaches the leaf node on a particular path and it finds that the node is actionable, it looks up in the map for the actionable node’s ID and generates the replacement code of right-hand side, as shown in lines 16 and 23, respectively for each peephole optimization. The traversal continues to move breadth-wise to visit the next linear tree. As an example, we have shown an automatically generated code using prefix tree pattern matching approach in Figure 5.6. However, in the baseline approach, the linear tree of each LHS is traversed and code is generated individually for each peephole, thus geenrating individual functions for each peephole in an optimizer. In this approach, we skip building the merged prefix tree and we build this as a reference implementation to verify that the functionality of peepholes is the same in both approaches, and also to study the performance impact of one approach with another. The code generated for an individual function is shown in Figure 5.7. Finally, an automatically generated peephole optimizer is plugged into Cranelift as an optimization pass.

5.2.4 An Example

Let’s take a look at an example to understand the input and output for Cranelift-PeepGen. The input is specified in Souper IR and the details of the syntax and semantics are described in Chapter 2.

The *infer* defines the root node of the directed acyclic dataflow graph of the left-hand side, and *result* is the root node of the right-hand side DAG. The matcher in Cranelift-PeepGen starts matching at $sub\ 0 : i32, \%1$ and traverses the DAG to match all the nodes until it reaches leaf nodes. The variable labeled %0 is defined on the left-hand side and is used by both left and right-hand side. The value computed by the *add* instruction is bound to %1 and this sub-expression is used by the *sub* instruction with label %2. A single *sub* instruction with a label %3 replaces the left-hand side with three instructions, saving two instructions. Cranelift-PeepGen parses the input represented in Souper IR and starts transforming it into Cranelift IR with all attributes of an IR that are required later on to generate the code. Cranelift-PeepGen generates an implementation of this function in the Rust programming language using *match* constructs as shown in Figure 5.7.

The first matching arm is evaluated on an input parameter (*inst*) of the function *superopt_1*, which is the root node of the *LHS*. All possible values of *InstructionData* types of an *inst* are matched. For instance, it matches the *BinaryImm64* instruction data type in this example and it continues to match the opcode and arguments list of that instruction. While the left-hand side is matched, the matcher continues to generate the *match*, nested *if* conditions, and so on. At the end of matching, it replaces the root node i.e., *inst*

```
pos.func.dfg.replace(inst)
```

with the instruction:

```
irsub_imm(arg_1, 1)
```

Now, we will look at another example with two optimizations to understand how the prefix-tree pattern matching approach generates an implementation of both optimizations in one function. The input in Souper IR is in Figure 5.5.

The first optimization is the same as in the previous example, and we added another peephole optimization that has some common IR with the first optimization. Cranelift-PeepGen generates an implementation using prefix-tree pattern matching, shown in Figure 5.6. Both of these optimizations share the root node that is at 'sub' (subtraction) operation with its first operand, a constant with value zero. The second operand of 'sub' at %2 is different in both optimizations. In the first peephole, it is an add operation labeled by %1, and in the second peephole, it is a shift left operation. This is where the prefix-tree approach forks as shown in Line 12 of the code in Figure 5.6. The match construct at this line now works individually for 'IaddImm' and 'IshlImm' operations. Each LHS has a different RHS that translates into code at Lines 16 and 23. The right-hand side on successful match of left-hand side, replaces the root instruction 'inst' with 'irsub_imm' and 'imul_imm' respectively.

5.3 Evaluation

In this section, we will discuss the impact of automatically generated peephole optimizer on compile time of Wasmtime, a runtime system for WebAssembly. We will also show the effect of peepholes on the run time performance of some WebAssembly benchmarks, and the Rustc compiler backend in debug mode.

5.3.1 Experimental Setup

We ran our experiments on a machine with AMD EPYC 7502 32-Core Processor at 3.1 GHz, 64 threads, and 256 GB RAM. We mainly run our experiments for Markdown [12] and Clang [2] that are compiled to WebAssembly. Markdown is a plain text format to create structured documents. This benchmark is implemented in Rust and it compiles a markdown (.md) source program to HTML [11]. Clang is an open-source compiler for the C family of programming languages. It builds one of the front-ends of the LLVM project and compiles C, C++, CUDA and other languages to LLVM IR.

5.3.2 Extraction and Synthesis of Peepholes

We extracted 3,098 potential candidates to be optimized from the Markdown benchmark using the harvester in Cranelift. 994 of them were unique, and we limited the maximum number of synthesized instructions to three. We also set the solver timeout to 15 seconds. Souper's enumerative synthesis successfully returned 109 peephole optimizations. Similarly, we collected 221,968 candidates from Clang. However, there were a huge number of duplicate expressions extracted by the harvester and we ended up running synthesis for 19,149 LHSes only. Souper resulted in synthesizing RHSes for 1,413 candidates.

5.3.3 Compile Time Performance of Wasmtime

We generated the peephole optimizer for Cranelift using Cranelift-PeepGen in two modes: baseline and prefix tree. The details of these approaches are already discussed in the Section 5.2.3. We register an auto-generated peephole optimizer as an optimization pass that operates on Cranelift IR. This pass is registered after a control flow graph is constructed and a very light-weight arithmetic simplification pass (called preopt) is performed. Another reason the superoptimization pass is registered after a preopt pass is because the left-hand sides are harvested from Cranelift IR that is canonicalized after the preopt pass. After peephole optimizations are performed, a dead code elimination pass cleans up redundant code. The remaining passes like, legalization, GVN, are called at the end.

We compile Wasmtime with the new peephole optimization pass embedded in the Cranelift code generator. We record the compile time of the Wasmtime runtime system rather than individually computing the compile time for Cranelift code generator because Wasmtime's compiled binary is later used to build other WASM benchmarks and it builds Cranelift as a library that performs optimizations added to it. We discuss the usage of Wasmtime in next section. We now discuss the performance for Markdown and Clang benchmark.

5.3.3.1 Effect of Peepholes from Clang Benchmark

Cranelift-PeepGen takes 6m23s to generate an optimizer which is 166,302 lines of code for Clang benchmark in baseline mode. In contrast, it takes 9 minutes 33 seconds for code generation of 87,414 lines of code in prefix tree mode.

Figure 5.8 shows the effect of increasing the number of peephole optimizations in both baseline and prefix tree mode on compilation time of Wasmtime. The compile time of Wasmtime is important for developers who develop and maintain this runtime system as each new feature in Wasmtime leads to an increase in development time. Besides this, Wasmtime is statically linked to an external project that uses it. It thus increases the compile time for external projects as well.

Wasmtime is implemented in the Rust programming language and on building this runtime system, the Rust compiler uses LLVM as the default backend. LLVM performs a lot of optimizations in release mode before it generates the code in comparison to debug mode. We recorded performance statistics while building Wasmtime and found out that the overhead is mainly because of LLVM optimizations and code generation.

On compiling Wasmtime in releasemode with our baseline peephole optimizer (100 peepholes) for Cranelift, 56% of time is spent under LLVM. However, the time spent under LLVM increases to 70% in release mode when our baseline peephole optimizer has 1300 peepholes. LLVM spends most of its time in preparing code for the SelectionDAG-based instruction scheduling phase, inlining functions, combining the instructions in LLVM IR. In contrast, the debug mode bypasses optimizations, such as instruction combining at the LLVM IR, inlining of functions, and straight away jumps onto instruction selection and scheduling pass in LLVM to generate the code. This explains that compile time in the debug mode takes less time in comparison to the release mode.

5.3.3.2 Effect of Peepholes from Markdown Benchmark

The auto-generated peephole optimizer for 109 peepholes from the Markdown benchmark comprises 15,016 LOC in baseline mode and 10,260 LOC in prefix tree mode. The time taken to generate an optimizer from Cranelift-PeepGen for baseline and prefix tree mode is 4.97 seconds and 12.33 seconds respectively.

Table 5.1 summarizes the compile time for release and debug build of Wasmtime without and with peephole optimizations in both baseline and prefix tree approach. The effect of the smaller number of peepholes is quite similar to Clang's first few hundred peepholes which showed that debug build time is initially a bit longer than the release build. The increase in compile time by 4 seconds with peephole optimizations is because a larger number of LLVM optimizations are invoked at the backend of Rustc compiler.

5.3.4 Number of Peepholes Triggered

In this section, we compute the total number of times peepholes are triggered o compiling the WebAssembly benchmarks.

- **Markdown Benchmark**: On compiling Markdown benchmark with Wasmtime, 109 peephole optimizations are triggered 904 times. The peephole optimization shown below in Souper IR is triggered the most for 634 times on building markdown benchmark.

```
%0:i32 = var
%1:i32 = ashr %0, 1:i32          %3:i32 = lshr %0, 31:i32
%2:i32 = lshr %1, 31:i32    ⇒    result %3
infer %2
```

  In this example, a 32-bit input variable is arithmetically shifted right by one, and the result of it is logically shifted right by 31. These sequence of two shift right operations can be optimized to one logically shift right operation thus saving one operation.

- **Clang Benchmark**: We compiled another WASM benchmark, Clang, with Wasmtime and enabled peephole optimizations in an increment of 100 to record the total number of peepholes triggered. The first 100 peepholes all sorted by length of instructions were invoked for 3,812 times. The maximum invocations were 13,780 times on enabling all 1300 peephole optimizations as shown in Figure 5.9.

5.3.5 Compile Time Performance of WebAssembly Benchmarks

In the previous section we enabled peephole optimizations in Wasmtime, a runtime system. We discussed the effect of adding optimizations on the compile time of Wasmtime. We further discuss the run-time performance of Wasmtime which determines how effectively it can be used to compile WebAssembly programs like Markdown and Clang. We compiled the source of Markdown and Clang to WASM and pass it as an input to Wasmtime to record the time taken to build these applications with Wasmtime (peephole optimizations enabled), and the time taken to execute the compiled binaries of applications.

5.3.5.1 Markdown Benchmark

We use the Markdown source from an existing open-source test suite and tool, sightglass [10], that compares different implementations of Wasmtime for benchmarking. The mechanism used for measuring the execution time is by adding the *bench::start()* and *bench::end()* APIs around the main functionality of the benchmark. The compile time is computed in both CPU cycles and nanoseconds for the total time taken to compile a WASM file using Wasmtime and to run the compiled binary with an input file passed as an argument. The execution time records the time taken to parse the string input of test markdown file and render to HTML. Each test is run for 500 iterations. The average compile time of the markdown with peephole optimizations enabled in Wasmtime increases by 0.12% in baseline mode and 0.11% in prefix-tree mode of the optimization pass. In this case, we only enabled 109 peephole optimizations that we extracted from Markdown using Cranelift's harvester and synthesized using Souper as discussed in Section 5.3.2. The slight increase in compilation time with both modes as shown in Table 5.2 is because the effective time spent by Cranelift codegen in optimizations has increased. We also found that these optimizations are invoked 904 times in total as discussed in the previous section. The additional overhead caused by the optimizations is 0.05% in baseline mode and 0.03% in the prefix-tree mode. The overhead caused by matching of InstructionData, Opcode, Arguments types increases in prefix-tree mode because the optimizations implemented in this approach heavily use the match constructs. But, the overhead causing by matching (like switch cases) is less in comparison to traversing call graphs with individual functions because matching can allow traversal to skip a lot of paths where it fails to match the intended types of instructions, opcode, etc.

5.3.5.2 Clang Benchmark

For benchmarking Clang compiled to WASM, we enable peephole optimizations in an increment of 100 in each run to record the time taken to compile Clang using Wasmtime. The optimizations are sorted by size and we plot the run-time performance of Wasmtime as shown in Figure 5.10 for both baseline and prefix-tree approach to implement the the peephole optimizer. Figure 5.8.

We used an existing WASM cross-compiled Clang binary for benchmarking and inserted the *bench::start()* and *bench::end()* APIs to record performance. The cross-compiled WASM binary of Clang is disassembled using an open-source tool, wasm2wat from the WebAssembly Binary Toolkit (WABT) [6]. We then inserted a start and end benchmarking APIs of sightglass test suite in the WebAssembly textual format, WAT. We re-assemble the WAT source to WASM using wat2wasm tool from the same WABT open-source repository and benchmark the compile time performance of Wasmtime to record the time taken to build Clang's WASM source with optimizations that are specifically extracted from Clang source, and not Markdown or any other source. The reason we want to adopt optimizations from Clang source for Clang's benchmarking is because we expect to see a lot of patterns being matched and thus optimizations to be invoked at compile time. We satisfied our expectations for total invocations of peepholes as shown in Figure 5.9.

From our analysis, we found that constant folding patterns have already been taken care of by another pass that is performed ahead of the superoptimization pass in Cranelift. However, it still leaves a larger number of opportunities for the superoptimization pass to optimize the patterns it comes across. The peephole pass in particular takes 1.09%

of the total build time in baseline mode as compared to 0.91% in prefix-tree mode. As we discussed, the overhead caused by individual functions in baseline mode in comparison to the match constructs in prefix-tree approach for Markdown benchmark, we observed a similar behavior for Clang as well. Besides that, lowering of optimized Cranelift IR to machine code in prefix-tree mode adds an overhead of 0.36%, whereas the baseline mode adds an overhead of 0.45%. This leads to more code generation time and an overall increase in compile time for baseline mode as compared to prefix-tree mode.

Overall, the compile time increases as more peepholes are invoked. The graph in Figure 5.10 shows that the compile time with 1300 peephole optimizations in prefix-tree mode is same as 500 peepholes in the baseline mode. It means that we can enable more than 50% of the optimizations in prefix-tree mode in comparison to baseline mode at the same compile time cost.

With increasing number of peephole optimizations, the code size of compiled Clang binary also decreases at most by 90 KB when 900 unique peepholes are added and they are invoked for 12,418 times as shown in Figure 5.11. The code size of compiled Clang binary starts to increase slightly when we enable a thousand or more peephole optimizations. This behavior is not expected as there are more peepholes invoked, but code size increases after 1000 peepholes, but we speculate that these peepholes interact with other optimizations in such a way that it leads to an effective increase in size, or there are any phase ordering issues.

5.3.5.3 Blake3 Benchmark

We now evaluate a different benchmark that we have not used earlier to extract the LHSes and synthesize the peepholes. In our previous benchmarking, we used peepholes synthesized from a benchmark and generated a peephole optimization pass for Wasmtime; we later used the Wasmtime runtime system to compile the same application. However, we want to perform an experiment in a different setting here.

We want to evaluate the performance of Wasmtime with optimizations synthesized from Markdown or Clang benchmarks enabled in it to compile Blake3. We want to determine if any optimizations are invoked from peephole optimizer in Cranelift's codegen.

Blake3 is faster than MD5, SHA-1, SHA-2, SHA-3 hash functions and is known to be more secure [1]. We use the WASM cross-compiled BLAKE3 benchmark from sightglass test suite [9]. It is built from the Blake3 Rust crate but with only scalar operations.

On compiling BLAKE3 with Wasmtime, we found that there were 15 unique patterns that matched and peephole optimizations were performed for a total of 46 times. These optimizations in Wasmtime are collected from the Markdown benchmark. However, when we enabled an optimization pass for the peepholes that we synthesized from the Clang benchmark, no pattern matched and hence no peephole optimization was performed.

These optimizations lead to a slight increase in an average compile time to build Blake3.wasm by 0.34% in baseline mode, and 0.31% in prefix-tree mode. However, these optimizations seem to make the code worse because execution time increases by 0.58% as shown in Table 5.2. We are not confident about this behavior but speculate that it could be because of phase-ordering problem which ends up making an IR in a shape that the instruction selector has not been tuned for; or the cost model from Souper is not well-suited for Cranelift IR because we just used an existing cost model from Souper that is designed for LLVM IR and assumed that it should be fitting in well for any other IR as well. However, these speculations need more investigation to prove what might have gone wrong.

The outcome from this experiment also suggests that Souper's synthesized peephole optimizations are not generic to allow more pattern matching coverage. These optimizations are too specific to constant values that it gets very hard to look for any optimization opportunities in an unknown benchmark for which we have not harvested any candidates. This experiment suggests a future improvement in Souper to allow generalization of peephole optimizations and we are aware that some efforts have already started to take place in this direction.

5.3.6 Rust Compiler's Backend Performance

The Rust compiler [5] does not implement its own code generator but instead builds on the LLVM optimizer and code generator, allowing it to provide high-quality optimization and code generation support for many targets.

However, it is not the fastest in the development phase when compiler developers use debug build of LLVM, even when the optimizations are disabled. For these reasons, we need an alternative code generator which can provide fast compile time in the debug mode.

In other words, we need an alternative light-weight compiler.

Cranelift has a potential to improve the compilation time because it was originally designed with the goal of efficiently generating code as opposed to generating an efficient code with lots of optimizations. Cranelift performs efficiently because it is designed to take advantage of multiple cores and make efficient use of the memory.

An open-source project on developing a new codegen backend for Rustc compiler based on Cranelift is already released [8]. As expected, Rustc with Cranelift backend takes less compile time as compared to Rustc with the LLVM backend in debug build mode for many tests.

In this section, we will evaluate the performance of the Cranelift backend with superoptimization based peepholes enabled in it. We collected the peepholes for a cross-compiled WASM benchmark, Markdown, as discussed in Section 5.3.2, and embedded them as a peephole optimization module in Cranelift in two different modes, baseline and prefix-tree.

We wanted to compile a benchmark implemented in Rust using Rustc compiler with different versions of LLVM and Cranelift backend. Markdown benchmark [12] that compiles markdown source to HTML is written in Rust. It seems obvious choice for our benchmarking because peepholes are also collected for a similar benchmark of Markdown (versions may vary. We will determine if these peepholes are really applied to generate an efficient code.

Table 5.3 summarizes the result of average time taken to compile the Markdown benchmark using Rustc compiler with different backends, execution time and code size of the compiled binary. Each experiment is run for 100 iterations.

LLVM is the default backend of Rustc compiler, Cranelift original backend does not include any peepholes, Cranelift baseline comprises of peepholes where each peephole optimization is implemented as an individual function, and Cranelift prefix-tree comprises of peephole optimization pass which is generated by prefix-tree pattern matching approach.

The change columns in Table 5.3 for compile time and runtime indicate the change in time with respect to the reference Cranelift original backend. Though LLVM is the actual reference line because Cranelift has replaced LLVM backend for the Rust compiler in debug build mode. This work builds on top of an existing Cranelift and therefore, we set the original Cranelift without any peepholes as the reference. The Cranelift backend with peepholes in baseline mode increases the compile time by 0.82%, and in prefix-tree mode by 0.41% as compared to the original Cranelift without peepholes. The time spent by a compiler backend with peephole optimizations increases the compile time to build Markdown's source code in Rust programming language.

The time to execute the compiled Markdown binary, which is optimized with peephole optimizations performs efficiently by 7.89%. We also found a small improvement in code size of a compiled Markdown executable that is built with a Cranelift backend with peephole optimizations. The backend with optimizations reduces the code size by 7,456 bytes as compared to the one where we don't have any peepholes in the Cranelift backend. Fast execution time and small code size are a good indication that a light-weight peephole optimization pass with 109 peepholes resulted in generating an efficient code with Cranelift.

5.4 References

[1] *Blake3: A cryptographic hash function.* http://github.com/BLAKE3-team/BLAKE3.

[2] *Clang: A c language family frontend for llvm.* http://clang.llvm.org.

[3] *Dsl-based optimization rules in go compiler.* http://github.com/golang/go/blob/master/src/cmd/compile/internal/ssa/gen/generic.rules.

[4] *Gcc peephole optimizations.* http://github.com/gcc-mirror/gcc/blob/master/gcc/match.pd.

[5] *The rust programming language.* http://github.com/rust-lang/rust.

[6] *Wabt: A webassembly binary toolkit.* http://github.com/WebAssembly/wabt.

[7] *Webassembly.* http://webassembly.org.

[8] bjorn3, *Cranelift codegen backend for rust*, 2019. http://github.com/bjorn3/rustc_codegen_cranelift.

[9] A. Brown, *Blake3 benchmark in sightglass*, 2021. http://github.com/bytecodealliance/sightglass/tree/main/benchmarks-next/blake3-scalar.

[10] BytecodeAlliance, *Sightglass: A benchmarking suite and tooling for wasmtime and cranelift.* http://github.com/bytecodealliance/sightglass.

[11] M. K. de Vries and R. Levien, *pulldown-cmark: A pull parser for commonmark.* http://github.com/bytecodealliance/sightglass/tree/main/benchmarks-next/pulldown-cmark.

[12] N. Fitzgerald, *Commonmark benchmark in sightglass*, 2021. http://github.com/bytecodealliance/sightglass/tree/main/benchmarks-next/pulldown-cmark.

[13] GCC, *An example of gcc peephole optimization.* http://github.com/gcc-mirror/gcc/blob/master/gcc/match.pd#L876-L884.

[14] LLVM, *An example of instcombine transformation.* http://github.com/llvm/llvm-project/blob/main/llvm/lib/Transforms/InstCombine/InstCombineAndOrXor.cpp#L1615-L1647.

[15] N. P. Lopes, D. Menendez, S. Nagarakatte, and J. Regehr, *Provably correct peephole optimizations with Alive*, in Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2015, pp. 22–32.

[16] X. Yang, Y. Chen, E. Eide, and J. Regehr, *Finding and understanding bugs in C compilers*, in Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '11, 2011, pp. 283–294.

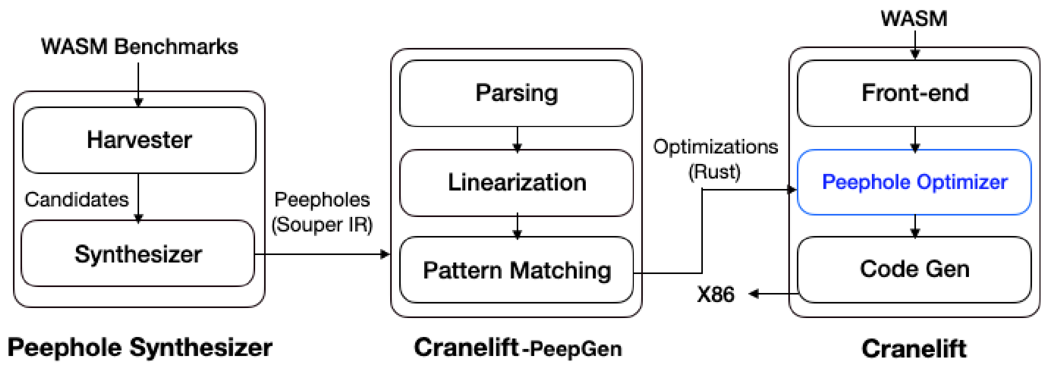


Figure 5.1. Cranelift-PeepGen: An automatic peephole optimization generator for Cranelift.

```
%0:i32 = var
%1:i32 = add %0, -1:i32
%2:i32 = sub 0:i32, %1
infer %2
->
%3:i32 = sub 1:i32, %0
result %3
```

Figure 5.2. Peephole optimization in Souper IR.

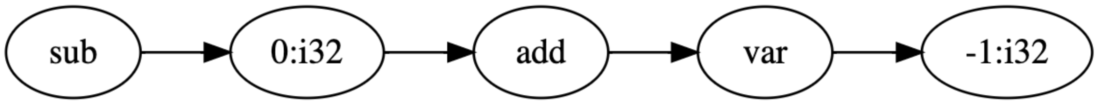


Figure 5.3. Tree nodes list generated by the linearization pass.

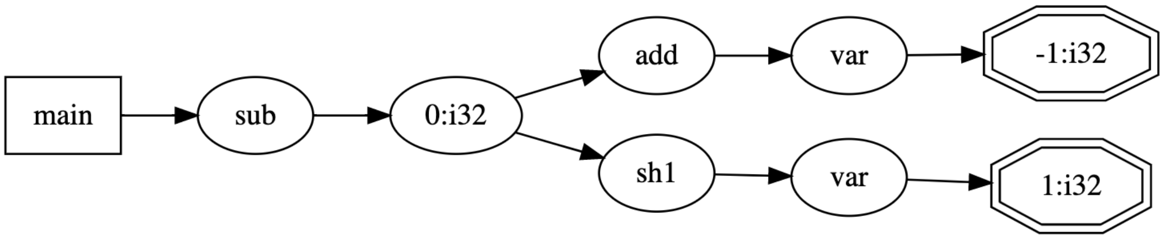
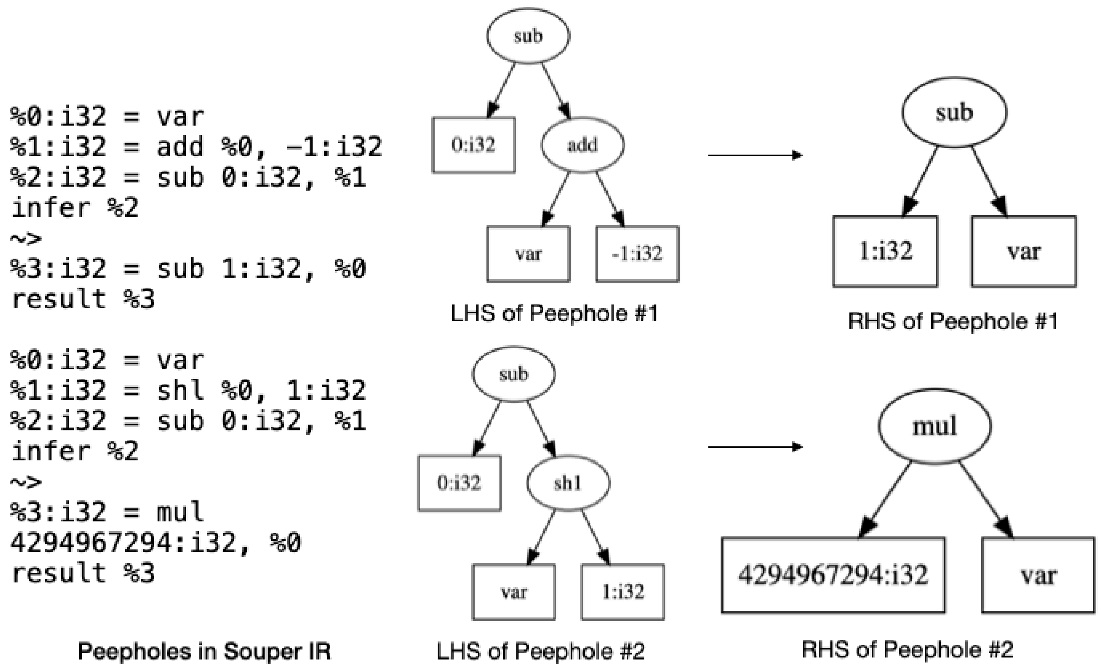


Figure 5.4. Merged prefix tree for multiple peepholes.



```
%0:i32 = var
%1:i32 = add %0, -1:i32
%2:i32 = sub 0:i32, %1
infer %2
~>
%3:i32 = sub 1:i32, %0
result %3
```

```
%0:i32 = var
%1:i32 = shl %0, 1:i32
%2:i32 = sub 0:i32, %1
infer %2
~>
%3:i32 = mul
4294967294:i32, %0
result %3
```

Figure 5.5. Multiple optimizations input in Souper IR for Cranelift-PeepGen.

```
fn superopt_1(pos: &mut FuncCursor, inst: Inst) {
  match pos.func.dfg[inst] {
    InstructionData::BinaryImm64 { opcode, arg, imm } => {
      let arg_0 = arg;
      let rhs_1: i64 = imm.into();
      match opcode {
        Opcode::IrsubImm => match pos.func.dfg.value_def(arg_0) {
          ValueDef::Result(arg_ty, _) => match pos.func.dfg[arg_ty] {
            InstructionData::BinaryImm64 { opcode, arg, imm } => {
              let arg_1 = arg;
              let rhs_2: i64 = imm.into();
                match opcode {
                  Opcode::IaddImm => {
                    if rhs_2 == -1 {
                      if rhs_1 == 0 {
                        pos.func.dfg.replace(inst).irsub_imm(arg_1, 1);
                      }
                    }
                  },
                  Opcode::IshlImm => {
                    if rhs_2 == 1 {
                      if rhs_1 == 0 {
                        pos.func.dfg.replace(inst).imul_imm(arg_1, 4294967294);
                      }
                    }
                  },
                  _ => {}
                }
                ...
}
```

Figure 5.6. An optimization implementation generated with prefix-tree pattern matching approach of Cranelift-PeepGen.

```
fn superopt_1(pos: &mut FuncCursor, inst: Inst) {
  match pos.func.dfg[inst] {
    InstructionData::BinaryImm64 { opcode, arg, imm } => {
      let arg_0 = arg;
      let rhs_1 : i64 = imm.into();
      match opcode {
        Opcode::IrsubImm => {
          match pos.func.dfg.value_def(arg_0) {
            ValueDef::Result(arg_ty, _) => {
              match pos.func.dfg[arg_ty] {
                InstructionData::BinaryImm64 { opcode, arg, imm } => {
                  let arg_1 = arg;
                  let rhs_2 : i64 = imm.into();
                  match opcode {
                    Opcode::IaddImm => {
                      if rhs_2 == -1 {
                        if rhs_1 == 0 {
                          pos.func.dfg.replace(inst).irsub_imm(arg_1, 1);
                        }
                      }
                    },
                    _ => {},
                  }
                ...
}
```

Figure 5.7. A single optimization implementation generated with Cranelift-PeepGen.

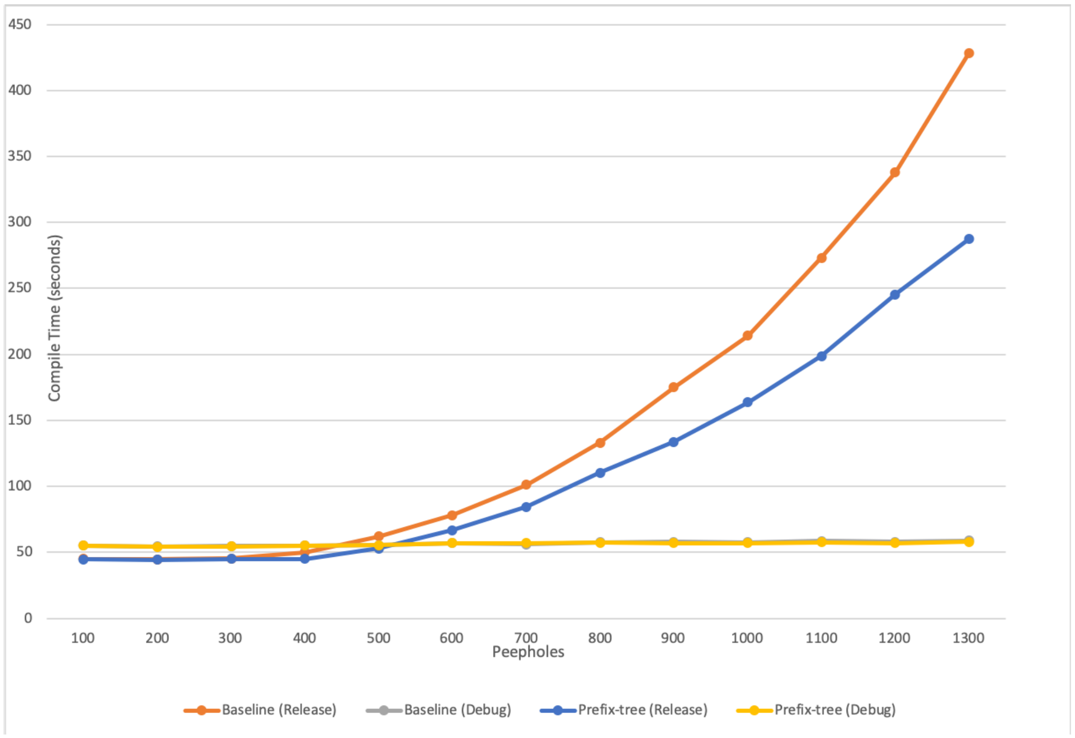


Figure 5.8. Effect of increasing number of peephole optimizations from Clang benchmark on compile time of Wasmtime.

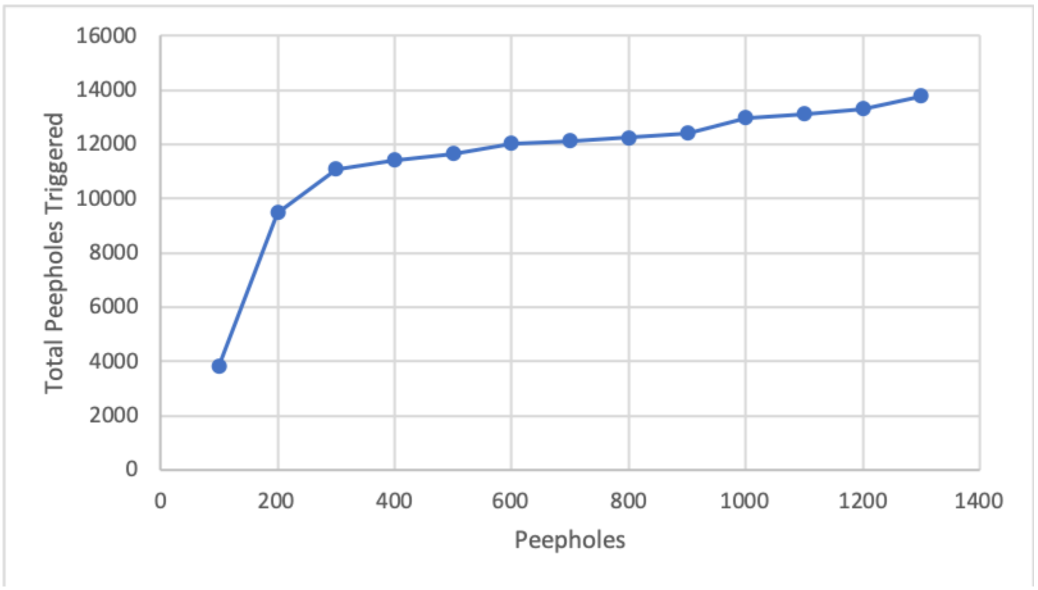


Figure 5.9. Total number of peepholes invocations on compiling Clang.wasm benchmark.

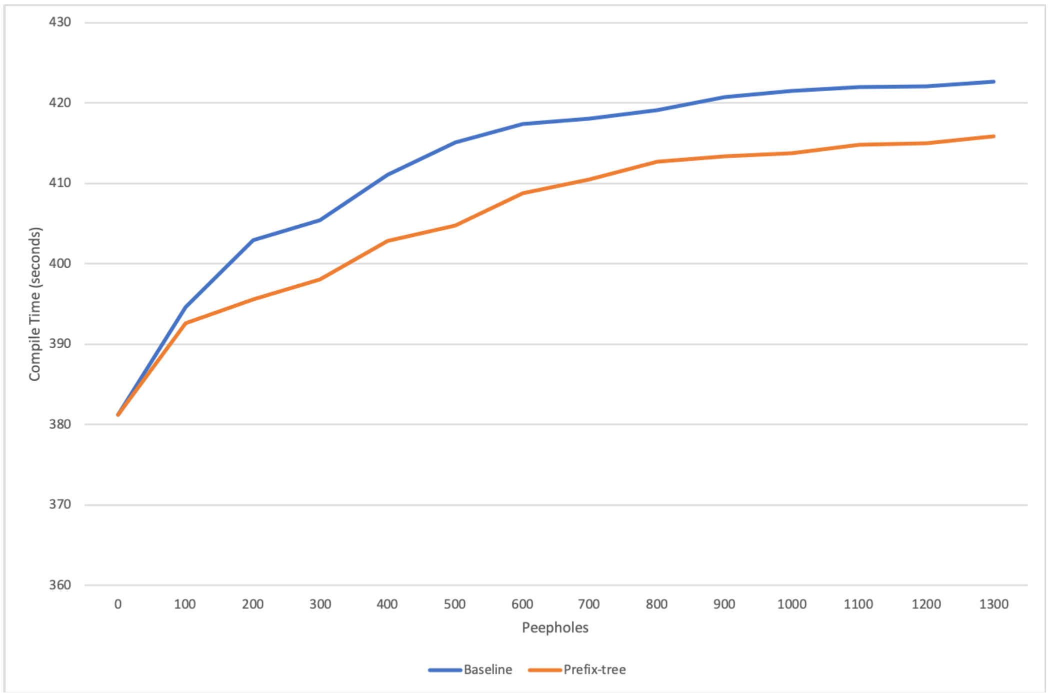


Figure 5.10. Effect of an increasing number of optimizations on the performance of Wasmtime to compile Clang.wasm benchmark in baseline and prefix-tree of peephole optimizer.

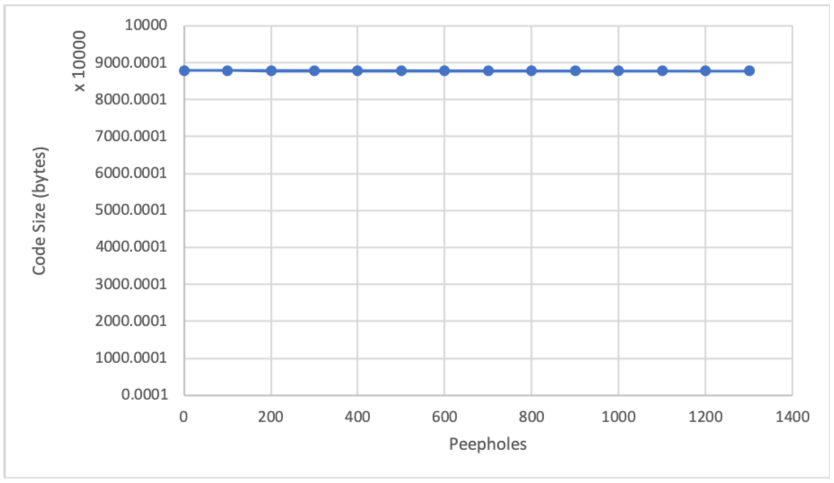


Figure 5.11. Effect of an increasing number of optimizations on the code size of compiled Clang binary.

Table 5.1. Compile time of Wasmtime without and with peephole optimizations from Markdown benchmark in release and debug build mode.

Mode	Release Build	Debug Build	Size (MB)
Without Peepholes	44.61s	55.45s	9.93
With Peepholes (Baseline)	47.41s	59.61s	10.04
With Peepholes (Prefix-tree)	46.86s	58.91s	9.98

Table 5.2. Performance of WebAssembly benchmarks, Markdown.wasm and Blake3.wasm, with and without peephole optimizations.

Benchmark	Metric (CPU cycles)	Without Peepholes	W/ Peepholes (Baseline)	Change	W/ Peepholes (Prefix-tree)	Change
Markdown	Compile Time	905268620	906356002	+0.12%	906246792	+0.11%
	Execution Time	19367812	19002583	-1.88%	19002314	-1.89%
Blake3	Compile Time	417516377	418933713	+0.34%	418817395	+0.31%
	Execution Time	568600	571930	+0.58%	571894	+0.58%

Table 5.3. Performance parameters to build markdown benchmark with different compiler backends in debug mode.

Backend	Avg. Compile Time	Change Compile time	Average Run-time	Change Run-time	Code Size (bytes)
Cranelift original	2.43s	Reference	0.038s	Reference	6536576
Cranelift baseline	2.45s	+0.82%	0.035s	-7.89%	6529120
Cranelift prefix-tree	2.44s	+0.41%	0.035s	-7.89%	6529109

CHAPTER 6

RELATED WORK

This chapter discusses the related work in the area of superoptimization in Section 6.1. Section 6.2 covers the research work done towards testing of static analyses. The literature on pattern matching techniques for code generation is described in Section 6.3.

6.1 Superoptimizers

Most previous superoptimizers were intended for offline use and it was not practical to use them online. The earliest superoptimizer by Fraser [10] in 1979 demonstrated the idea of superoptimization to extract instruction sequences from a program, finding a cheaper solution, and using an equivalence checker to verify each solution. Given an assembly language program and a symbolic machine description, Fraser's portable peephole optimizer worked in multiple passes to perform optimization. In the first pass, it studied the effect of each instruction. Second, it worked on reducing all adjacent pairs of instructions by a single instruction. Third, it replaced an expensive instruction with an equivalent cheaper instruction. Fraser's superoptimizer was machine-independent and was easily retargeted by other machine descriptions. This work is particularly impressive because theorem provers did not exist when it was developed.

The term Superoptimizer was coined by Massalin in 1987 [21]. Her tool was unsound and offline. It extracted the left-hand sides (i.e., ASM instructions sequence to superoptimize) manually. It used enumeration to list all possible right-hand sides of a given potential optimization, and testing to filter out inequivalent right-hand sides.

The two superoptimizers that inspired the design of Souper are by Sands [25] and Bansal [1]. The superoptimizer by Sands pointed out many optimization opportunities in LLVM that had not yet been implemented in 2010-2011. Sands' superoptimizer extracted directed acyclic graphs (DAGs) of LLVM IR during compilation and then optimized them. This superoptimizer was offline and unsound because it used testing for equivalence checking.

However, the results of this superoptimizer did not lead to any miscompilations because this tool only gave suggestions to LLVM developers for missing optimizations. Most of the superoptimizers were offline, but by Bansal et al. [1] designed a sound and online superoptimizer. It used a SAT solver for equivalence checking. Additionally, it cached its results in a database to achieve good performance and worked efficiently in an online mode. This tool used enumeration to discover right-hand sides and had no knowledge about dataflow facts, but it could deal with memory accesses. Bansal's superoptimizer showed that they can optimize small functions that cannot be optimized by GCC. Their optimized results were equivalent to -O2 optimization level, but they did not superoptimize any real benchmarks.

Another closely related superoptimizer that operates on a compiler intermediate representation (as opposed to working on the machine instructions, which is what most superoptimizers do), and also interacts with dataflow analyses is OPTGEN [4]. It generates all optimizations up to a cost limit, rather than extracting code sequences from programs. It can synthesize optimizations containing symbolic constants. Our work [28] supports a broader variety of analyses and, additionally, Optgen has no support for spotting imprecisions in the compiler's dataflow analyses. In fact, we are unaware of any previous effort, in research or in practice, that automatically searches for imprecisions in the dataflow analyses performed by a real compiler.

Recently, we have seen a significant progress in the research of superoptimizers. All the superoptimizers that we have discussed so far either use enumerative or solver-based synthesis search procedures, but STOKE [26, 27] uses randomized search procedure to find an equivalent program (right-hand sides). This approach does not guarantee to find an optimal equivalent program. However, the randomized search space has proven to be much better at superoptimizing larger programs. It performs transformations by either replacing an opcode by a random opcode; replacing an operand by a random operand; switching any two randomly selected instructions; or inserting or deleting any random instruction.

GreenThumb [24] is a generic superoptimizer that can be easily reused for multiple ISAs. The search procedure in this superoptimizer makes uses of three state-of-the-art methods: enumerative, randomized, solver-based synthesis. It is used to superoptimize LLVM IR.

6.2 Testing Static Analyses

In this section we focus on methods for testing static analyses. Bugariu et al. [5] test numerical abstract domains by creating a collection of representative abstract values, applying a sequence of abstract transfer functions, and then testing the results using a collection of properties that check for unsoundness, imprecision, or failure to converge. They found bugs in libraries such as Apron. This work has the same goals as our work [28], but uses a very different test oracle: theirs is based on high-level properties and ours is based on computing sound and precise results using algorithms that call out to an SMT solver.

Randomized differential testing has been used to find soundness and precision issues in static analyzers. Cuoq et al. [7] instrumented an analyzer with assertions to compare the inferred values with the values obtained from concrete execution. Klinger et al. [15] used differential testing to evaluate both the soundness and precision of program analyzers. Their work tested six different program analyzers and found defects in four of them. Since it is difficult to randomly generate programs from scratch that can find all precision and soundness issues in multiple program analyzers, this technique uses a given set of seed programs to automatically generate program analysis benchmarks. Their tool, α-diff, selects a seed program that has no assertions in it. It then selects a program point to instrument an assertion, and tests a property of interest for soundness and precision. It iterates to select a new location and instrument different assertions for a single seed program until the synthesized checks for a program have depleted. Each program analyzer is safe (if the assertion passes), unsafe (if the assertion fails) or unknown (if it cannot reach a conclusion). To detect potential soundness issues, α-diff checks if most of the analyzers return unsafe, but a few of the analyzers return safe. In contrast, a precision issue is detected if a few analyzers return unsafe, but a majority of them return safe. Klinger et al.'s goals are the same as ours [28], but we avoided differential testing for two reasons. First, there do not exist any other implementations of most of the static analyses for LLVM IR that we target. Second, our solver-based techniques are maximally precise, avoiding the limitation that differential testing cannot, in general, find imprecisions in the most precise tool being tested.

Midtgaard and Møller [22] used ideas from QuickCheck [6] to test properties of static analyses using a domain-specific language to specify properties to be tested.

Madsen et al. [20] verify the correctness and safety of abstract domains implemented in static analysis tools. Their technique uses SMT solvers and is specifically designed for Flix, a functional and logic programming language designed for implementing static analyses. This technique allows a user to define a lattice and transfer functions in a functional language that is intended to be verified. For this to work, they extended Flix to support annotations and laws to specify properties of abstract domains to be tested. It verifies the correctness of user-defined transfer functions by ensuring that results of any operation applied on lattice elements do not under-approximate. However, it assumes that constraints defined in a logical language part are safe and sound by default because these cannot be verified and it is the responsibility of the analysis designer to define the constraints correctly. They verify abstract domains using an SMT solver and also support testing based on QuickCheck [6, 22]. This work is good and useful for testing soundness of user-defined lattice and its properties. However, our work presented in Chapter 4 that designs solver-based algorithms tests both soundness and precision of heavily used and complicated LLVM's dataflow analyses. With this technique, we have reported imprecisions in LLVM's static analyses in the past that have been fixed already, and now we have an interesting list of imprecisions that needs to be addressed in LLVM.

Rhodium by Lerner et al. [18] uses DSL and SMT solver to prove correctness of transformations and dataflow analyses. It uses local propagation rules to implement flow functions that can be automatically verified. These rules define how the dataflow facts propagate across CFG nodes. The verified flow functions are later used to derive different kinds of dataflow analyses: flow-sensitive, flow-insensitive, interprocedural.

Most recently, Brown et al. [3] presented a system, VeRA, to verify an integer range analysis in the Firefox browser just-in-time compiler. While most of the prior systems aim to finding soundness bugs in optimizations [19, 23, 17, 16] or static analyses [18, 28, 15] of a compiler, this work guarantees absence of bugs as well. The system, VeRA, provides a subset of C++ for writing verified range analysis functions for browser developers, a DSL to define the semantics of each operations applied on an integer range to verify the analysis by properties. It translates the VeRA C++ to SMT queries to verify the correctness. This tool detected a soundness bug in Firefox browser that existed for the last six years.

6.3 Pattern Matching

The earliest implementations of peephole optimizations were written by hand, and this is still largely true. For example, LLVM's peephole optimizer has a large number of peephole optimization patterns implemented manually.

Recognizing the need for automation, Fraser [10] in 1979 was the first to design a peephole optimizer that was automatically generated from a formal description. The formal description specified the semantics of each instruction and was encoded in the form of machine description. These machine descriptions described the effects of each instruction on registers of the target machine, and thus named register transfer list (RTL). Fraser's automatic peephole optimizer (named, PO) looked at RTL of adjacent instructions to find out if the pair of RTLs can be replaced by a single RTL. PO implemented a string-based pattern matcher to match pair of RTLs and find out their string replacement.

Fraser's peephole optimizer (PO) had some limitations. It supported only a pair of instructions at a time, and these instructions belonged to a same basic block.

- **String-based pattern matching:** Much research has been dedicated to improve the automated generation of peephole optimizers using string-based pattern matching. Fraser's peephole optimizer's [10] limitation was removed by Davidson-Fraser approach [8] as it considered triplets of instructions at a time. Later, more efforts were put to eliminate the need of fixed window size [12, 14]. To improve the code quality, more research contributions were made towards simplifying the RTLs and machine descriptions of each instruction [31, 9, 11].

- **Tree-based pattern matching:** Wasilew [29] and Weingart [30] are the first ones to use tree-based pattern matching for automatic code generation. Wasilew's approach represented the program in postfix notation. It started pattern matching at a leaf node of the tree (i.e., starting at last line of a program and traversing backwards) and matched all patterns stored in a table to find out where the match was successful. If the match succeeded, it continued to match the remaining subtree of a given program until no match was found, to find out the largest pattern that matched successfully. And, finally, transformation or replacement was performed with the result of a pattern. Wasilew's approach had an extensive support of each instruction in the form of a pattern. However, this approach was too expensive to be used in real compilers because of high compilation time.

In comparison to Wasilew's approach, Weingart's approach [30] was designed for single tree of patterns. This approach was believed to be more useful for compilers. However, this did not work well for PDP-11. Second, the design assumed that there is at least one pattern for each instruction.

Johnson's pattern matching approach was simple and straight-forward and used in portable C compiler (PCC) [13]. The pattern matching algorithm compared the node of a given program tree with a root node of each pattern. If the match was successful, it continued to compare the remaining subtree of a program until leaf nodes in a pattern were reached. However, a few contributions were made towards reducing the complexity by minimizing the redundant checks.

All these approaches by Wasilew, Weingart, and Johnson were designed for instruction selection. Thus, their goal was to cover maximum tree [2]. Production compilers like GCC, Go, PCC use rewriting rules and a domain specific language or an intermediate representation to automatically implement peephole optimizations. These compilers have been using a sequential tree-based pattern matching approach, where each program tree is matched individually to generate a successful match. This method adds a lot of compilation time cost in general because each rewrite rule maps to a different function.

6.4 References

[1] S. Bansal and A. Aiken, *Automatic generation of peephole superoptimizers*, in Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '06, 2006, pp. 394–403.

[2] G. H. Blindell et al., *Instruction Selection*, Springer, 2016.

[3] F. Brown, J. Renner, A. Notzli, S. Lerner, H. Shacham, and D. Stefan, *Towards a verified range analysis for javascript jits*, in Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, 2020, pp. 135–150.

[4] S. Buchwald, *Optgen: A generator for local optimizations*, in 24th International Conference on Compiler Construction (CC 2015), B. Franke, ed., London, UK, Apr. 2015, pp. 171–189.

[5] A. Bugariu, V. Wustholz, M. Christakis, and P. Muller, *Automatically testing implementations of numerical abstract domains*, in Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ACM, 2018, pp. 768–778.

[6] K. Claessen and J. Hughes, *Quickcheck: A lightweight tool for random testing of haskell programs*, ACM SIGPLAN Notices, 46 (2011), pp. 53–64.

[7] P. Cuoq, B. Monate, A. Pacalet, V. Prevosto, J. Regehr, B. Yakobowski, and X. Yang, *Testing static analyzers with randomly generated programs*, in NASA Formal Methods Symposium, Springer, 2012, pp. 120–125.

[8] J. W. Davidson and C. W. Fraser, *Code selection through object code optimization*, ACM Transactions on Programming Languages and Systems (TOPLAS), 6 (1984), pp. 505–526.

[9] J. Dias and N. Ramsey, *Converting intermediate code to assembly code using declarative machine descriptions*, in International Conference on Compiler Construction, Springer, 2006, pp. 217–231.

[10] C. W. Fraser, *A compact, machine-independent peephole optimizer*, in Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL '79, 1979, pp. 1–6.

[11] C. W. Fraser and A. L. Wendt, *Automatic generation of fast optimizing code generators*, vol. 23, ACM, 1988.

[12] R. Giegerich, *A formal framework for the derivation of machine-specific optimizers*, ACM Transactions on Programming Languages and Systems (TOPLAS), 5 (1983), pp. 478–498.

[13] S. C. Johnson, *A portable compiler: theory and practice*, in Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, ACM, 1978, pp. 97–104.

[14] R. R. Kessler, *Peep: an architectural description driven peephole optimizer*, in ACM SIGPLAN Notices, vol. 19, ACM, 1984, pp. 106–110.

[15] C. Klinger, M. Christakis, and V. Wustholz, *Differentially testing soundness and precision of program analyzers*, arXiv preprint arXiv:1812.05033, (2018).

[16] S. Kundu, Z. Tatlock, and S. Lerner, *Proving optimizations correct using parameterized program equivalence*, ACM Sigplan Notices, 44 (2009), pp. 327–337.

[17] S. Lerner, T. Millstein, and C. Chambers, *Automatically proving the correctness of compiler optimizations*, in Proceedings of the ACM SIGPLAN 2003 conference on Programming Language Design and Implementation, 2003, pp. 220–231.

[18] S. Lerner, T. Millstein, E. Rice, and C. Chambers, *Automated soundness proofs for dataflow analyses and transformations via local rules*, ACM SIGPLAN Notices, 40 (2005), pp. 364–377.

[19] N. P. Lopes, D. Menendez, S. Nagarakatte, and J. Regehr, *Provably correct peephole optimizations with Alive*, in Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2015, pp. 22–32.

[20] M. Madsen and O. Lhotak, *Safe and sound program analysis with flix*, in Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ACM, 2018, pp. 38–48.

[21] H. Massalin, *Superoptimizer: A look at the smallest program*, in Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems, ASPLOS '87, 1987, pp. 122–126.

[22] J. Midtgaard and A. Mller, *Quickchecking static analysis properties*, Software Testing, Verification and Reliability, 27 (2017), p. e1640.

[23] G. C. Necula, *Translation validation for an optimizing compiler*, in Proceedings of the ACM SIGPLAN 2000 conference on Programming Language Design and Implementation, 2000, pp. 83–94.

[24] P. M. Phothilimthana, A. Thakur, R. Bodik, and D. Dhurjati, *Greenthumb: Superoptimizer construction framework*, in Proceedings of the 25th International Conference on Compiler Construction, CC 2016, 2016, pp. 261–262.

[25] D. Sands, *Super-optimizing LLVM IR*, Nov. 2011. Presentation at the 2011 LLVM Developers' Meeting.

[26] E. Schkufza, R. Sharma, and A. Aiken, *Stochastic superoptimization*, in Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, 2013, pp. 305–316.

[27] ——, *Stochastic optimization of floating-point programs with tunable precision*, in Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, 2014, pp. 53–64.

[28] J. Taneja, Z. Liu, and J. Regehr, *Testing static analyses for precision and soundness*, in Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization, 2020, pp. 81–93.

[29] S. G. Wasilew, *A compiler writing system with optimization capabilities for complex object order structures*, (1972).

[30] S. W. Weingart, *An efficient and systematic method of compiler code-generation.*, (1974).

[31] A. L. Wendt, *Fast code generation using automatically-generated decision trees*, vol. 25, ACM, 1990.

CHAPTER 7

FUTURE WORK

This chapter focuses on discussing some avenues that could be explored to continue my efforts to improve compiler construction using formal methods-based techniques.

7.1 Generalized Framework for Testing Static Analyses

Chapter 4 presented solver-based algorithms to test LLVM's static analyses for precision and soundness. The testing was limited to a small number of abstract domains such as integer ranges, known bits, and demanded bits. An obvious extension to this work would be to write more solver-based algorithms to test more complicated abstract domains. For instance, analyses like live variable analysis and available expression analysis operate on a variable and an expression, respectively, as compared to the already explored known bits analysis that operates at a bit level. However, the entities in these analyses can be represented in the form of a bit vector. For instance, each variable is represented as one bit in a bit vector. Similar algorithms could be implemented to test their soundness and precision. The goal is to verify these static analyses as much as possible so that we can completely trust them, since they drive many optimizations in a compiler.

As a long term goal, it would be interesting to verify more compilers for sound and precise dataflow analyses. In order to achieve this goal, we cannot always rely on writing separate algorithms for each analysis of a different compiler, or writing new DSLs for every new compiler under test. We need to work towards designing a generalized framework to test static analyses of different compilers such as LLVM, GCC, MSVC, and so on. We want this framework to be easily customized and pluggable so that semantics of different compilers' IRs under test can be easily turned on and off. The good part about testing dataflow analyses is that their properties are already well defined in the literature. We can design algorithms and validate their correctness and precision on the basis of these properties. For instance, the well defined properties of bitvector dataflow analyses helped us to prove that dataflow facts computed by our known bits and demanded bits algorithms are most precise. In the same way, we are expecting to learn different shapes of lattices and their properties to assist us in the design of precise algorithms.

7.2 Synthesizing Transfer Functions

The generalized frameworks to test static analyses can help in identifying the soundness bugs or imprecisions in the analyses. However, it cannot automatically fix existing analyses and rather only provide suggestions to analyses' developers. Another idea to explore in this direction is to synthesize precise transfer functions instead of writing them by hand for each and every operation of a specific abstract domain. For instance, if we know the semantics of LLVM IR, and most precise dataflow results computed by SMT solver-based algorithms that are presented in Chapter 4, we could use these results as an oracle of expressions and dataflow facts to learn different rules and properties of transfer functions and synthesize the most precise flow functions of an analysis.

As an example, older versions of LLVM lacked a precise shift-left transfer function for the known bits abstract domain for one of the simple and most frequently occurring cases; a constant value shifted left by an unknown shift amount. Our experiment on testing static analyses collected ∼270,000 DAGs and we computed dataflow results for all of them, stored it in an oracle. We can use this as a training set to separate out the DAGs on the basis of the type of root node. Now, these examples can guide the synthesis algorithm to derive the precise and generalized transfer function. We can visualize this idea as given a set of concrete values, we want to generate an abstraction from the set.

7.3 Generalization of Optimizations

The experiment performed for Blake3 benchmark in Section 5.3.5.3 found that there were only three peepholes that were invoked when we compiled the benchmark with Wasmtime and peephole optimizations enabled synthesized from Souper enabled in it. We cannot always rely on harvesting the candidates and synthesizing peepholes for a benchmark under test. We rather want to collect a set of generalized peephole optimizations, generate an automatic peephole optimizer, and use it to optimize any programs. My experiment confirmed that Souper lacks the ability to synthesize generalized optimizations. A tool, Alive-Infer [1], has done some prior work to derive the weakest preconditions that is a first step towards generalization. But, the preconditions derived by this tool are so complicated that they cannot be used easily. We can make use of the dataflow facts expressed along each instruction in Souper and use them as a precondition instead. We can then synthesize the weakest dataflow fact and use it as a precondition.

7.4 More Superoptimization Opportunities

How would one choose where to superoptimize in a given pipeline? For instance, a superoptimizer could be implemented at three different levels in the LLVM pipeline, shown in Figure 2.1. It can be at the source programming language level, LLVM IR, or at the target instructions like x86 or ARM. So, it depends on the type of optimizations we might be interested in or it might be driven by the idea of covering a winder audience of users that can take benefit of this technique. Another question is: Is superoptimization at one level enough? The answer is probably no, rather we want to perform superoptimization at all possible levels to generate a highly efficient (optimized) code. However, the superoptimization technique is slow as it queries a SMT solver quite a lot. For the systems that do not care about high compilation time, can take the benefits of online superoptimization.

There are so many programming languages, like Rust and Swift, that have a high-level IR with programming language features in it, and both of these programming languages use LLVM as a backend and rely on it to perform optimizations. However, it is hard to capture optimization opportunities at the PL level, once the high-level PL is translated to an IR. One of the future extensions of Souper can be to extend its design for high-level programming languages, or design a new superoptimizer completely. For this, we will first start by writing semantics of the language itself. These solver-assisted explorations can help us in understanding the future directions of compiler construction.

7.5 References

[1] D. Menendez and S. Nagarakatte, *Alive-infer: Data-driven precondition inference for peephole optimizations in llvm*, in Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2017, pp. 49–63.

CHAPTER 8

CONCLUSION

This chapter concludes my dissertation by summarizing its contributions. The work established in this dissertation is done on different components of two different production compilers, LLVM and Cranelift. The thesis that this dissertation has supported is that *Techniques based on formal methods can be used to make production-grade compilers more correct and more effective.*

This dissertation has contributed a collection of novel solver-based algorithms for computing maximally precise dataflow analysis results. These algorithms have found numerous imprecisions in LLVM's static analyses and many of them have already been fixed. The algorithms have not found any new soundness bugs. However, they have demonstrated that they can find such bugs if they existed, by reintroducing previously fixed bugs.

This dissertation has also presented an automatic peephole optimizer based on a superoptimization technique for the Cranelift compiler. These optimizations guarantee correctness as they are synthesized by Souper that uses an SMT solver to verify their correctness. Cranelift with peephole optimizations enabled, when used as a backend for the Rust compiler, has shown a speedup by saving execution time of an application as compared to Cranelift without any peepholes.

The traditional design of compilers is to implement front-end; middle-end comprising of dataflow analyses, optimizations; and backend. Compiler developers should try to make some abstractions and lift them in form of a specification (i.e., DSL). This can allow developers to individually focus on writing formal semantics of the DSL to verify their correctness, and others to contribute in synthesizing these specifications. The developers can finally implement a compiler-compiler to generate the phases of compiler. These ideas can lead to correctness and more effectiveness in compilers.