Bachelor Thesis

Superoptimization for RISC-V using Program Synthesis

Superoptimizer für RISC-V mittels Programmsynthese

Lisa Hofert

Matriculation Number: 4719482

Email: hofertlisa@gmail.com

Writing Period: 21. 08. 2023 – 21. 11. 2023

Examiner: Prof. Dr. Peter Thiemann

Albert-Ludwigs-Universität Freiburg

Technische Fakultät

Institut für Informatik

Lehrstuhl für Programmiersprachen

21. November 2023

Erklärung

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen/Hilfsmittel verwendet habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Darüber hinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt wurde.

Ihringen, 21.11.2023

Ort, Datum

Unterschrift

Abstract

Superoptimization allows us to automatically generate optimal code sequences that would be hard for a programmer to find by hand. This can be useful for problems such as finding peephole optimizations.

In this paper, we explore the superoptimization of RISC-V code for arithmetic expressions. We discuss some existing tools for superoptimizing assembly in general, explain some of the theory behind synthesis and SMT-Solvers, and take a more in-depth look at CEGIS-based synthesis approaches.

We also present an framework for working with RISC-V assembly in Python, and an implementation of a superoptimizer using CEGIS-based synthesis that can compute the shortest possible sequence of RISC-V assembly instructions equivalent to a given arithmetic expression. Our approach is limited to finding instructions of a length of three or less. At the end of the paper we present a brief benchmark for our implementation and give an outlook on possible future extensions of our work.

Zusammenfassung

Superoptimierung ist ein Prozess, der es uns ermöglicht, automatisch optimale Code-Sequenzen zu generieren. Dies ist z.B. für die Verbesserung von Peephole-Optimizations in Kompilern hilfreich, da das Finden von optimalen Sequenzen von Hand oftmals sehr schwer ist.

In dieser Arbeit betrachten wir die Superoptimierung von RISC-V Assembler code für arithmetische Ausdrücke. Wir charakterisieren bestehende Programme zur Superoptimierung, erklären die Konzepte hinter Programmsynthese und SMT-Solvern, und fokussieren uns dann speziell auf Programmsynthese mithilfe von CEGIS.

Wir stellen eine Implementierung eines Frameworks zur Arbeit mit RISC-V in Python vor, und nutzen dieses Framework, um einen CEGIS-basierten Superoptimierer für das Generieren von optimalen RISC-V Code-Sequenzen für einen gegebenen arithmetischen Ausdruck zu schreiben. Dieser Superoptimierer kann Sequenzen mit einer Länge von drei oder weniger finden. Am Ende dieser Arbeit analysieren wir kurz die Laufzeit unserer Implementierung und geben einen Ausblick auf mögliche zukünftige Erweiterungen.

Contents

1 Introduction

In recent years, the topic of AI has become increasingly pervasive in many areas of life [22]. Public interest has grown even further with the release of ChatGPT, a large-scale language model (LLM), at the end of 2022. LLMs aim to solve a large variety of tasks with natural language as their input.

The question arises: What can efficiently be achieved using these systems?

One area in which LLMs might be helpful is software engineering. As noted by Haoye Tian et al. in their 2023 paper about ChatGPT as a Programming Assistant [23], "[it] has gained considerable attention for its potential as a bot for discussing source code, suggesting changes, providing descriptions, and generating code" (p.1). In their analysis of the code generation capabilities of LLMs, they find that the AIs perform well with common problems, even though they start to struggle when the difficulty of the tasks increases.

Automating program development through synthesis has long been a goal in software development [15]. LLMs seem to make this process easy and provide a useful starting point. But there may be drawbacks to using them as a method of synthesis:

According to the results of Haoye Tian et al., the time complexity achieved by the synthesized solutions is typically good, but not at the theoretical ideal [23]. The authors of a different paper also note that over time, ChatGPT is getting worse at solving problems involving program synthesis, and results by the bot can not always be relied on [8].

From this, we conclude that if the goal is to get reliable solutions specifically for program synthesis, there still is merit in exploring specialized algorithms.

For synthesis, there are many areas one could focus on. This work aims to look into the issue of creating optimal code, using the example of creating a superoptimizer for RISC-V assembly.

Synthesizing code that is optimal is usually a very time-intensive task not suited for large programs, because we are faced with a large search space. This, combined with the fact that we ideally want to strive to maintain completeness and soundness, restricts the use of heuristics in our synthesizer. This also results in some limitations in achieving fast runtimes.
Therefore, we focus on creating a specialized synthesizer for optimizations of small instruction sequences.

1.1 Related Works

One early example of an attempt at superoptimization is the GNU Superoptimizer superopt, also called GSO [2]. It is further described in a 1992 paper by Torbjörn Granlund and Richard Kenner [11], which in turn builds on the 1987 paper by Henry Massalin [18] that first introduced the topic of superoptimization.
GSO is based on enumeration. It was developed between 1991 and 1995 and supports, among other instruction sets, x86. According to the documentation, the program has a runtime of $O(m * n^{2m})$, where m is the number of available instructions in the chosen instruction set and n is the length of the shortest possible sequence implementing the goal function.
It searches through sequences of increasing length until solutions are found or a limit is reached. Obviously incorrect or sub-optimal sequences are discarded during generation [11].
The program is lightweight and easy to use, but has several drawbacks. For the x86 instruction set, solution sequences that are longer than 4 instructions are not generated. Generating long sequences would be impractical due to the exponential runtime. Furthermore, constants other than -1, 0 and 1 are not included. Some instructions are also not implemented, and there is a very small chance of generating an incorrect solution, making the program unsound.
Because constants are largely unsupported, superopt is not very suitable for our purpose of generating code for arithmetic expressions. Still, it is fairly fast and can be interesting to look at as a brute-force superoptimizer for simple functions that operate mainly on registers.

In recent years, more complicated and efficient algorithms for superoptimization have been proposed.

One of these more recent contributions to the field is Souper, a superoptimizer for the LLVM IR and the Microsoft Visual C++ compiler. LLVM is a compiler framework, and IR is the assembly-like intermediate representation of code used internally by LLVM. Superoptimization for IR is attractive because it is language-independent and can potentially benefit a large number of projects.

The goal of Souper is to automate the creation of optimizations on small compiler-generated instruction sequences, which are also called peephole optimizations [19], but the tool can be used in other ways as well. The program uses an SMT-Solver in its search strategy. Specifically, they use an CEGIS algorithm, which is an improved version of the one developed by Gulwani et al. in their 2011 paper about synthesis of loop-free programs [14].

The idea is to find an optimized sequence of instructions such that for some set of example inputs, the results of the optimized and original sequences are the same. This sequence is found by solving a query, where the result of the query is an instruction sequence that works for all the given examples. This possible solution is then verified on not just the examples, but on all possible inputs. If it does work on all inputs, the synthesis was successful. Otherwise, we add a counterexample that the proposed solution didn't work on to our list of input examples, and try finding another instruction sequence that works on this new set.

While in the worst case, the performance of CEGIS-based algorithms is still as bad as for exhaustive search, in practice algorithms using this idea achieve results that are substantially faster than exhaustive search. We use an approach based on CEGIS in our work we well.

Souper specifically is fast enough to be used as an automated optimization pass, and successfully creates optimizations not included in standard LLVM. Some of these created optimizations have now been included in the compiler [19].

There are also some restrictions to Souper's effectiveness: Souper exploits undefined behavior in an application or LLVM itself, which is not desirable. Bugs from the used SMT-Solver also carry over. Furthermore, optimizations synthesized by Souper are often more specific than the generic ones created by compilation developers, restricting some of the usefulness of the tool. As an optimizer on IR, it also misses out on specific optimizations that are only applicable to the final goal architecture [19].

Another very recent superoptimizer developed by Zhengyang Liu, Stefan Mada and John Regehr in 2023 is Minotaur [16]. It also optimizes for LLVM's intermediate representation. It focuses on integer single instruction/multiple data (SIMD) instructions.
Instruction sequences are created using exhaustive enumeration, however any literal constants are generated by an SMT Solver. Like with Souper, the main goal is to find missing peephole optimizations in compilers and again, the tool does find optimizations that LLVM is not able to generate on its own.
The program builds on Souper, in that it should be able find more transformations and also avoid some pitfalls with undefined behavior in LLVM. It also uses CEGIS as its basis for synthesis.
Minotaur can be loaded into the LLVM optimization pipeline, where it finds program fragments without loops and tries to find optimizations for these fragments. When extracting fragments, it is more aggressive than Souper. In practice, the amount of instructions in these fragments was capped to five.
Much of the ideas implemented in Minotaur are based on work by Sorav Bansal and Alex Aiken [6] from 2006, which optimizes x86 assembly. As there is a lot of overlap between their paper and the topics discussed so far, it will not be described in depth.

Lastly, we want to mention STOKE, which uses a stochastic search strategy [20]. The approach of STOKE is very different from the previously mentioned synthesizers, and is also less similar to our work. The authors use a cost function and randomized search to create programs. Synthesis and optimization are done simultaneously. The search is guided by a Markov Chain Monte Carlo sampler. The synthesis is most effective when it can discover effective rewrites incrementally, avoiding the issue of randomly searching in a large space.
In practice, the tool can generate more optimizations than Souper, because it also supports features like floating points instructions and memory. The developers of the Minotaur tool also claim that "STOKE can potentially perform transformations that Minotaur cannot, but we believe that its results are more difficult to translate into standard peephole optimizations than are Minotaur's" (p. 8, [16]).
The creators of STOKE themselves note that some rewrites are not found, and that in

general the tool will not always find rewrites that are as good as production compiler code. They believe an improved cost function would alleviate these issues.

1.2 Contributions

One obvious distinction of this work is that we optimize for the RISC-V instruction set. In our search, we found few existing works concerning superoptimization for RISC-V, "Synthesizing JIT Compilers for In-Kernel DSL" being one of the only examples [10]. That paper also goes in a different direction from our implementation, focusing on the creation of just-in-time compilers given an interpreter between two languages.

We also restrict ourselves to a smaller instruction subset than what is usual, only supporting arithmetic operations. This makes it easier to focus on correctness and experiment with different approaches; once a good approach is found, this could serve as a proof of concept and enable work on larger superoptimizers for RISC-V. Furthermore, arithmetic and bitwise operations are already often unintuitive to find by hand, which makes it more useful to focus only on these types of programs.

We implement a CEGIS-based synthesis using enumeration, with three different variants that iteratively improve upon one another. We show that it is possible to synthesize optimized RISC-V instruction sequences up to a length of four instructions using our implementation.

We compare our CEGIS-based implementation to an approach that uses enumeration without synthesis and show that CEGIS does indeed provide a significant improvement to runtime. Our work also provides a framework for working with RISC-V code in Python, which may make further exploration into the topic of RISC-V assembly or superoptimization easier. We discuss potential future improvements from our own results at the end of this report.

1.3 Document Structure

At the beginning in Chapter 2, we introduce and define concepts that were used in this work. More specifically, we give a more in-depth look at arithmetic expressions, the RISC-V architecture, SMT-Solvers, program synthesis and superoptimzers, and CEGIS.
In Chapter 3, we detail the implementation of our synthesis algorithm, looking at each major iteration and providing explanations for the thought process behind some of the decisions.
Next, Chapter 4 shows some benchmarks, comparing the different implemented synthesis approaches introduced in the previous chapter.
Lastly, in Chapter 5 we summarize our findings and talk about potential future improvements.

2 Background

In this chapter, we introduce the concepts that are used in our implementation of a superoptimizer.
First, we introduce arithmetic expressions, which is what we perform our superoptimization on.

2.1 Arithmetic Expressions

Arithmetic expressions are expressions that evaluate to a single integer value. They may include integer constants, integer variables, arithmetic operators, and parentheses.
There are many arithmetic operators that could be used in an arithmetic expression, but for our purposes, we support only a predefined list of operators. This list consists of some basic operators on integers, specifically addition, subtraction, negation, multiplication, integer division and modulo, as well as the bitwise operator arithmetic shift.
The reason we include arithmetic shift is that it is often useful for faster multiplication or division. Shifting left by n is equivalent to multiplying by 2^n, while shifting right by n is equivalent to dividing by 2^n and rounding down. Note that because right shift rounds down, it is not equivalent to our definition of integer division for negative quotients. Still, using arithmetic shift in general is helpful for synthesizing instruction sequences with a faster execution time.
The following is a definition of the syntax of arithmetic expressions as we use them.

Definition 1 (Arithmetic Expression)**.** A BNF-Grammar defining the syntax of a valid arithmetic expression e:

$\langle digit \rangle$ ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

$\langle const \rangle$ ::= 0 | $\langle digit \rangle$ | $\langle digit \rangle$ $\langle const \rangle$

$\langle var \rangle$ ::= 'a' | 'b' | 'c' | ... | 'z' | $\langle var \rangle$ $\langle var \rangle$ | $\langle var \rangle$ $\langle 0 \rangle$ | $\langle var \rangle$ $\langle digit \rangle$

$\langle op \rangle$ ::= + | - | * | / | % | >> | <<

$\langle expr \rangle$ ::= $\langle const \rangle$ | $\langle var \rangle$ | $\langle expr \rangle$ $\langle op \rangle$ $\langle expr \rangle$ | - $\langle expr \rangle$ | ($\langle expr \rangle$)

Semantically, we interpret the operations in these expressions according to their definitions in Python, with the exception of integer division and modulo. This is because Python behaves differently than other programming languages for those operators if one of the operands is negative. In Python, integer division is equivalent to dividing and rounding down. However, in our target language of RISC-V assembly, integer division is equivalent to dividing and disregarding everything after the comma. We use the definition from RISC-V assembly for our integer division. As for modulo, in Python the sign of the result matches the sign of the divisor, while in RISC-V assembly the sign of the result matches the sign of the dividend. Again, we use the RISC-V definition instead of the one from Python.
Later on, when we mention user input of arithmetic expressions, we also expect this user input to conform to the above definition to be valid.
As an example, consider the following arithmetic expressions:

Example 1.
Valid examples: e_1 = 100
e_2 = - -x / 12
e_3 = (var1 >> var2) * (3 + 3 % var1)

Invalid examples: e_4 = 1.3
e_5 = (3 - * x

2.1.1 Implementing Arithemtic Expressions in Practice

For expressions containing variables to evaluate to a single number, the variables have to have been instantiated to some value beforehand. However, in the context of programming, we might not know the value of a variable at the time of compilation. After all, if we did

always already know the value, we could just use a constant in place of the expression. Because the output is dependent on the value of the variables, what we are synthesizing is actually functions calculating arithmetic expressions instead of just expressions themselves. This means that we will interpret variables as function arguments.
It is also important to note that while in theory, two given arithemtic expressions are equivalent when both evaluate for the same value for all integers, we do not actually use an infinite domain for our variables. While we could verify equivalence on all possible values with our SMT Solver, we work with signed 64-bit integers in practice, so equivalence on numbers smaller than -2^63 or larger than $2^63 - 1$ are not relevant to us.

2.2 RISC-V

RISC-V is what is called a instruction set architecture (ISA) or computer architecture. CPUs can then implement such an ISA and execute its instructions.
One of the most well known ISAs is x86 (or its 64-bit version, x64). x86 is what is known as a complex instruction set computer (CISC) architecture, which means it has a lot of very specialized instructions, even for infrequently used operations. RISC-V is instead classified as a reduced instruction set computer (RISC) architecture. That means it does not directly implement those specialized instructions that you would find in CISC, simplifying the instruction set at the cost of small losses in execution time for some operations. The reasoning behind this is that many highly specialized instructions are used infrequently enough for it to not matter in the long run. RISC architectures are typically also load-store, which means that ALU operations may only be executed on registers, and access to the main memory of the computer has to be done separately. This further simplifies the design. A RISC instruction set may also be large in size, but would still be considered RISC, as the instructions themselves are simplified.
For an example of an instruction that might be found in a CISC architecture, consider `STRING MOVE`, which moves a whole block of data. This instruction might use flag bits and access the computers memory multiple times, which would not be obvious from the outside. In a RISC architecture, the exact implementation for moving a string would instead be left to the compiler.
A more commonly known example of a RISC architecture is ARM.

Unlike many of the other commonly used ISAs, RISC-V is open-source. It was first introduced in 2015 by Krste Asanović, who saw a need for for an instruction set architecture that is free and can be easily used by students as well as hardware designers [5]. He was later joined by David Patterson among others, who had already worked on the RISC ISAs RISC-I to RISC-IV.

The continued development of RISC- is managed by the organization RISC-V International, which is a collaborative effort to manage the publishing and maintenance of information regarding the RISC-V specification.

We chose RISC-V as our ISA because its simpler structure is well suited to academical work, and because it is an emerging architecture with no already established superoptimizers. To better understand how we utilize RISC-V, we now describe some more technical aspects of the RISC-V specification, based on the officially available documentations for the RISC-V ISA (Document Version 2.2) and RISC-V Assembly (1st edition) [4] [3].

2.3 RISC-V Specification Details

There are multiple versions of the RISC-V ISA, such as base integer instruction sets for both 32-bit and 64-bit systems, as well as multiple extensions. Some examples of extensions are the support of floating-points, multiplication and division, vector operations or SIMD instructions.

We use the 64-version of RISC-V RV64I with the extension M (for multiplication and division).

The specification for our version of RISC-V provides several registers with a size of 64 bits. The following is a list of registers relevant to our work. Many important registers like the stack pointer do not come up in our synthesis.

Register	Alternative Name	Description	Saved by
`x0`	`zero`	Hard-wired to zero	n/a
`x5-7`	`t0-2`	Temporary Registers	Caller
`x10-11`	`a0-a1`	Return Registers	Caller
`x12-17`	`a2-a7`	Function Arguments	Caller
`x28-31`	`t3-6`	Temporary Registers	Caller

Table 1: Subset of RISC-V registers used in this work

Some things to note:

First, RISC-V uses different registers for floating point values, which we do not support; the listed registers are for integers only. Second, there are no registers for integer overflow or integer underflow flags; RISC-V does not have flags for these events, so overflow or underflow has to be checked for manually. We usually do not perform these checks in our code, meaning if the arithmetic expression used is susceptible for overflow or underflow errors for 64-bit values, our assembly code will have these issues as well.

Lastly, we only use caller saved registers. This means we can overwrite them as we wish during our code; we also only implement expressions that do not contain function calls, and so don't have to worry about saving registers ourselves.

When synthesizing instruction sequences implementing an arithmetic expression, we act as if the sequence is a function. That is, we assign any used variables in the arithmetic expression to corresponding function argument registers and save the result of our expression in the return register `a0` at the end of the sequence. If we insert our resulting sequence as a function into existing RISC-V assembly code, we therefore also assume that any calls to the new function were done in a way that follows the correct calling conventions and saves the values of the caller saved registers before the function call. This leaves us free to potentially overwrite any register in our synthesized instruction sequences.

Because we only want to consider arithmetic expressions, we use and support a small subset of instructions from RISC-V assembly. Supporting fewer types of instructions speeds up our superoptimization at the cost of potentially synthesizing less optimal assembly.

As an example, we do not support the pseudo-instruction `li rd, imm`, which loads an immediate value `imm` to a register `rd`. Instead, we use `addi rd, zero, imm` for the same purpose, which is not technically the recommended way of loading immediate values by the official documentation.

Additionally, we do not support bitwise boolean instructions like `or rd rs1, rs2`, which performs bitwise or on `rs1` and `rs2` and stores the result in `rd`. This would be a useful extension to our subset, however it was left out for now for efficiency reasons.

The following is a collection of all the instructions in our subset of RISC-V assembly:

Instruction	Description
`addi rd, rs, imm`	`rd = rs + imm`
`subi rd, rs, imm`	`rd = rs - imm`
`slli rd, rs, imm`	`rd = rs << imm`
`srai rd, rs, imm`	`rd = rs >> imm`
`add rd, rs1, rs2`	`rd = rs1 + rs2`
`sub rd, rs1, rs2`	`rd = rs1 - rs2`
`mul rd, rs1, rs2`	`rd = rs1 * rs2`
`div rd, rs1, rs2`	`rd = rs1 // rs2`
`rem rd, rs1, rs2`	`rd = rs1 % rs2`

Table 2: Subset of RISC-V assembly instructions used in this work

There are some extra instructions that are not part of the synthesis, but may be optionally added to output assembly code. This is because they are needed to make our synthesized instruction sequence executable, if it is meant to stand alone.
Specifically, we add `.global start` and `_start` at the beginnning of the sequence, which specifies the entry point to our code.
At the end we add `addi a7, zero, 93` and `ecall`, which saves the opcode 93 for the exit syscall in an argument register `a7` and executes that syscall with `ecall`. When exiting, we also return whatever value we last saved in the return register `a0`. As discussed earlier, after executing our synthesized instruction sequence, the value in `a0` will already correspond to the result of the arithmetic expression we wanted to synthesize. Therefore, if we assign values to the variables used in the original arithmetic expression, we can check if the synthesized sequence computes the correct result by reading the exit code.
Here is an example of RISC-V assembly that computes the expression $e = x + 1$.

Example 2.

```
1. .global _start
2. _start:
4. addi a0, a2, 1
5. addi a7, x0, 93
6. ecall
```

Line 3 computes the actual result of the expression. x is the only variable and is stored in the first argument register `a2`. The immediate value `1` will be added to whatever is in `a2`, and the result will be stored in the return register `a0`. The rest of the code is just to ensure that we can execute the file properly.
Now that we discussed what it is we want to implement, we will go into more detail about the concepts and methods that are used in our code.

2.4 SMT Solvers

Throughout our work, we often want to establish whether or not two specifications are equivalent. The issue is that these specifications often contain variables in a large or infinite domain. To tackle this problem, we make use of SMT solvers. We translate the equivalence of our specifications to a set of logical constraints, and the solver can then prove for us whether there exists a satisfying assignment for our equivalence constraints that holds for all possible values.
SMT solvers work using SAT solvers; therefore we first briefly introduce SAT and SAT solving techniques.

2.4.1 SAT

SAT, or the Boolean satisfiability problem, is an NP-complete problem [9] that tries to determine whether there is a satisfying assignment of values for a given Boolean formula. It finds its basis in propositional logic, which allows for formulas corresponding to the following definition:

Definition 2 (Boolean formula)**.** A Boolean formula ϕ is built of variables x_i, negation ($\neg$), conjunction ($\wedge$), disjunction ($\vee$), implication ($\rightarrow$), equivalence ($\leftrightarrow$) and parenthesis. Variables x_i are in the domain $\{true, false\}$.

A SAT solver will try to determine if there is any assignment of Boolean values to variables for which the ϕ evaluates to $true$. This assignment can be expressed as a mapping function:

Definition 3 (Assignment)**.** For a formula ϕ containing a set of variables X, a mapping $\mu : X \rightarrow \{true, false\}$ is called an assignment.

We can say that a SAT solver, for a formula ϕ, will try to find any assignment μ that makes ϕ evaluate to $true$. If we find such a μ, we also say that μ satisfies ϕ.
Using this method, we can prove that a formula is false as well: If an assignment is found for $\neg\phi$, then ϕ must have been unsatisfiable. In this case, the assignment can be seen as a counterexample to the statement that ϕ is satisfiable.

When referring to SAT, we in reality usually mean CNF-SAT, which is the problem of searching for an assignment to a Boolean formula ϕ that is in conjunctive normal form (CNF).
Every ϕ can be rewritten to be in CNF, which means it doesn't matter if we try to solve CNF-SAT instead of regular SAT. Many algorithms take advantage of this fact, and work on ϕ that are in CNF, as it is more efficient.
Some of these algorithms are the Davis-Putnam-Logemann-Loveland algorithm (DPLL), conflict clause learning (CDCL), and stochastic local search. They have good performance for a large portion of formulas, but because SAT is NP-hard, they will still exhibit exponential runtime in the worst case.

Aside from the importance of SAT to theoretical computer science, SAT Solvers are also of great practical importance in many fields such as AI, hardware verification, solving graph problems or product configuration.
Next, we will look at another one use case of SAT solvers, which are SMT solvers.

2.4.2 Satisfiability Modulo Theories

Some problems cannot be captured using a simple boolean formula like the ones we introduced earlier. In this case, we need to use formulas with greater expressiveness. Our specific goal is to reason about more than just booleans, for example about integers, floats, bitvectors or arrays. To achieve this, we can turn to SMT solvers.
Unlike SAT solvers, SMT solvers now accept formulas in first-order logic, which is an extension from our previous Boolean formulas. In addition to the symbols defined earlier in definition 3, we now may use the following symbols as well:

- the equality symbol $=$
- the universal quantifier $\forall$ and existential quantifier $\exists$

We now also allow the use of predicate and function symbols. When considering a

formula ϕ, we now need to have a corresponding background theory $\mathcal{T}$, that defines the interpretation of the symbols used. A predicate may operate on non-binary values, but will evaluate to a boolean value. A typical example for a predicate would be a linear inequality on integers like $\leq$. A valid formula ϕ will consist of a conjunction of such predicates, each of which is sent to its corresponding theory solver. ϕ is satisfiable if we find a valid assignment $\nu[\mathcal{T}]$, that is an assignment of values with regards to the theory $\mathcal{T}$, for each predicate in the formula. Additionally, assignments for variables that are used in multiple predicates need to be agreed on by all predicates.

Some examples for theories that are frequently supported by SMT solvers would be Linear Integer Arithmetic, Arrays, or Bitvectors. We will need to use Bitvector theory in our implementation, because it bitvectors can best represent RISC-V registers and support arithmetic as well as bitshift operations.
Here is an example for a formula ϕ in the Bitvector theory:

Example 3. $\phi_{BV} = (x << 2 == 4) \wedge (x > 0)$

This formula consists of two predicates. It is satisfiable, because the valid assignment $\nu_{BV} = \{x \mapsto 1\}$ exists.

SMT Solvers can be applied in many different ways, such as in software verification, test case generation, model checking, scheduling, and like in our case, program synthesis.
The specific solver we use is called Z3. Z3, or Z3 Theorem Prover, is a SMT solver implemented in C++ and developed by Microsoft Research. It can be used in SMTLIB2 Format, which is commonly used for SMT Solvers; however, there also exist bindings for programming languages such as Python, Rust, Java, .Net, Ocaml, C, and C++. When speaking of Z3, we will mainly refer to the Python implementation, as this is the version we use in our code.
We chose Z3 because it integrates well with our programming environment, is an established tool, supports a bitvector theory, and is well documented.
Next, we want to discuss program synthesis, an area where SMT solvers are frequently used.

2.5 Program Synthesis

Three sources were used to inform much of the content in this section. First, there is the work-in-progress version of a book on Program Synthesis provided by Nadia Polikarpova and written by Loris D'Antoni and Nadia Polikarpova [17]. The slides and git repository of the CSE 291 Program Synthesis course, also by Nadia Polikapova, were used as well [1]. Lastly, we draw on information from another book called "Program Synthesis" published by Microsoft in 2017 and written by Sumit Gulwani, Oleksandr Polozov and Rishabh Singh [15].

In program synthesis, we try to automate the process of finding a program that meets a given higher-level specification defined by the user.
Some of the first program synthesizers were mentioned in 1969 by Cordell Green in "Application of Theorem Proving in Problem Solving" [12] in relation to logic programming. The synthesis worked by specifying constraints and solving them using proof techniques. However, most advancements in the field only came after 2000, as constraint solving techniques were refined, computing power increased, and new search techniques were developed.
We want to take a look at how the problem of synthesis can be defined, and what some techniques to solve it are.

Sumit Gulwani introduces three dimensions by which we can classify a program synthesis problem: The behavioral specification, the structural specification, and the search strategy [13]. In the following, we elaborate on those terms further.
The behavioral specification corresponds to the higher-level specification given by the user. What exactly this behavioral specification looks like depends on the task and approach used; it could be a logical formula, a natural language text, a reference implementation, a list of input/output examples, or something else entirely. Unlike in a compiler, most of these behavioral specifications cannot be directly mapped to a program using simple translation rules.
We search for fitting programs in a predefined search space, the structural specification for the problem. Ideally, this space should be expressive enough to include all interesting programs while still being as small as possible for efficiency. We usually look at subsets of programming languages, or domain-specific languages (DSLs). The size of these search

spaces tends to explode quickly.
The complexity and size of the search space make it important to include domain-specific insights into the synthesis process whenever possible.
These difficulties also mean that we need to select an efficient search strategy, which is the third dimension defined by Gulwani. The search also benefits from being guided by knowledge on the task. We briefly describe some search strategies which contain aspects that we use in our implementation.
First, there is enumerative search, where all possible programs of increasing size are tested. Duplicates or obviously unpromising programs may be pruned to combat the issue of search space size.
Representation Search uses abstraction to group large numbers of programs together and searches on this new space.
Constraint-based search translates the behavioral and structural specifications to logical constraints, and uses a solver to find a fitting program.
We briefly summarize the goal of synthesis in the following definition:

Definition 4 (Synthesis)**.** Given a specification γ and search space S, synthesis is the process of finding a program in S that satisfies γ.

One example of an already existing popular framework for synthesis is SKETCH, which takes an already written program that contains holes as input and then automatically fills the holes using a behavioral specification. There is also PROSE, which is similar to Microsoft Excel's FlashFill feature in that it works by producing programs from examples. Lastly, ROSETTE enables solver-aided programming for easier creation of synthesizers and verifiers in new languages.
These frameworks can be very useful, however we chose not to use them in this work both to have full control over the details of our implementation and to gain a better understanding of the inner workings of the techniques used in synthesis.

To close out this section, we want to mention the general challenges in program synthesis. First, synthesizing a program involves verifying whether a program meets a given specification. That means synthesis is a harder problem than verification. For Turing-complete programming languages and arbitrary constraints, verification is already an undecidable problem, so in these cases synthesis will also be undecidable.
We already mentioned the large search space, which in fact grows exponentially with

program size for most behavioral specifications, because we are usually have to combine all possible programs of the previous size with each possible new instruction.
Lastly, communication of the correct behavioral specification can also be difficult. If the user has to be overly specific when inputting a specification, not much is gained over writing a program by hand in the first place. If the input is too unspecific, there will be too many possible solutions.
Next, we will talk about one area where synthesis is applied: Superoptimization.

2.6 Superoptimizers

In section 1.1 we already talked a bit about the history and some approaches to superoptimization. Here, we aim to explain the concept itself a bit more generally.
Given an instruction set and function, superoptimization is the task of finding an optimal instruction sequence that computes the function [18]. The function corresponds to the behavioral specification γ of the synthesis task, and is traditionally a loop-free (straight-line) code fragment written using the goal architecture's instruction set. Some attempts have been made to optimize loops as well [7].
Typically, the instruction set is not used in full, but restricted to a meaningful subset. Most research focuses specifically on operations on bitvectors.
Superoptimization often requires an exhaustive search of all possibilities in the worst case. This would mean a search space of size $O(m^n)$ where m is the number of possible instructions and n is the length of our solution instruction sequence.
This fact is easy to prove inductively:
For the initial case $n = 1$, we have sequences of length 1, which is equivalent to the instruction set of size m. $m^1 = m$, so our assumption holds for the initial case.
For $n' = n+1$, we combine every sequence with length n with our m options for instructions. Using the inductive hypothesis that m^n is the number of all sequences of length n, we get $m^n * m = m^{n+1} = m^{n'}$ options for n'. Thus our assumption also holds in the inductive step, meaning we have proven our assumption for all $n \geq 1$.

We can try to lessen the issue of search space size by using clever synthesis techniques.
Still, it is best to focus on smaller problems. Henry Massalin notes that "[T]he best use of [the] superoptimizer has been as an aid to the assembly language programmer" (p.3, [18]). This leads to our decision to implement our own superoptimizer on RISC-V assembly.

Next, we will discuss CEGIS, a main idea we use in the synthesis of programs in our superoptimizer.

2.7 CEGIS

CEGIS, or counter-example guided synthesis, is an iterative synthesis approach that was first written about in a 2006 paper by Solar-Lezama et. al [21].
It avoids the issue of having to create a solution to a problem that works for all infinitely many input options. It does this by combining a synthesizer and a verifier:
First, CEGIS synthesizes a program that satisfies the specification γ for a finite set of examples. Because the set is finite, we can avoid solving formulas with difficult universal quantifiers when using an SMT solver.
Once a program that works for all examples is found, a verifier is used to check if the solution candidate is actually correct for all possible inputs. This verification step either proves that our candidate satisfies γ, or it provides a counter-example. When using an SMT solver, we obtain this counter-example by checking for inequivalence instead of equivalence of our candidate to the specification. The solver will try to prove that there exists an input for which the synthesized program and the problem specification are not equivalent, that is, that there is an input for which they each produce a different output. If the formula for this is satisfiable, we can obtain the counter-example directly from the model generated by the solver; if the formula is unsatisfiable for all possible inputs, we have proven through contradiction that the synthesized program is correct.
If we obtain a new counterexample, we add it to our list of examples used in the synthesis step. This ensures that we do not try the same solution twice. Often, only a few examples are needed to find a solution to a problem. CEGIS calculates both the minimum number of examples needed for synthesis and a synthesized solution.

CEGIS is sound if the verifier is sound: Unless the verifier itself makes an error or does cannot reach a decision, we can be sure that the synthesized program and the specification are equivalent on all inputs.
CEGIS is also guaranteed to find a solution to a synthesis problem, but only if the search space is finite. If the space of possible programs is infinite, CEGIS might only find the solution in infinite time and simply never terminates. CEGIS is therefore not complete
In the worst case, the time complexity of CEGIS is also as bad as naive enumerative

search on all possibilities. This would happen if our counter-examples end up being so unrestrictive to the search space that every program is checked as a solution candidate.

One benefit of using CEGIS is that there still is a lot of freedom for actually implementing it. The only thing that matters for implementing the synthesis is that it can produce candidates when given examples. Similarly, the verification step needs to be capable of proving equivalence and providing counter-examples, but the actual implementation is up to the programmer. In our work, we use an SMT solver for both the synthesis and verification, and use slighly modified enumeration to generate candidates.

3 Approach

The code we implemented can be found at `https://github.com/lh535/superoptimizer_with_program_synthesis` and was written in Python 3.10, using the `z3` and `z3-solver` libraries (version 4.12.2).
For compilation, the `gcc-riscv64-linux-gnu` library was used, which also comes with the QEMU RISC-V emulator that we use to run any compiled RISC-V assembly.

3.1 Problem Definition

We want to create a superoptimizer for RISC-V assembly instruction sequences that computes arithmetic expressions. As input for describing the specification γ, we want to allow either an arithmetic expression, or existing RISC-V assembly which computes an arithmetic expression. Our implementation should then generate the shortest sequence of instructions that is equivalent to γ.
We want to use a synthesizer to achieve this, specifically one that utilizes CEGIS. We also want to implement a version of the synthesizer that does not use CEGIS as a comparison point. As a secondary objective, we want to create a framework for working with RISC-V in Python.

3.2 Implementation

We briefly summarize the features that were implemented as part of this work:

- An implementation of a RISC-V DSL for representing RISC-V assembly in Python. In addition, the capability to convert from user input, Python functions or RISC-V assembly to this DSL. Execution of instruction sequences in the DSL can be simulated.
- Generating RISC-V Assembler code for arithmetic expressions from user input using a naive bottom-up compilation approach.
- Checking if a given RISC-V instruction sequence is equivalent to a python arithmetic function or another RISC-V instruction sequence (For sequences conforming to the RISC-V DSL).
- Synthesis and superoptimization of RISC-V assembly matching a arithmetic expression, using CEGIS or a simple enumerative approach.

The synthesis using CEGIS can be executed by simply executing 'main.py', which will provide the necessary instructions. There are also the option to execute a bottom-up compilation approach to generate non-efficient RISC-V assembly. By using `make run`, it is possible to compile and execute the generated code, provided `riscv64-linux-gnu` is installed. For simple debugging, the result for the function, if all variables are set to 0, is returned in the console in the form of the exit code. A drawback of using this method of returning the value is that the exit code is limited to a number between 0 and 255.
The verifier for arithmetic expression equivalence and the synthesis functions are contained in `cegis_verify.py`. The functions enabling synthesis are contained in `synthesis.py`. Benchmarking of the different methods implemented is implemented in `benchmarking.py`; the results used for our own analysis are already stored in the Benchmarking folder.
The internal RISC-V assembly DSL is defined in `riscv_dsl.py`. This also contains replacement functions for Python's modulo and floor division functions, to match other programming languages. Naive compilation for generating RISC-V assembly can be found in `python_ast_to_func.py`.
Conversion from user input or a python function to RISC-V DSL can be found in `python_ast_to_dsl.py`, conversion from RISC-V assembly code to the DSL and back in `dsl_input_output.py`, conversion from RISC-V DSL to a python function in `dsl_to_func`.

3.2.1 RISC-V DSL

To effectively work with and generate RISC-V assembly code, we have to create a representation we can use in our Python code. This RISC-V DSL is defined in `riscv_dsl.py`. There is a class for registers called `Reg(num: int)`, where `num` is the register number. This class is to be used for temporary registers, in RISC-V those are `x5 - x7` and `x28 - x31`. Upon initialization, an assertion is used to ensure the register number is within the valid range. Non-temporary registers get their own subclasses and assertions.
When using `repr()` on an object of type `Reg`, the resulting string is in RISC-V assembly format again. This behaviour of `repr()` is the same for all defined types in this DSL, enabling easier comparability and conversion.
There are multiple subclasses of `Reg(num: int)`, the most important being
`Regvar(num: int, name: str)`. This class represents all function argument registers we can use (`a2-a7`). The attribute `name` is used when the argument register corresponds to a specific variable from the original specification; in this case the name of that variable is saved so we can refer back to it.
Aside from `Regvar`, the two other subclasses of `Reg` are `Zero()` and `ReturnReg()`. They correspond to the hard-wired zero register `x0` and the register for the function return value `a0`, respectively.

Now that we have a representation for all types of registers we want to use, we can also create a representation for instructions on those registers.
`Instr(op: str, *args: Reg | int | BitVecRef)` is a class capturing all types of instructions we use in our code. The main intended use is to create binary arithmetic operations, but we can also create RISC-V instructions like `ecall` by only providing the instruction string as an argument: `Instr("ecall")`.
For arithmetic instructions, we use the RISC-V assembly name of the instruction for the argument `op`. The next three arguments are registers for the destination as well as the left and right operand of the operation. While we technically allow for all these three arguments to have types other than `Reg`, when representing a well formed instruction only the right argument should ever be anything other than a register. This is because in immediate RISC-V instructions like `addi`, only the right operand is an immediate. As we work with 64-bit integers, immediate operands can be of type `int` or `BitVecRef`. `BitVecRef` is a type from Z3, and for our purposes stands for a symbolic 64-bit integer constant.

As a subclass of `Instr`, we also have `Regassign(reg: Reg, num: int)`, which just provides a shorter way of writing `Instr("addi", reg, Zero(), num)`.
These classes are enough to represent the entire subset of RISC-V assembly instructions defined in chapter 2.3.

3.2.2 Naive Bottom-Up Compilation

We want to implement a bottom-up conversion from user input to RISC-V assembly as a comparison point and as input for our superoptimizer. We do this by applying syntactic conversion rules, like the ones found in most compilers.
The code for this can be found mainly in `python_ast_to_dsl`.
First, an object from class `Compiler()` needs to be created. Afterwards we can convert user input to a list of instructions in the RISC-V assembly DSL by using the function `compile_input(self, input: str) -> List[Instr]`, where `input` is a string of an arithmetic expression conforming to Definition 1. Alternatively, `compile(self, e: AST) -> List[Instr]` converts directly from an AST representation of a Python function computing an arithmetic expression to a RISC-V assembly DSL instruction list.

We convert the arithmetic expression from the bottom up into a list of RISC-V DSL instructions. For binary operations, we first compute the result of the left operand, then the result of right operand.
Our conversion fails if the input was invalid or if we run out of temporary registers to assign constants to. We free up the temporary registers for re-use after the result of binary operation that contained the corresponding constant was completed. Therefore we run out of constant registers only in very specific cases. For example, because we always convert the left operand first, an expression of form `c1 + (c2 + (c3 + (c4 + (...))))` may cause issues: We save the constants on the left side of every binary operation, but have to resolve the right operand recursively, needing another temporary register for the left constant in each operation. In total, there are seven temporary registers available.
At the end of our computed instruction sequence, we add an instruction that saves the final result of the arithmetic expression into the return register `a0`.

3.3 General Approach for Synthesis

We use CEGIS to find superoptimized RISC-V instruction sequences. As discussed in section 2.7, CEGIS consists of two phases that alternate: The synthesis of candidates, and the verification of the equivalence of those candidates to our input specification γ. The kickoff to our CEGIS loop and the verification componenets are implemented in the file `cegis_verify.py`; the synthesis of candidates is implemented in `synthesis.py`.

To start the synthesis, we need to create an object for the class `Verifier` for our input arithmetic expression. We can pass the arithmetic expression as a Python function, as a String containing the expression, or a list of RISC-V instructions.

Once we have our verifier object, we can use the method `verify(guess: List[Instr]` to directly check if a RISC-V instruction sequence `guess` is equivalent to the arithmetic expression associated with our Verifier.
If we want to synthesize a superoptimized sequence corresponding to the expression instead, we can call the method `cegis_0`, `cegis_1`, or `cegis_2`. These methods correspond to the different implementations of a synthesis algorithm to be used in CEGIS, and each implementation is explained in more detail in the coming sections. Performance should improve when using a method with a higher number, because the implementations iterate upon one another. The result returned by each method is always the same.
The general implementation of the CEGIS loop itself is the same for each approach: We create an instance of a Z3 solver and add Z3 variables for each of our function arguments to it. Then we create a starting example of an input-output pair for our arithmetic expression. This example is added to a list, which we pass to our synthesizer. The implementation of this synthesizer differs for each approach, but it will always return a guess that produces the same output for every example input in the current list of examples.
We verify if the guess is equivalent to our original arithmetic expression for all possible inputs. If this is true, we return the guess as our synthesized solution; otherwise, the verification will have returned a counter-example which we can add to our list of examples, and we try the synthesis again.

There are some other general things to note for our implementation:
First, our Z3 variables are of type `BitVec`. This is because they allow operations on bits like left/right shift, and because they model RISC-V assembly registers most closely. We put a constraint on our variables to not exceed a minimum and maximum value, because otherwise Z3 might not assign values to our variables when checking for equivalence.
We also want to specifically mention division and modulo again: In Python, division and modulo involving negative numbers operate differently from other programming languages, including RISC-V assembly and Z3 bitvectors. For this reason, we convert any use of division or modulo into our own functions `pydiv` and `pymod`, which are defined in `riscv_dsl.py`. It is possible to circumvent this conversion if a python function is provided to the initializer of `Verifier` directly. This would result in solutions that do not actually correctly model the input.
The last thing to mention is that in RISC-V, instructions for multiplication, modulo and division do not have a version that accepts immediates directly as operands. This differs from addition, subtraction and bitshift, which have a version that accepts an immediate as the right operand. We always prefer immediate instructions, because having to use constant registers means we also have to have an additional instruction for initializing that register.

Now we describe each implementation of a synthesis used with CEGIS. We also implemented a synthesizer that uses a bottom-up approach to superoptimization without CEGIS, which is briefly described at the end of this section.

3.4 Cegis0: Exhaustive Enumerative Synthesis

In this synthesis as well as all the following approaches, we use symbolic constants. This means that any time we have an instruction that would usually take an integer constant as one of its arguments, we use a Z3 variable instead. When checking whether a candidate containing a constant works for all input/output examples, we need to use a call to the Z3 solver to verify there is a valid assignment for the symbolic constant.
All of our synthesizers for creating candidates have an enumerative approach as their foundation. This current implementation generates a list of all possible combinations of our available RISC-V assembly instructions up to a certain length. We limit the possible length of instruction sequences to two, because the list of possible combinations grows very

quickly, potentially leading to an out of memory error thrown by Python for sequences that would be longer than two. If we didn't use symbolic constants, we would run out of space even faster.

When creating all possible instruction sequences of length one, we have to iterate over all possible operators defined in section 1 of the background chapter. In this loop, we again have to iterate over all possible registers (all temporary registers, any argument registers needed in the original arithmetic expression, the zero register and return register) for the first instruction argument. For immediate instructions, we then use a symbolic constant as the second argument. For non-immediate instructions we have to iterate over all available possible registers for the second argument once more. The destination for the instruction is just the return register `a0`.

When creating instruction sequences of a length n greater than one, we do the same thing we did for sequences of length one, but also have to combine each generated instruction with all possibilities generated for $n - 1$. Additionally, unlike before, we also have to allow arbitrary destinations, meaning we have to iterate over all possible registers again.

We generate all possibilities for a given length first, and then check if any of the generated instruction sequences has a valid assignment of constants for the example list provided by the CEGIS loop in our verifier. If this is the case, we can pass the instruction sequence as a possible solution candidate to the verifier in the CEGIS loop.

When using this method for generating instruction sequences, we end up with a large number of invalid snippets. The reason for this is that we will often use uninitialized registers, which will lead to an error. For example, `addi a0, x5, 2` is a complete instruction sequence contained in the list of candidates returned by Cegis0; however, `x5` is not initialized and therefore the sequence is not valid. An obvious next improvement to our approach would be to prune our search space to avoid generating invalid snippets in the first place. We should also avoid generating all candidates at once, to avoid running out of memory.

3.5 Cegis1: Pruning Search Space and Generators

This approach builds on the last one, but prunes the search space when generating possible candidates. We first want to avoid generating instruction sequences that are clearly invalid. The main way we could generate invalid instruction sequences in our last attempt was by using temporary registers which were not initialized yet. To solve this, we only allow

temporary registers to be used as a destination, not as an argument, in the first instuction of a sequence. Only temporary registers that were used as a destination before can be used as an argument in later instructions.
As a further optimization, when introducing a new temporary register as a destination of an instruction, we do not iterate over all possible unused temporary registers but just try one of the possible registers. This does not result in a loss of expressivity, because the interchanging the constant register with a different one does not change the result of the computation.
We also eliminate some options for addition or multiplication instructions. Because these operations are commutative, it does not matter which order the operands are in. Therefore, if we already generate instructions of form `add rd, rs1, rs2` or `mul rd, rs1, rs2`, the instructions `add rd, rs2, rs1` and `mul rd, rs2, rs2` would be redundant and we do not have to consider them.
We also make use of a specific property of division, modulo and subtraction, which is that when both operands in the operations are identical, the result is always a constant. This means we could also just assign that constant to the destination register instead of using these operations and get the same result. We exclude such instructions for this reason, again without losing any expressivity.

Lastly, we want to mention another big improvement to this method over the last one, which is the use of generators. We test for equivalence of any of our generated instruction sequences to the input-output examples one by one. Additionally, each full instruction sequence is independent from any of the other generated sequences. Therefore, we never actually need to have the full list of possible instruction sequences of a given length in memory. Instead, we use a Python generator to yield a new possibility one after the other, which requires a lot less memory. This also means we can stop generating all possible candidates for a given length if a solution is found within some of the first candidates, saving us a lot of unnecessary steps.
Cegis1 is already a big improvement and introduces a lot of important concepts. We still try to add one additional improvement to our implementation.

3.6 Cegis2: Memoization

We notice that when adding instructions to the end of an existing sequence, we always combine the existing sequence with a list of the next possible instructions. This list is very similar every time. Computing a single instruction itself is very cheap, but the computation of all possible single instructions to be added at the end of a sequence is done very frequently in our code and could be considered an expensive operation. Therefore, it would make sense to use memoization in our implementation. We want to save the result of the repeated operation of computing all instructions that can be added, and reuse it instead of re-computing each instruction every time.

Note that while the list of instructions we can add is very similar each time, it is not always exactly the same. This is due to the fact that we can have a different amount of constant registers currently in use. To solve this issue, we save a list of possible single instructions for each number of constant registers in use. The number of registers is bounded by the maximum length of instruction sequences we are currently generating, as we use at most one new constant register with each instruction that we add to the sequence. We save these sets of instructions in a dictionary, which is also saved between calls to the candidate generator for re-use. We could also go one step further and save not only all single instructions that can be added at the end, but also all possible sequences of length n-1 when generating sequences of length n. We could then also use multithreading, where we run independent threads that compute these sets of instructions simultaneously for each length. Implementing this idea was theoretically possible, but we observed that it did not speed up computation. Likely, this is because we now have to hold large lists in memory and we cannot reach instruction sequences of a length high enough for the optimization ideas to start making a difference. Furthermore, when using memoization of large lists and multithreading, we can no longer use a generator, which would result in a big hit to performance and memory usage. This is the reason why we limit ourselves to saving only the lists of single instructions, computed without multithreading.

One last thing to note is that we copmute a separate list of instructions that can be put at the very end of a sequence. This list has to be different from the others because we have to force the use of the return register as a destination.

Cegis2 the last of the CEGIS-based approaches; the next approach was implemented as a point of comparison, and just tries each candidate directly instead of using CEGIS.

3.6.1 Simple Enumerative Synthesis

We implemented a simple enumerative synthesis approach without CEGIS. This allows us to see whether or not CEGIS is truly effective at speeding up the synthesis process.
We use the same function as Cegis2 to get possible candidates, but we do not check for equivalence on a list of examples like in CEGIS. Instead, we ask Z3 to determine the equivalence of each candidate to the specification for all possible inputs. This means we now have to use the `forall`-quantifier for each variable from the expression. Usage of the universal quantifier generally slows down our calls to Z3 significantly, which is why we tried to avoid them by using CEGIS. To see whether or not the overhead from using CEGIS can balance out the avoidance of `forall` quantifiers, we run some benchmarks on our implementations and present the findings in the next section.

4 Benchmarking

4.1 Benchmarking Method

We use Python's `timeit` and `memory_profiler` libraries to benchmark our code and compare our different approaches. The results presented here were obtained on a machine with 16GB of RAM and AMD(R) Ryzen 5 3600 6-core processor @3.6GHZ, running on Ubuntu 22.04.3 LTS.
We run the synthesis functions on RISC-V instruction sequences of different lengths and monitor both the time and maximum memory used by each function. The length of the solution instruction sequence as well as the types of operators used in the instructions are the determining factors for performance, as we synthesize shorter sequences before long ones and always use the same order when trying operators.

4.1.1 Examples used in the Benchmark

We define five groups of examples to use in benchmarking:

- 1, Simple: "3", "x + 10 - 5", "x * 1"
- 1, Complex: "x * 4", "x % y", "x / 16"
- 2, Simple: "(x + 2) * 4", "x * 2 + 20", "x + y + 1"
- 2, Complex: "(((x + 3)* 4) - 1) * 2", "x % 3", "x / 3"
- 3: "(x / 3) + 3"

The groups are ordered in ascending difficulty, and are named after the number of lines the solution will contain. The denotation "Simple" is for instruction sequences containing operators that are synthesized first, while anything in a "Complex" group will have to

have a solution that contains operators that are synthesized later.

The order in our synthesis tries operators is +, -, «, », *, /, %.

Note that for all of these examples, the solutions presented by all of the synthesis algorithms are identical. The only difference is in the speed and memory consumption during execution.

Also note that group 3, only two out of the four approaches succeed in providing a solution. Furthermore, the runtime increases sharply for this group, which is why it will not be included in any graphs so as to not distort the scale too much.

4.2 Comparing CEGIS to Simple Enumeration

First, we want to compare our CEGIS-algorithm 3.6 to the simple enumerative synthesis without CEGIS 3.6.1.

The following is the result of taking the average time taken for each example in the group by both of these approaches.

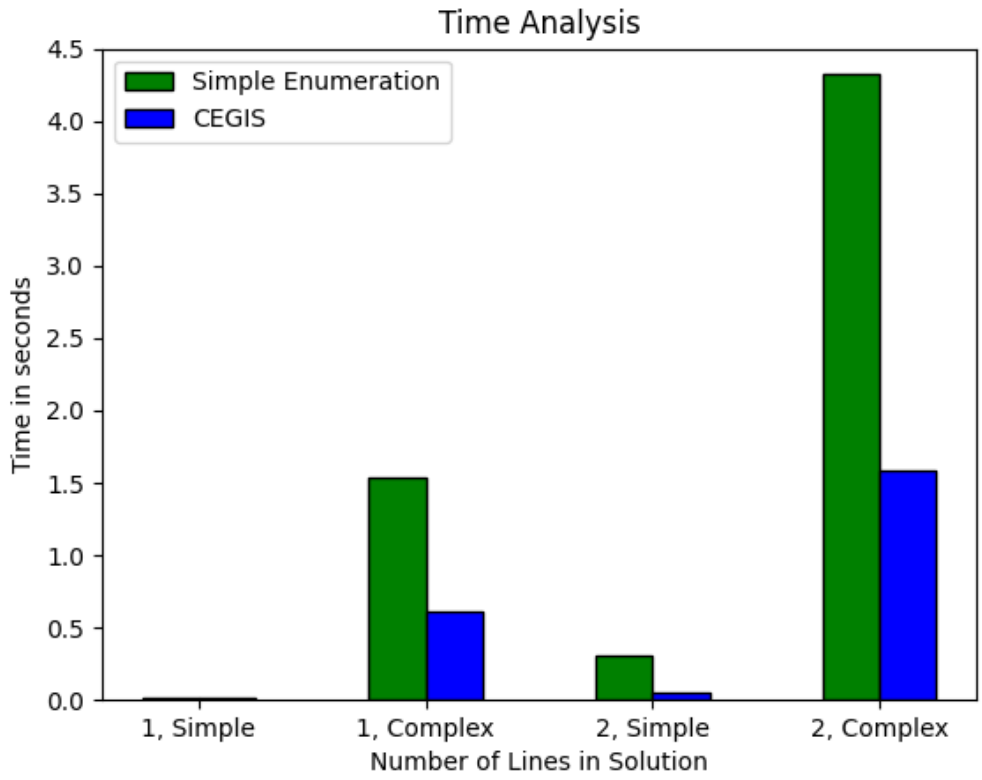

Figure 1: Comparison between the computation time taken by the simple enumeration and CEGIS algorithm, for the first four example groups defined in 4.1.1

Expression	**Enumeration** runtime in sec	**CEGIS** runtime in sec
3	0.013	0.014
x + 10 - 5	0.010	0.013
x * 1	0.009	0.013
1 Line, Simple	**0.011**	**0.013**
x * 4	0.027	0.023
x % y	2.217	0.448
x / 4	2.189	1.464
1 Line, Complex	**1.144**	**0.978**
(x + 2) * 4	0.195	0.060
x * 2 + 20	0.138	0.039
x + y + 1	0.465	0.045
2 Lines, Simple	**0.266**	**0.048**
(((x + 3)* 4) - 1) * 2	3.474	1.143
x % 3	4.541	4.520
x / 3	3.732	0.946
2 Lines, Complex	**3.916**	**2.536**
(x / 3) + 3 (3 Lines)	Error	70.213

Table 3: Detailed results for computation time taken by the simple enumeration and CEGIS algorithm, for all examples defined in 4.1.1

In graph 1, we can easily see that our CEGIS algorithm is considerably faster at finding a solution than the simple enumeration. Given that we use the same method to get candidate programs in both cases, we can conclude that simply using CEGIS as a method is the reason for this improvement.

When looking at the detailed results from the table 3, we also notice that for very simple expressions, the CEGIS approach is slightly slower. This is likely due to the overhead from the additional code and from using two instead of one Z3 call - first for checking equivalence on the examples, then for verifying the solution - however, the impact seems to be inconsequential. The CEGIS approach outperforms the simple enumeration in every case outside of the first example group.

Interestingly, a complex single line solution is harder to compute than a simple two line solution. This is a somewhat surprising result; it is likely that the reason lies with the operators division and modulo, which we needed to modify from their original definitions in Python. Having to handle user-defined functions likely slows down the calls to the Z3 solver.

Lastly, we want to look at the result of the last group, which contains expressions that would need three instructions to compute. When attempting to synthesize a solution

using simple enumeration, the program will freeze; the reason for this is slightly unclear, but the issue seems to lie with the Z3 calls, which are more complex in this method compared to the ones in the CEGIS approach. When using CEGIS, we do get a result, albeit with a highly increased computation time. This is to be expected, as the number of potential solution candidates increases exponentially. Specifically, the number of possible instruction sequences increases from roughly 1 million to roughly 389 million when going from sequences of length 2 to sequences of length 3.

After determining that using CEGIS does improve synthesis performance for our superoptimization, we now want to compare our different approaches when implementing CEGIS.

4.3 Comparing different CEGIS variants

We compare the runtime of each of our CEGIS variants, namely Cegis0 3.4, Cegis1 3.5, and Cegis2 3.6.

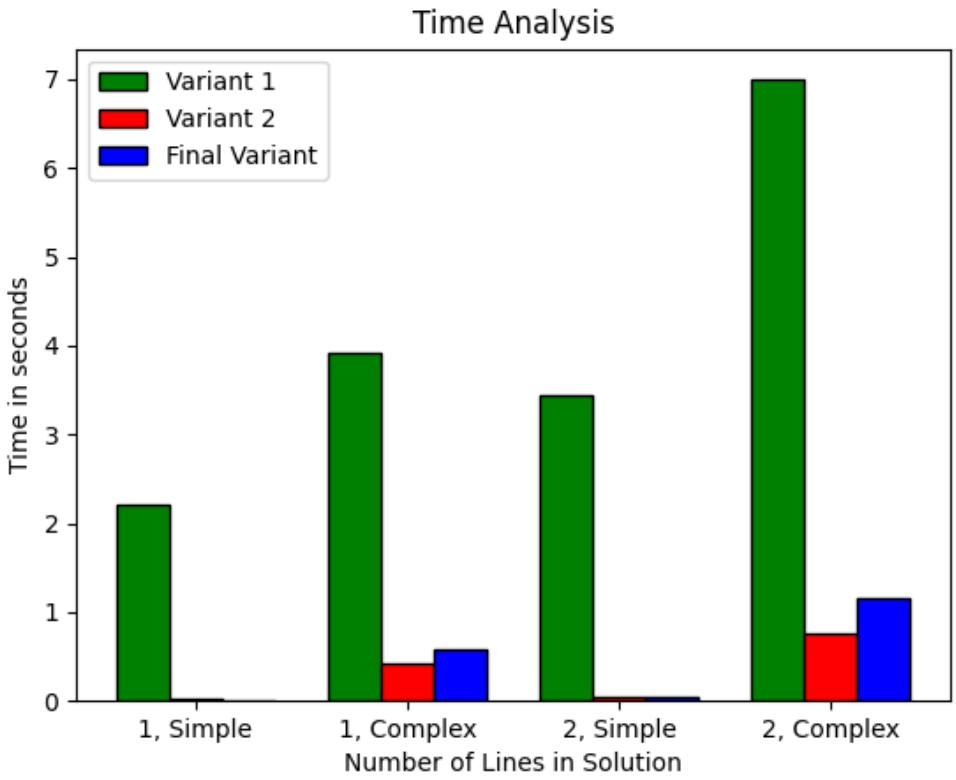

Figure 2: Comparison between the computation time taken by the three CEGIS algorithms we implemented, for the first four example groups defined in 4.1.1

Expression	**Cegis0** runtime in sec	**Cegis1** runtime in sec	**Cegis2** runtime in sec
3	0.973	0.017	0.012
x + 10 - 5	2.831	0.013	0.012
x * 1	2.799	0.012	0.012
1 Line, Simple	**2.534**	**0.014**	**0.012**
x * 4	2.825	0.032	0.019
x % y	4.618	0.223	0.164
x / 4	4.326	0.062	1.442
1 Line, Complex	**3.923**	**0.106**	**0.875**
(x + 2) * 4	2.972	0.043	0.034
x * 2 + 20	2.991	0.038	0.035
x + y + 1	4.332	0.050	0.041
2 Lines, Simple	**3.432**	**0.043**	**0.037**
(((x + 3)* 4) - 1) * 2	13.861	0.562	1.143
x % 3	4.374	0.661	4.520
x / 3	2.896	0.702	0.946
2 Lines, Complex	**7.377**	**0.642**	**2.869**
(x / 3) + 3 (3 Lines)	Error	276.104	68.919

Table 4: Comparison between the computation time taken by the three CEGIS algorithms we implemented, for the example groups defined in 4.1.1

Since we iteratively improved upon each previous approach, we expect the runtime to decrease with each variant as well. As we can see in the graph 2, this is only partially true. The first approach is predictably much slower than the next two. However, the second approach, which implemented search space pruning and generators, is actually slightly faster than the third one for the first four example groups. This is because the third approach uses dynamic programming, which does not help performance when generating short solutions. We still conclude that this last attempt is the more promising one due to the greatly improved runtime for solutions longer than two lines, as we can see from the last line in the table 4, where Cegis2 is nearly four times faster than Cegis1.

4.4 Memory Usage

Lastly, we want to analyze the maximum memory usage of all of our implementations.

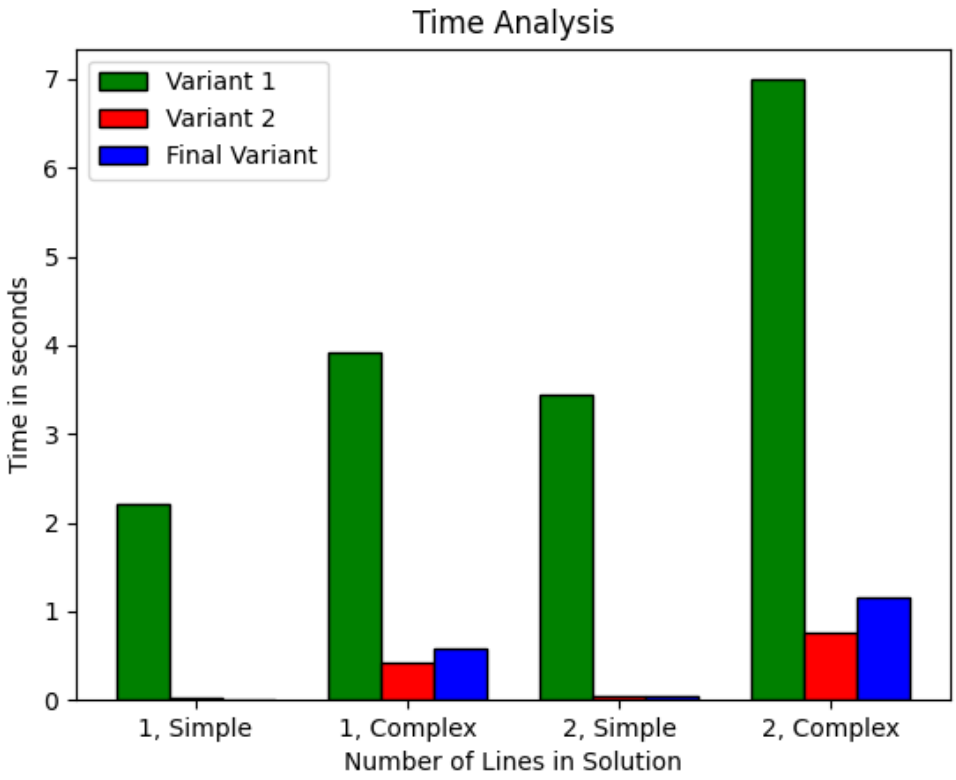

Figure 3: Comparison between the maximum memory usage observed by running the CEGIS and simple enumeration algorithms on the first four example groups defined in 4.1.1.

As we can see, Cegis0 uses noticeably more memory, which can be attributed to the fact that it generates all possible candidates for a given instruction length at once. All other methods make use of a generator, meaning they do not need to hold a large list in memory. Otherwise, the results are very similar for all methods. We do observe that Cegis1 performs slightly better than all others for instruction lengths lower than three, but for the example of length three reaches nearly double the maximum memory consumption compared to Cegis2. This mirrors our findings in the runtime analysis of the different CEGIS variants.

Expression	**Cegis0**	**Cegis1**	**Cegis2**	**Enumeration**
3	755.473	181.258	200.258	206.008
x + 10 - 5	1085.246	181.258	204.508	206.258
x * 1	1086.430	187.133	205.383	206.258
Average	**975.383**	**183.883**	**203.383**	**206.175**
x * 4	1145.574	262.063	321.766	354.098
x % y	1589.355	294.570	353.309	354.098
x / 4	1168.172	359.695	353.973	354.098
Average	**1301.367**	**305.776**	**343.683**	**354.098**
(x + 2) * 4	1251.723	326.215	323.215	322.301
x * 2 + 20	1229.371	325.215	323.215	322.301
x + y + 1	1640.340	324.215	322.301	322.301
Average	**1373.811**	**325.215**	**322.910**	**322.867**
(((x + 3)* 4) - 1) * 2	1229.469	331.137	387.992	400.270
x % 3	1229.516	331.117	400.273	398.273
x / 3	1229.387	387.992	400.273	398.273
Average	**1229.457**	**350.416**	**396.846**	**398.938**
(x / 3) + 3	Error	994.340	549.660	Error

Table 5: Comparison between the maximum memory usage observed by running the CEGIS and simple enumeration algorithms on the example groups defined in 4.1.1.

5 Conclusion

5.1 Results

In this work, we successfully implemented a superoptimizer for RISC-V assembly representing arithmetic expressions. We achieved this by using synthesis, either with or without CEGIS. We can successfully synthesize instruction sequences with a length of three or less. In "Synthesis of Loop-free Programs" [14], Gulwani et. al. claim that Superoptimizers are "amenable to only discovering optimal instructions of length four or less" (p.11), which we are quite close to, albeit with a much more restricted search space compared to superoptimizers that can produce any type of straight line instruction sequence as their output. The papers for Souper and Minotaur do not explicitly mention a limit for the length of optimized sequences they can synthesize; however, the authors do mention that they only use instruction sequences up to a length of five as their input to optimize [19] [16]. Given the more limited goals of this work, we consider our implementation a successful exploration of the concepts used in superoptimization.

Still, our enumeration-based approach would not be feasible for real life application. We also do not provide meaningful interfacing options between our work and compilers, meaning the current state of our implementation is more of a proof of concept.

Because we built a framework for working with RISC-V in Python and provided a starting point for implementing synthesis, it would also be possible to extend the ideas presented in this paper further. We discuss some possibilties for this in the next section.

5.2 Possible Future Extensions

Firstly, we could optimize some of the techniques we used. One big thing would be to adjust the synthesis part of the CEGIS loop. We achieve a speedup over a purely enumerative system because we avoid having to verify each possible sequence on all possible inputs, but we still check every possible solution manually in some capacity.

In contrast, a method introduced by Sumit Gulwani encodes the synthesis of a candidate as a single call to the solver [14]. This is called component-based CEGIS, and programs like Souper adapt a similar approach.

We could also try to expand the types of expressions that can be synthesized. If we want to stay close to the concept of arithmetic expressions, we could add more boolean operators like `and`, `or` and `xor`. This would be useful, as optimizations using these operators are also not obvious to find. Of course it would also be possible to extend our RISC-V DSL to support completely different types of RISC-V instructions.

Lastly, we could extend this work with a slicer, which could extract snippets of code to optimize from a larger code fragment. This could be a step to making our superoptimization more practically applicable in areas such as peephole optimization.

Bibliography

[1] cse291-program-synthesis. `https://github.com/nadia-polikarpova/cse291-program-synthesis`. Accessed: 2023-08-24.

[2] Gnu superoptimizer documentation. `https://www.gnu.org/software/superopt/`. Accessed: 2023-08-23.

[3] An introduction to assembly programming with risc-v. `https://riscv-programming.org/book/riscv-book.html`. Accessed: 2023-08-27.

[4] The risc-v instruction set manual volume i: User level isa. `https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf`. Accessed: 2023-08-27.

[5] Krste Asanović and David A Patterson. Instruction sets should be free: The case for risc-v. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146*, 2014.

[6] Sorav Bansal and Alex Aiken. Automatic generation of peephole superoptimizers. In John Paul Shen and Margaret Martonosi, editors, *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*, pages 394–403. ACM, 2006.

[7] Gilles Barthe, Juan Manuel Crespo, Sumit Gulwani, César Kunz, and Mark Marron. From relational verification to SIMD loop synthesis. In Alex Nicolau, Xiaowei Shen, Saman P. Amarasinghe, and Richard W. Vuduc, editors, *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, Shenzhen, China, February 23-27, 2013*, pages 123–134. ACM, 2013.

[8] Lingjiao Chen, Matei Zaharia, and James Zou. How is chatgpt's behavior changing over time? *CoRR*, abs/2307.09009, 2023.

[9] Stephen A. Cook. The complexity of theorem-proving procedures. In Michael A. Harrison, Ranan B. Banerji, and Jeffrey D. Ullman, editors, *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*, pages 151–158. ACM, 1971.

[10] Jacob Van Geffen, Luke Nelson, Isil Dillig, Xi Wang, and Emina Torlak. Synthesizing JIT compilers for in-kernel dsls. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II*, volume 12225 of *Lecture Notes in Computer Science*, pages 564–586. Springer, 2020.

[11] Torbjörn Granlund and Richard Kenner. Eliminating branches using a superoptimizer and the GNU C compiler. In Stuart I. Feldman and Richard L. Wexelblat, editors, *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI), San Francisco, California, USA, June 17-19, 1992*, pages 341–352. ACM, 1992.

[12] C. Cordell Green. Application of theorem proving to problem solving. In Donald E. Walker and Lewis M. Norton, editors, *Proceedings of the 1st International Joint Conference on Artificial Intelligence, Washington, DC, USA, May 7-9, 1969*, pages 219–240. William Kaufmann, 1969.

[13] Sumit Gulwani. Dimensions in program synthesis. In Roderick Bloem and Natasha Sharygina, editors, *Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2010, Lugano, Switzerland, October 20-23*, page 1. IEEE, 2010.

[14] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. *SIGPLAN Not.*, 46(6):62–73, jun 2011.

[15] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program synthesis. *Found. Trends Program. Lang.*, 4(1-2):1–119, 2017.

[16] Zhengyang Liu, Stefan Mada, and John Regehr. Minotaur: A simd-oriented synthesizing superoptimizer. *CoRR*, abs/2306.00229, 2023.

[17] Nadia Polikarpova. Loris D'Antoni. Program synthesis: The book. Accessed: 05.09.2023.

[18] Henry Massalin. Superoptimizer - A look at the smallest program. In Randy H. Katz and Martin Freeman, editors, *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II), Palo Alto, California, USA, October 5-8, 1987*, pages 122–126. ACM Press, 1987.

[19] Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Jubi Taneja, and John Regehr. Souper: A synthesizing superoptimizer. *CoRR*, abs/1711.04422, 2017.

[20] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In Vivek Sarkar and Rastislav Bodík, editors, *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2013, Houston, TX, USA, March 16-20, 2013*, pages 305–316. ACM, 2013.

[21] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In John Paul Shen and Margaret Martonosi, editors, *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*, pages 404–415. ACM, 2006.

[22] Peter Stone, Rodney Brooks, Erik Brynjolfsson, Ryan Calo, Oren Etzioni, Greg Hager, Julia Hirschberg, Shivaram Kalyanakrishnan, Ece Kamar, Sarit Kraus, Kevin Leyton-Brown, David C. Parkes, William H. Press, AnnaLee Saxenian, Julie Shah, Milind Tambe, and Astro Teller. Artificial intelligence and life in 2030: The one hundred year study on artificial intelligence. *CoRR*, abs/2211.06318, 2022.

[23] Haoye Tian, Weiqi Lu, Tsz On Li, Xunzhu Tang, Shing-Chi Cheung, Jacques Klein, and Tegawendé F. Bissyandé. Is chatgpt the ultimate programming assistant - how far is it? *CoRR*, abs/2304.11938, 2023.