EFFICIENT PREOPTIMIZATION SEQUENCES FOR POLLY

CHRISTOPH WOLLER

Bachelor's Thesis

Programming Group
Department of Informatics and Mathematics
University of Passau

Supervisor: Prof. Lengauer, Ph.D.

Tutor: Dr. Armin Größlinger
August 2014

ABSTRACT

The LLVM compiler framework comes with a plugin called Polly that enables polyhedral optimization on the LLVM intermediate representation (IR). The LLVM-IR code has to match a canonical form, called a SCoP (Static Control Part), such that Polly can transform it into the polyhedral model for later optimization. Consequently, the improvements Polly can achieve depend on the amount of code that is detected as SCoPs. LLVM analysis and transformation passes can increase the amount of code which is recognized to be a SCoP, but the optimal sequence of preoptimizations that maximizes the number of SCoPs is unknown.

This thesis discusses four different heuristic approaches, two genetic algorithms, a hill climbing algorithm, and a greedy algorithm, that all aim to generate good preoptimization sequences. We define an objective function as a measure for the quality of a preoptimization sequence. We compare the heuristic approaches with present preoptimization sequences and the effects of different optimization passes are examined. From the sequences found using heuristics we extract a partial order on the individual passes and by an exhaustive search on the possible topological sortings of this order find a new fixed preoptimization sequence for Polly that provides better results than the currently used preoptimization sequence.

ACKNOWLEDGMENTS

I have to thank the following people for their help throughout this thesis:

Dr. Armin Größlinger and Andreas Simbürger for supporting and guiding me through this thesis. All members of the Programming Group for providing helpful tips.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

LISTINGS

Part I

BACKGROUND AND FUNDAMENTALS

1

INTRODUCTION

This chapter explains the topic of this thesis and the motivation behind it. The first part of the chapter describes the basic structure of a compiler and introduces the phase-ordering problem in compilers. The second part points out the benefits of an examination of the phase-ordering problem with regard to Polly. The last part gives an overview of how this thesis is organized.

1.1 BASIC UNDERSTANDING OF A COMPILER

Modern computer programs are mostly written in high-level programming languages. On the one hand, this approach makes it easier for people to learn coding but, on the other hand, a computer does not understand high-level constructs. Hence, a program has to be translated into a representation understandable for a computer. This translation is done by compilers.

A compiler is a program that takes as input a program written in one language, the source language, and translates it into a program in another language, the target language, while preserving the semantic of the input program [7, p. 1], [13, p. 1]. The compilation process is a series of different phases. Aho et al. divide these phases in an analysis part and a synthesis part [7, p. 4f.]. The analysis part, or front end, of the compiler performs syntax and semantic analysis on the source program. The front end checks the consistency of the syntax and the semantic with the language definition. The front end informs the user if any phase detects an error in the source code. The analysis part ends with the generation of an intermediate representation of the code. The synthesis part or back end takes as input this intermediate representation and translates it into the target code. The single phases often share resources with each other. For example, the back end often requires information provided by previous analysis phases to create the target code. Each phase possibly works on a different representation of the source code. For example, the syntax analysis is often performed on a tree-like intermediate representation called syntax tree [7, p. 8]. So there may be different representations of the source code that are used during the compilation process. In this thesis such representations are also called intermediate representations, or short IR, of the source program. Figure 1 shows the basic structure of a translation process. The different phases take as input different intermediate representations which are illustrated by the *IR_x* labels on the arrows, where *x* is the number of the respective IR.

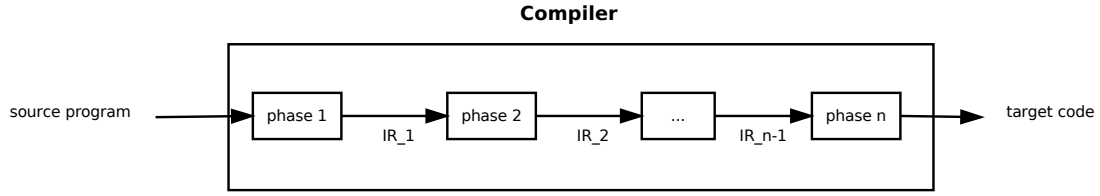

Figure 1.: A basic compiler

The division of the translation process into phases helps to understand the logical structure of a compiler, but implementations often group activities from different phases to a single pass that reads an input file and writes an output file [7, p. 11]. For example, the analysis phases may be combined to one pass or the generation of the target program is an own pass. Modern compiler frameworks such as LLVM are built around a well specified, low-level intermediate language and provide a large set of tools and passes that work on this IR. If you have a set of n source languages that should be transformed to a target language, you only have to implement n front ends that translate the source languages into the intermediate language and one back end that translates the intermediate language into the target language. This is a huge advantage over writing a back end for each source language because the translation of a source language directly into a target language is often more difficult than the detour over an intermediate representation and you do not have the overhead of n different back ends.

OPTIMIZATIONS Aho et al. describe the opportunity of performing code optimizations on the intermediate representation prior to the generation of the target code [7, p. 10]. So, an optimization phase takes as input an intermediate representation of the source program, performs analyses and/or code transformations, and provides the possibly modified representation as output. An optimization is often implemented as a pass of its own or combined with other optimizations to a single pass. In this thesis, an optimization is a transformation or an analysis that aims to improve the resulting target code, for example regarding execution speed, code size, or energy efficiency. The terms optimization, optimization phase, and optimization pass are used interchangeably. Most modern compilers come with a large set of optimizations. Users of compilers such as *clang* (see [4]) or *gcc* (see [5]) have the opportunity to enable different optimizations by setting the corresponding command-line flags.

1.2 PHASE-ORDERING PROBLEM

As explained in the previous section, modern compilers such as *clang* or *gcc* come with a large set of optimizations and the user of the compiler can enable an optimization by setting the corresponding command-line flag. Assume the following scenario. There are n different command-line flags available. Each of these flags enables a distinct optimization phase. The user of the compiler has a source code file and wants to translate it into some target language. Additionally, the user wants to get the best performing target program that is possible. Which of the n optimization flags should the user set? In which order should the user set the flags? Should he set a certain flag more than once? These questions lead to the problem of finding the best sequence of optimization phases for a given application which is better known as the *phase-ordering problem* in compilers [14, p.1]. The aspects that contribute to the complexity of this problem are summarized in the following list:

- A compiler may provide n optimization phases, but not all of these phases have to improve the resulting target code.
- Research has shown that one optimization phase can increase/decrease the opportunities for the next phases. So optimizations are dependent on other phases [14, p. 1].
- Some compilers allow the repeated application of optimization phases.
- The search space of possible optimization sequences is very large. Assume a compiler supports n different phases. When looking for an optimization sequence containing m phases, one can select between n^m different sequences.

Due to the complexity of the phase-ordering problem, many compilers offer predefined, fixed optimization sequences that can also be used via command-line flags. Probably the most prominent example is the *-O3* flag provided by compilers such as *clang* and *gcc*. These predefined sequences are meant to be a good solution for each application. However, research has shown that a single sequence of optimizations does not produce the optimal results for the infinite number of different programs whether compiling for speed, for space, or for other metrics [8], [9], [14], [19]. Consequently, scientists have studied the phase-ordering problem in more detail. A total exploration of the search space is unfeasible due to the number of possible optimization sequences. Therefore, researchers have presented different heuristic approaches meant to solve the phase-ordering problem. The heuristics often evaluate only a portion of the search space and, consequently, are not able to provide any guarantees about the quality of the solutions obtained [14, p. 1]. But even if this methods do not provide the best possible optimization sequence, experiments showed that they find yet effective ones [8], [9], [14], [19]. For example in [9] a genetic algorithm is used to find an optimization sequence that generates the smallest possible target code for an application. In [19] a custom sequence of flags, which should optimize the performance of a program, is created with a statistical approach.

1.3 MOTIVATION

Most of the heuristic approaches that generate custom optimization sequences have been used only for finding sequences which optimize for the performance, the size or the energy efficiency of the resulting target code up to now [8], [14]. In view of the positive experiences with heuristic approaches to the phase-ordering problem you can assume that heuristic approaches are also able to generate good preoptimization sequences for other metrics.

Polly (see [6]) is an infrastructure for polyhedral optimization in LLVM. It takes as input a program in the intermediate representation of LLVM (LLVM-IR) and searches for parts of the application which can be optimized. These parts are called Static Control Parts (SCoPs) and the amount of detectable code depends on the LLVM-IR. On the other hand, the LLVM-IR depends on the sequence of optimizations previously performed on it. But which optimizations should be applied to the LLVM-IR, so that the amount of code that can be detected as SCoPs is as high as possible? This question directly leads again to the phase-ordering problem described in the last section. Even though Polly comes with an own pass that can be used to prepare the LLVM-IR code, the last section explained that preoptimization sequences generated by heuristic approaches are able to provide better results than a fixed preoptimization sequence does. Consequently, custom generated preoptimization sequences might be useful to gather information about optimization passes and this information could help to create fixed preoptimization sequences that are better than current fixed sequences.

This thesis examines if heuristic approaches to the phase-ordering problem supply custom preoptimization sequences that are better than the fixed sequence of Polly. In this context, better means a higher amount of SCoPs. Beyond that, optimization sequences are examined in detail to get information about which optimization passes are useful for preoptimization and which are not. The gathered information about the optimization passes is then used to create a new fixed preoptimization sequence that is better than the current preoptimization sequence of Polly.

1.4 OUTLINE OF THE THESIS

The next chapter lays the foundation for the second part of the thesis. The used compiler framework LLVM is presented and Polly is explained. The latter includes some basic definitions from the context of Polly's SCoP detection.

The second part of the thesis covers the four heuristic approaches used, two genetic algorithms, a greedy algorithm, and a hill climbing algorithm. Furthermore, the second part presents the performed experiments and discusses the gathered results.

2

POLLY

2.1 LLVM FRAMEWORK

LLVM is a compiler framework first developed by Lattner [15], [16]. The idea Lattner realized with LLVM was and still is to enable program optimizations at compile time, link time, and run time [16, p. 1]. Consequently, the core of LLVM is an intermediate representation, called LLVM-IR, that is used by all brought analysis and transformation passes. LLVM-IR can be used in three different forms. There is an in-compiler IR, a human readable assembly language called LLVM assembly or LLVM bytecode, and an on-disk bitcode representation simply called LLVM bitcode [3]. The file extension for LLVM bytecode files is usually ".ll" and ".bc" is the usual one for LLVM bitcode. The LLVM Core libraries implement various parts of a compiler that operate on the LLVM intermediate representation. One can use these libraries to build a new compiler or embed them into existing ones. Additionally, LLVM comes with different command-line tools that makes it possible to directly use library functionality [2]. For each tool one can set different command-line flags that induces the tool to run corresponding LLVM passes.

2.1.1 *Compilation with LLVM*

An easy way to explain the architecture of LLVM is on the basis of an exemplary compilation that uses the LLVM tools. The example is derived from the one presented in [10, p. 8] and is depicted in Figure 2. A full description of the available LLVM tools is presented in [2]. The example contains two source files written in the programming language C. First, the source files must be translated into LLVM-IR by a language specific front end. In the example, *clang* is used to translate the source files into LLVM bytecode. The results are written to corresponding LLVM bytecode files. In the next step the tool *opt* is used to perform optimizations on the LLVM-IR files. The *llvm-link* tool combines the optimized LLVM-IR files to a single module which can again be optimized by *opt*. The resulting optimized LLVM-IR file is then compiled into assembly language for a specified architecture by *llc*. The tool *llvm-mc* takes this output and creates native assembly code. In the last step a system linker generates the native executable.

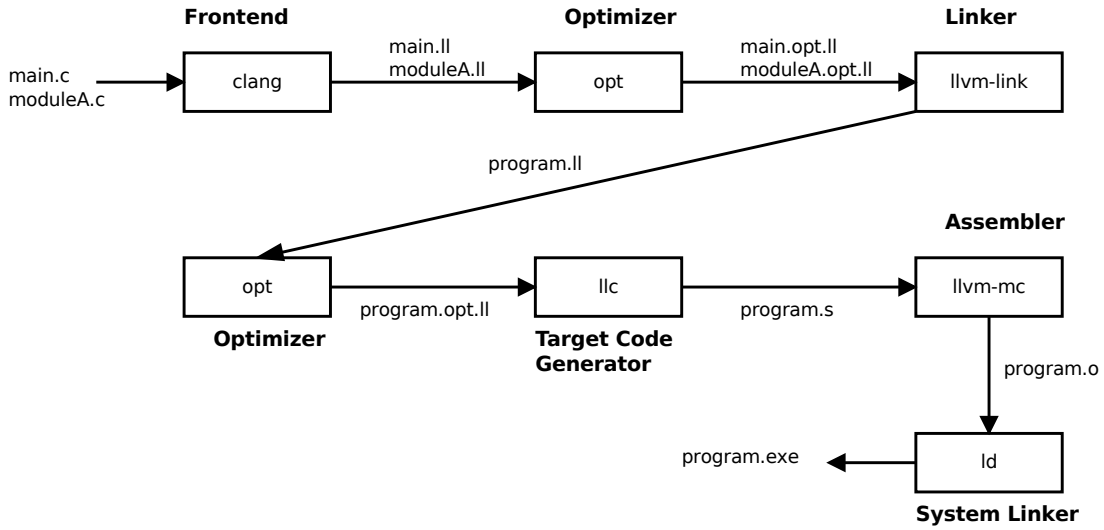

Figure 2.: Compilation with LLVM

2.1.2 *LLVM Command-line Tool opt*

This section discusses the LLVM command-line tool *opt* in more detail because it is used all over the thesis. The tool *opt* is the modular optimizer and analyzer of LLVM [2]. It reads in a LLVM-IR file and performs the specified analyses and/or transformations on it. One can specify analysis and transformation passes by setting the corresponding command-line flags. The available passes can be divided into three different categories. There are the analysis passes which for example compute information that is available to other passes. The transformation passes change the program in some way and are able to use information provided by analysis passes. The last category is the one of the utility passes. These passes provide functionality like writing a module to an LLVM bitcode file.

2.2 POLYHEDRAL OPTIMIZATION IN LLVM

Polly is a framework for LLVM which optimizes for data locality and parallelism using polyhedral techniques [12, p. 4f.]. Polly works on the intermediate representation of LLVM (LLVM-IR) which enables language independent optimizations. It is implemented as a set of LLVM passes which can be divided up into front end, middle end, and back end passes. Figure 3 shows the architecture of Polly.

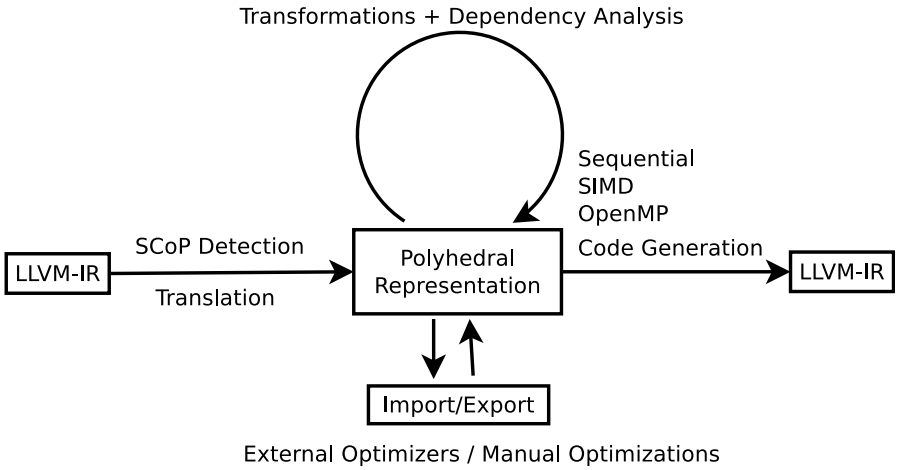

Figure 3.: Architecture of Polly [11, p. 4]

The division into front end, middle end, and back end passes already indicates that Polly uses a three-step approach for the optimization:

The front end passes take a program/code in LLVM-IR and are responsible for detecting parts of the code that Polly can analyze and optimize [12, p.5]. These parts are called Static Control Parts (SCoP) and are explained in the next section. Furthermore the front end translates these SCoPs into a polyhedral representation. For the sake of simplicity, Polly detects only SCoPs that match a canonical form. Hence, the code is usually canonicalized before the detection begins.

The dependency analysis and the polyhedral optimization take place in the middle end. Optimizations on the polyhedral representation can be performed in two different ways. First, there are optimizations offered by Polly itself. In addition the middle end is able to export the polyhedral representation, that can then be manually optimized or loaded into an external optimizer. Afterwards, the polyhedral representation can again be reimported.

Finally, the back end passes generate new LLVM-IR code from the polyhedral representation. While generating, Polly replaces parallel loops with OpenMP parallel loops or SIMD instructions. The new code replaces the old LLVM-IR.

Polly can be used in two different ways [12, p.5f.]. The first possibility is to run certain passes of Polly individually with the LLVM tool *opt*, which can run an arbitrary sequence of passes on a provided LLVM-IR file. In this way only specific parts of Polly can be used,

for example the SCoP detection. The other possibility is to use Polly as an integrated part of a compiler. Polly can be loaded into *clang* and, if loaded, a predefined sequence of Polly passes is automatically run when compiling with the option *-O3*. These passes apply polyhedral optimizations during the normal compilation of a program.

2.3 BASIC DEFINITIONS

We give some basic definitions before describing what parts of the program are recognized by Polly as SCoPs. The following definitions are based on the ones provided in [10, p. 16f.]. For all definitions assume a control flow graph (CFG) G of an arbitrary program with $G = (V, E)$, where V represents the set of basic blocks and E is the set of directed edges representing jumps in the control flow. As an example we use the code in Listing 1. The code represents a simple method containing a single for-loop and an if-condition with two statements S_1 and S_2.

```
void f(int* A) {
        for (int i = 0; i < 100; i++) {
                if (i < 100 / 2)
S1:                     A[i] = 1 + 2 + 3;
                else
S2:                     A[i] = i * 2;
        }
}
```

Listing 1: Simple example program

Definition 2.3.1 (Dominance Relation) *Let $i, j \in V$ of CFG G. Basic block i dominates basic block j, iff every path which passes through j must also pass through i. The basic block i is then called the dominator of j. It is called immediate dominator of j, if there exists no other basic block that is dominated by i and that also dominates j.*

Definition 2.3.2 (Post-Dominance Relation) *Let $i, j \in V$ of CFG G. Basic block j post-dominates basic block i, iff every path that passes through i must also pass through j. The basic block j is then called the post-dominator of i. It is called immediate post-dominator of i, if there exists no other basic block that is post-dominated by j and that also post-dominates i.*

Definition 2.3.3 (Simple Region) *A subgraph R of G, $R = (V_R, E_R) \subseteq G = (V, E)$, is called simple region, iff two basic blocks i, j, with $i \in V_R$ and $j \in V \setminus V_R$, exist such that i dominates every basic block of R and j post-dominates every basic block of R. The content of R does not influence the control flow outside of R and, consequently, R can be modeled as a function call, that can be replaced with an optimized version of it.*

Definition 2.3.4 (Canonical Simple Region) *A simple region R is a canonical simple region, iff there are no two adjacent simple regions R_1 and R_2, such that $R = R_1 \cup R_2$ holds.*

Definition 2.3.5 (Contains Relation for Simple Regions) *A simple region R contains another simple region R', iff its nodes are a superset of the nodes of the other simple region, brief if $R' \subseteq R$.*

Definition 2.3.6 (Region Tree/Program Structure Tree) *A region tree, also called refined program structure tree, is a tree of simple regions connected by the contains relation, see Definition 2.3.5.*

Definition 2.3.7 (Region) *A subgraph* R *of* G, $R = (V_R, E_R) \subseteq G = (V, E)$, *is called region iff it is not necessarily a simple region, but can be transformed into a simple region.*

Definition 2.3.8 (Natural Loop) *A subgraph* $L = (V_L, E_L)$ *of the CFG* $G = (V, E)$, $L \subseteq G$, *is called a natural loop of an edge* $(a, d) \in E$, *if* $d \in V_L$ *and if all basic blocks* $b \in V$ *which can reach* a *(including* a *itself) without passing through* d, *are in* V_L. *The basic block* d *is then called the loop header.*

Figure 4 shows the corresponding CFG for the code in Listing 1. The rectangles represent the basic blocks and the different colors highlight the different regions. The green region forms a natural loop as defined in Definition 2.3.8 and the vertex "for.body" is the loop header.

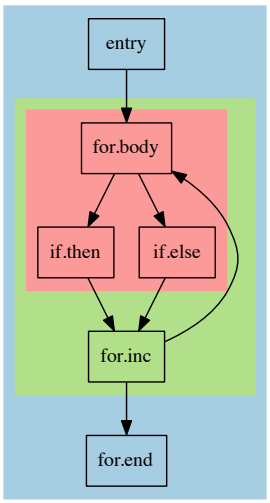

Figure 4.: The CFG for the code in Listing 1 with marked regions

2.4 STATIC CONTROL PARTS IN LLVM-IR

After clarifying some definitions we can proceed with examining how Polly recognizes parts of the program code as SCoPs. The previously described working method of Polly can be summarized as follows:

Polly accepts an LLVM-IR file as input and searches for Static Control Parts (SCoPs) [12, p.6]. The detected SCoPs are translated into a polyhedral representation and on this abstraction of the code Polly performs its transformations and optimizations. After the transformations new LLVM-IR code is generated from the polyhedral representation.

So the interesting question is, how must code look like to be detected as a SCoP? A very basic example of program code that represents a SCoP is provided by the example

code in Listing 1. The loop header consists of a single integer induction variable that is incremented from a lower to an upper bound by a constant stride of one and the bounds of the induction variable are both affine expressions. The condition of the if-statement is a comparison of two affine expressions. The two statements are both assignments of the results of an expression to an array. The array accesses are also affine expressions.

A common way to detect SCoPs in a function is to use pattern matching on a high-level abstract syntax tree (AST) representation of the code. A definition of a SCoP that is based on the AST representation of a program is taken from [12, p. 6f.]:

Definition 2.4.1 (Static Control Parts (SCoP)) *A Static Control Part (SCoP) is a part of the program which fulfills the following constraints:*

- *The part of the program contains only for-loops, while-loops, and/or if-statements and no other control flow structures.*
- *Each loop has a single integer induction variable. This variable is incremented from a lower to an upper bound by a constant stride of one.*
- *Upper and lower bounds are affine expressions of surrounding loop induction variables and parameters.*
- *The condition of an if-statement is a comparison of the values of two affine expressions of surrounding loop induction variables and parameters.*
- *All statements are assignments of an expression to an array element. There are only side effect free operators and/or function calls in the expression. The operands and arguments of a function call are induction variables, parameters, or array elements.*
- *Array indexes are affine expression of surrounding loop induction variables and parameters.*
- *Parameters in expressions are integer variables and are not modified inside the SCoP.*

The drawback of pattern matching on an AST representation is that the code must have a certain syntactic representation to get recognized as a SCoP. The large number of possible syntactical representations of a specific semantic makes this approach very complex. Additionally, the AST approach is programming language specific.

Therefore, Polly takes another way to detect SCoPs. It looks directly for SCoPs in the LLVM-IR of the program. This approach is programming language independent. LLVM-IR is a very low-level language that has no high-level constructs like loops or if-statements. Instead, there are jumps and goto statements. Arrays are represented by pointers. For a full description of LLVM intermediate representation the reader is referred to [3]. Polly takes the LLVM-IR of a program and uses a region-based approach to detect SCoPs available in a function [11, p. 2]. Polly searches in every function for the maximal regions that are SCoPs. It starts with the outermost simple region (Definition 2.3.3) and searches for canonical simple regions (Definition 2.3.4) in the region tree (Definition 2.3.6) that are SCoPs. If a canonical simple region is a SCoP, Polly stores it. Otherwise, Polly checks all its children. The result of this analysis is a set of canonical

simple regions that are SCoPs. In the last step of the detection, Polly combines these regions to larger non-canonical regions (Definition 2.3.7). Consequently, the final result is a set of non-canonical regions that are SCoPs. A canonical simple region is recognized as SCoP if the low-level code in the simple region is semantically equivalent to a SCoP, as in Definition 2.4.1, written in a high-level language. Listing 2 contains the LLVM-IR representation of the code from Listing 1 and represents a SCoP in LLVM-IR.

```
define void @f(i32* %A) #0 {
entry:
  br label %for.cond

for.cond:                                         ; preds = %for.inc, %entry
  %i.0 = phi i32 [ 0, %entry ], [ %inc, %for.inc ]
  %cmp = icmp slt i32 %i.0, 100
  br i1 %cmp, label %for.body, label %for.end

for.body:                                         ; preds = %for.cond
  %cmp1 = icmp slt i32 %i.0, 50
  br i1 %cmp1, label %if.then, label %if.else

if.then:                                          ; preds = %for.body
  %idxprom = sext i32 %i.0 to i64
  %arrayidx = getelementptr inbounds i32* %A, i64 %idxprom
  store i32 6, i32* %arrayidx, align 4
  br label %if.end

if.else:                                          ; preds = %for.body
  %mul = mul nsw i32 %i.0, 2
  %idxprom2 = sext i32 %i.0 to i64
  %arrayidx3 = getelementptr inbounds i32* %A, i64 %idxprom2
  store i32 %mul, i32* %arrayidx3, align 4
  br label %if.end

if.end:                                           ; preds = %if.else, %if.then
  br label %for.inc

for.inc:                                          ; preds = %if.end
  %inc = add nsw i32 %i.0, 1
  br label %for.cond

for.end:                                          ; preds = %for.cond
  ret void
}
```

Listing 2: A SCoP in LLVM-IR

To specify this more clearly we give a definition of a SCoP in LLVM-IR that conforms to Definition 2.4.1 of a SCoP. The definition is derived from [12, p.8].

Definition 2.4.2 (SCoP in LLVM-IR) *A SCoP in LLVM-IR is a subgraph of the CFG, which fulfills the following constraints:*

- *The subgraph forms a simple region as defined in Definition 2.3.3.*
- *It is semantically equivalent to a syntactically defined SCoP (see Definition 2.4.1).*

With the SCoP in Listing 2 in the back of our minds, we can explain how Polly verifies that a simple region in this LLVM-IR matches Definition 2.4.2 of a SCoP. Polly checks if the simple region complies with the following constraints which are based on the ones listed by Grosser in [10]. The constraints are adjusted to Polly's current state of development.

- The control flow in the simple region only consists of perfectly nested conditions and natural loops (Definition 2.3.8). All if-statements with a then-branch and an else-branch are interpreted as perfectly nested conditions. Among other things, Polly uses for the recognition of a natural loop an analysis pass provided by LLVM. A loop must have a single integer induction variable that is updated from a lower to an upper bound by a constant stride of one. Unconditional branches are also allowed in addition to conditional branches.

- The condition of a conditional branch has to be an integer comparison between two affine expressions of surrounding induction variables and parameters or the condition must be constant. Furthermore, the lower bound of a loop has to be zero and the upper bound has to be an affine expression of surrounding induction variables and parameters. To verify this, Polly uses the natural loop analysis and the scalar evolution analysis of LLVM to receive a closed form expression for the loop bounds and expressions of the comparison. A closed form expression is equivalent to an affine expression if it only contains integer constants, parameters, additions, multiplications with constants and add recurrences that have constant steps. Occurring parameters must be integer variables, must be defined outside the simple region, and are not allowed to be modified inside this simple region. If a loop cannot be analyzed and, therefore, its loop bounds cannot be assumed as affine, it cannot be part of a SCoP.

- Computational instructions, vector management instructions, and type conversion instructions in LLVM-IR are side effect free and can appear in a SCoP.

- Function calls are valid if LLVM can provide the information that the function call is side effect free, always returns, and does not access memory. At the current stage of development, function calls are not supported yet.

- Compiler internal intrinsics are not supported yet.

- Occurring PHI nodes must be in a canonical form. A PHI node is canonical if it has the form `phi i32 [0, %predecessor1], [%nextvalue, %predecessor]`. The result type need not be `i32`, it just has to be an integer.

- Exception handling instructions are not allowed in a SCoP because they describe control flow that cannot be modeled statically.

- Base addresses must refer to distinct memory spaces or they must be identical. A memory space in this context is the set of memory elements that can be accessed by adding an arbitrary offset to the base address. LLVM has alias analysis passes that provide Polly with the required information. The result of such analysis is

ether that two base addresses *must alias*, then the base addresses are identical, or that they do *not alias*, then there is no intersection of the memory spaces or that they *may alias*. Base addresses that *may alias* are not allowed in a SCoP.

- Every scalar variable referenced must either be defined in the basic block it is used, it must be a loop induction variable, or it must be defined outside the simple region. This ensures that no scalar dependences exist between two different basic blocks.
- Memory accesses are allowed as long as they behave like array accesses with affine subscripts.

2.5 PREPARING PASSES OF POLLY

For the rest of the thesis we use the LLVM tool *opt* to individually run the available LLVM and/or Polly passes. This section shows how to basically use *opt*, with Polly integrated, for SCoP detection and presents some passes that Polly uses to increase the amount of detectable code in an LLVM-IR file [12, p. 11]. In the examples provided below, *popt* represents *opt* with the Polly extension.

The command-line tool *opt* takes an arbitrary sequence of available flags that enable certain passes and a file containing the LLVM-IR code as input[1]. A basic call of the *opt* tool might look like as follows:

```
$ opt pass_sequence LLVM-IR_FILE
```

The variable `pass_sequence` represents the sequence of optimization passes to run. The variable `LLVM-IR_FILE` represents the LLVM-IR file on which the optimizations should be performed. If we want Polly to detect SCoPs in the provided LLVM-IR file, we have to set the flag *-polly-detect* like shown:

```
$ popt -polly-detect -stats LLVM-IR_FILE
```

The flag *-stats* tells *opt* to print out a statistical overview of the detection results. The detection flag induces *opt* to run several passes before it runs the actual detection pass. The results of the detection will be printed to the standard output. The automatically applied passes are listed in Table 1. The description of the passes is derived from [1].

Pass name	Pass description
-no-aa	Performs the default alias analysis of LLVM. It always returns "may alias" for alias queries.
-domtree	Starts a simple dominator construction algorithm for finding forward dominators.
-postdomtree	Starts a simple post-dominator construction algorithm for finding post-dominators.
-loops	Analyses natural loops.

1 For how to make Polly passes available in *opt* please see: `http://polly.llvm.org/example_manual_matmul.html`

-scalar-evolution	Can be used to analyze and categorize scalar expressions in loops.
-domfrontier	Starts a simple dominator construction algorithm for finding forward dominator frontiers.
-regions	Detects single entry single exit regions in a function.
-polly-detect	Detects SCoPs in the provided LLVM-IR code.

Table 1.: Passes induced by -polly-detect

Polly offers a pass that is meant to increase the amount of LLVM-IR code that is detected as SCoP. A point, which contributes to this, for example is the canonicalization of the loop bounds that possibly helps the scalar evolution analysis to provide better results. The pass can be run by setting the flag *-polly-canonicalize*:

```
$ popt -S -emit-llvm -polly-canonicalize LLVM-IR_file
```

The flags *-S* and the *-emit-llvm* tell *opt* to replace the resulting LLVM-IR code with the old one in the file. Otherwise, *opt* will print the new code to the standard output. The canonicalize pass of Polly induces *opt* to run several canonicalization passes. These passes are listed in Table 2 below. The description of the passes is derived from [1].

Pass name	Pass description
-domtree	Starts a simple dominator construction algorithm for finding forward dominators.
-mem2reg	Promotes memory references to be register references.
-instcombine	Combines instructions to form fewer, simple instructions.
-simplifycfg	Performs dead code elimination and basic block merging.
-tailcallelim	Performs tail call elimination.
-reassociate	Reassociates commutative expressions in an order that is designed to promote better constant propagation.
-loops	Analyses natural loops.
-loop-simplify	Performs several transformations to transform natural loops into a simpler form.
-lcssa	Transforms loops by placing phi nodes at the end of the loops for all values that live across the loop boundaries.
-loop-rotate	Performs a simple loop rotation.
-scalar-evolution	Can be used to analyze and categorize scalar expressions in loops.
-iv-users	Bookkeeping for “interesting” users of expressions computed from induction variables.
-polly-indvars	Simplifies induction variables.
-polly-prepare	Prepares code for Polly.

Table 2.: Passes induced by *-polly-canonicalize*

2.6 EXAMPLES THAT POLLY CANNOT HANDLE

In this section we discuss some examples in which Polly is not able to detect SCoPs for some reason. The examples should help us to get more familiar with the definition of SCoPs in LLVM-IR.

EXAMPLE 1 – NON SINGLE ENTRY IN A LOOP Polly cannot detect any SCoP in the C code shown in Listing 3 because the occurring loop has two entry blocks. This gets more obvious if we take a look at the LLVM-IR version in Listing 4 and at the CFG shown in Figure 5.

```
void f(long A[], long N) {
        long i;

        if (true)
                goto loop;
        else
                goto loop;

loop:
        for (i = 0; i < N; ++
            i)
                A[i] = i;
}
```

Listing 3: C code with non single entry in a for-loop

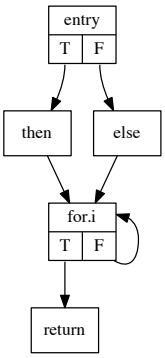

Figure 5.: The CFG for the code in Listing 3

```
define void @f(i64* %A, i64 %N) nounwind {
entry:
  fence seq_cst
  br i1 true, label %then, label %else

then:
  br label %for.i

else:
  br label %for.i

for.i:
  %indvar = phi i64 [ 0, %then ], [ 0, %else], [ %
      indvar.next, %for.i ]
  %scevgep = getelementptr i64* %A, i64 %indvar
  store i64 %indvar, i64* %scevgep
  %indvar.next = add nsw i64 %indvar, 1
  %exitcond = icmp eq i64 %indvar.next, %N
  br i1 %exitcond, label %return, label %for.i

return:
  fence seq_cst
  ret void
}
```

Listing 4: LLVM-IR code of code shown in Listing 3

EXAMPLE 2 – NON CANONICAL INDUCTION VARIABLE Polly cannot detect the SCoP in the C code shown in Listing 5 because the induction variable of the for-loop is not in a canonical form. It should be an integer variable and not a long variable.

```
void f(long A[], long N) {
        long i;
        for (i = 0; i < N; ++i)
        A[i] = i;
}
```

Listing 5: Source-code with non canonical induction variable

Listing 6 shows the corresponding LLVM-IR code of the code in Listing 5.

```
define void @f(i64* %A, i64 %N, i64 %p) nounwind {
entry:
  fence seq_cst
  br label %pre

pre:
  %p_tmp = srem i64 %p, 5
  br i1 true, label %for.i, label %then

for.i:
  %indvar = phi i64 [ 0, %pre ], [ %indvar.next, %for.i ]
  %indvar.p1 = phi i64 [ 0, %pre ], [ %indvar.p1.next, %for.i ]
  %indvar.p2 = phi i64 [ 0, %pre ], [ %indvar.p2.next, %for.i ]
  %sum = add i64 %indvar, %indvar.p1
  %sum2 = sub i64 %sum, %indvar.p2
  %scevgep = getelementptr i64* %A, i64 %indvar
  store i64 %indvar, i64* %scevgep
  %indvar.next = add nsw i64 %indvar, 1
  %indvar.p1.next = add nsw i64 %indvar.p1, %p_tmp
  %indvar.p2.next = add nsw i64 %indvar.p2, %p_tmp
  %exitcond = icmp eq i64 %sum2, %N
  br i1 %exitcond, label %then, label %for.i

then:
  br label %return

return:
  fence seq_cst
  ret void
}
```

Listing 6: LLVM-IR code of code shown in Listing 5

Part II

PREOPTIMIZATION SEQUENCES FOR POLLY

3

GENERATING CUSTOM PREOPTIMIZATION SEQUENCES

The last chapter presented Polly and showed how to use the *opt* tool to prepare an LLVM-IR file and run the SCoP detection on it. Like previously mentioned, Polly comes with a pass called *-polly-canonicalize* that can be used to prepare an LLVM-IR file for SCoP detection. The idea behind this preparation is to get the amount of code that can be detected as SCoP as high as possible. However, *-polly-canonicalize* contains optimization passes of which it is assumed that they have a positive effect on Polly's SCoP detection, but *-polly-canonicalize* has never been compared with other preoptimization sequences. Furthermore, there are no surveys that investigate which optimization passes are useful for preoptimization and which are not. So the best preparation sequence for a given LLVM-IR file is searched. This chapter describes this problem more precisely and presents heuristic approaches to it.

3.1 PROBLEM FORMULATION

Assume you have an LLVM-IR file that should be optimized by Polly. For this purpose you want to prepare the LLVM-IR code using an optimization sequence that modifies the code in a way Polly can detect a higher number of SCoPs. Consequently, all sequences that are worthy for considerations have to be evaluated to find the best among them. One could evaluate a sequence by measuring the runtime of the LLVM-IR code after applying this sequence for preoptimization and after optimizing the LLVM-IR code with Polly. This measure would point out how much LLVM-IR code could be optimized by Polly after applying a certain preoptimization sequence. The drawback of this approach is that such runtime checks would probably exceed the available time and another drawback is that running LLVM-IR code for runtime checks often requires test inputs. Hence, the evaluation of a sequence is as follows. First, the specified LLVM-IR file is optimized with this sequence. Then, Polly performs a SCoP detection on the optimized LLVM-IR. The number of detected SCoPs and the total number of regions is measured. The difference between these two numbers indicates how many regions are not SCoPs. This difference is used for the evaluation of a sequence. Thus, for a given LLVM-IR file, the best sequence is that which has the lowest difference. This measure is used because one wants SCoPs to cover as much of the LLVM-IR code as possible and this measure is better than the ratio of the number of SCoPs to the number of regions. Assume you have a certain LLVM-IR code and two preoptimization sequences a and b.

Let s_a be the number of SCoPs detected in the code using sequence a and let r_a be the corresponding number of regions in the code. Let s_b and r_b be the number of SCoPs and the number of regions, if sequence b is used. Let $s_a = 1$, $r_a = 2$ and the sequence b does nothing more than sequence a but duplicates the SCoP in the LLVM-IR code, then $s_b = 2$ and $r_b = 3$ holds. If the ratio is used, the sequence b would be better than the sequence a because $s_a/r_a = 1/2 < 2/3 = s_b/r_b$. This is false because b just duplicates already existing code. If the difference is used, the two sequences are equal because $r_a - s_a = 2 - 1 = 1 = 3 - 2 = r_b - s_b$ is true.

Since the length of the best sequence is unknown, one would have to work with variable sequence lengths. However, this would greatly increase the complexity of the problem. Therefore, often only sequences with a fixed length as possibly solutions are considered [14, p. 6]. The sequence length of the possible solutions is usually determined arbitrarily. To reduce the complexity of the problem this thesis also works with fixed sequence length.

The overall problem can be formulated more precisely as minimization problem:

Given:

1. A program p of the set P of programs in LLVM-IR that should be prepared for SCoP detection.
2. The set O that denotes the set of the available optimizations.
3. An arbitrary chosen sequence length $m \in \mathbb{N}$, where $\mathbb{N}$ denotes the naturally numbers including zero.
4. $S_m = \underbrace{O \times \cdots \times O}_{m \text{ times}}$, the set of all available optimization sequences with length m. This set is also called *search space*, while its elements are called *candidate solutions*.
5. The evaluation function:
   $v : P \times S_m \to \mathbb{N}, (p, s) \mapsto \mathtt{regions}(p, s) - \mathtt{scops}(p, s)$
   The function $\mathtt{regions}$ provides the number of regions in a program p if a sequence s is used for preoptimization and $\mathtt{scops}$ provides the corresponding number of detected SCoPs in this program.

Sought: An element s_0 in S_m such that $v(p, s_0) \leqslant v(p, s)$ holds for all s in S_m

For the remaining part of the thesis O denotes the set of available optimization passes, S_m denotes the search space of sequences of length $m \in \mathbb{N}$, and v is the evaluation function. The terms evaluation function and fitness function are used interchangeable. Consequently, the fitness value or just fitness of a sequence s for a program p refers to the value $v(p, s)$. It has to be noted that the fitness of a sequence depends on the program that is optimized with this sequence. Hence, the evaluation function v assigns

a sequence $s \in S_m$ a different fitness value depending on the program that is optimized with this sequence.

3.2 HEURISTIC APPROACHES

In order to solve the minimization problem described in the past section for a program p, one would have to perform an exhaustive exploration of the search space S_m with $m \in \mathbb{N}$. So one would have to calculate $v(p, s)$ for each $s \in S_m$ and compare the resulting evaluations of the candidate solutions with each other. Assume that the set O has n, with $n \in \mathbb{N}$, elements, then there would be n^m candidate solutions who would have to be evaluated and compared. In most cases, this is a too expensive task and because of that heuristic approaches are used to solve the phase-ordering problem [8], [9], [14], [19].

In this thesis, a heuristic is interpreted as problem specific information that allows to find the desired solution hopefully faster than an exhaustive search would do [20, p. 319]. With the problem specific information in mind, heuristic algorithms often evaluate only a portion of the search space. However, this means that there is no guarantee whether the found solution is near to the optimal solution or not. Fortunately, it has been shown that heuristic approaches are able to generate custom optimization sequences that are almost as good as the optimal one [8], [14]. In this thesis, a hill climbing, a greedy, and two genetic algorithms are used to generate custom preoptimization sequences for Polly. The choice of the algorithms is attributable to the investigations and performed experiments presented in [8], [9], [14], where good results were obtained with these algorithms.

3.2.1 *Hill Climbing*

Hill climbing is a local search technique. In general, a local search technique selects a random candidate solution s from the search space S_m as base solution [20, p. 327 f.]. The base solution is then modified to generate another candidate solution s'. If s' is better than s, s' becomes the new base solution. This process is repeated till there is no more improvement.

The special feature of hill climbing is that a hill climber does not only compare the base solution s with a single other solution, but it calculates all the neighbors of s and compares it with the best neighbor [8, p. 6]. The set of neighbors N_s of the base solution s contains all candidate solutions that differ from s at exactly one position. For example assume $O = \{a, b\}$ and let $m = 2$. Then $S_2 = \{(a, a), (a, b), (b, a), (bb)\}$ is the search space. If $s = (a, a)$ is the base solution, its neighbors are $n_1 = (a, b)$ and $n_2 = (b, a)$, but not $n_3 = (b, b)$, because this candidate solution differs from s at position 1 and 2. If the best neighbor is better than s, this neighbor becomes the new base solution. This process is repeated till there is no more improvement.

The hill climbing algorithm used in this thesis is derived from [8, p. 6] and is adjusted according to the proposals of [14, p. 7]. The hill climber is presented in Algorithm 1.

Algorithm 1 Hill Climbing

```
1: procedure HILLCLIMBER(p, m, O)
2:     ▷ LLVM-IR program p, Sequence length m ∈ ℕ, set of available optimizations O
3:     ▷ Keep track of the best sequence found
4:     solution ← ()
5:     iterations ← 0
6:
7:     while iterations < 100 do
8:         ▷ Start with a randomly selected sequence
9:         base ← random element (o_1, ..., o_m) ∈ S_m
10:        stop ← False
11:
12:        repeat
13:            ▷ Calculate the neighbors of base
14:            N_base ← ∅
15:            j ← 1
16:            while j ⩽ m do
17:                for all o ∈ O do
18:                    neighbor ← (base[1], ..., base[j − 1], o, base[j + 1], ..., base[m])
19:                    N_base ← N_base ∪ {neighbor}
20:                end for
21:                j++
22:            end while
23:
24:            ▷ Check if the best neighbor is better than the base sequence
25:            stop ← True
26:            for all neighbor ∈ N_base do
27:                if v(p, neighbor) < v(p, base) then
28:                    base ← neighbor
29:                    stop ← False
30:                end if
31:            end for
32:        until stop
33:
34:        if iteration == 0 ∨ v(p, base) < v(p, solution) then
35:            solution ← base
36:        end if
37:        iterations++
38:    end while
39:
40:    return solution                ▷ The best solution found in 100 iterations
41: end procedure
```

The problem with hill climbing is that such an algorithm is likely to get stuck in a local optimum and does not come closer to the global optimum. For example if the base solution s has reached the local optimum shown in Figure 6, it evaluates its neighbors and they are all worse solutions than s. Consequently, the hill climber gets stuck in this local optimum even if there are better solutions near the global optimum, see Figure 6. Whether the hill climber gets stuck in a local optimum or not, depends on the initial base solution that is selected randomly. In order to reduce the noise of this randomness and so reduce the probability to get a solution where the hill climber got stuck in a local optimum, the whole hill climbing process is repeated 100 times in Algorithm 1 like proposed in [14, p. 7].

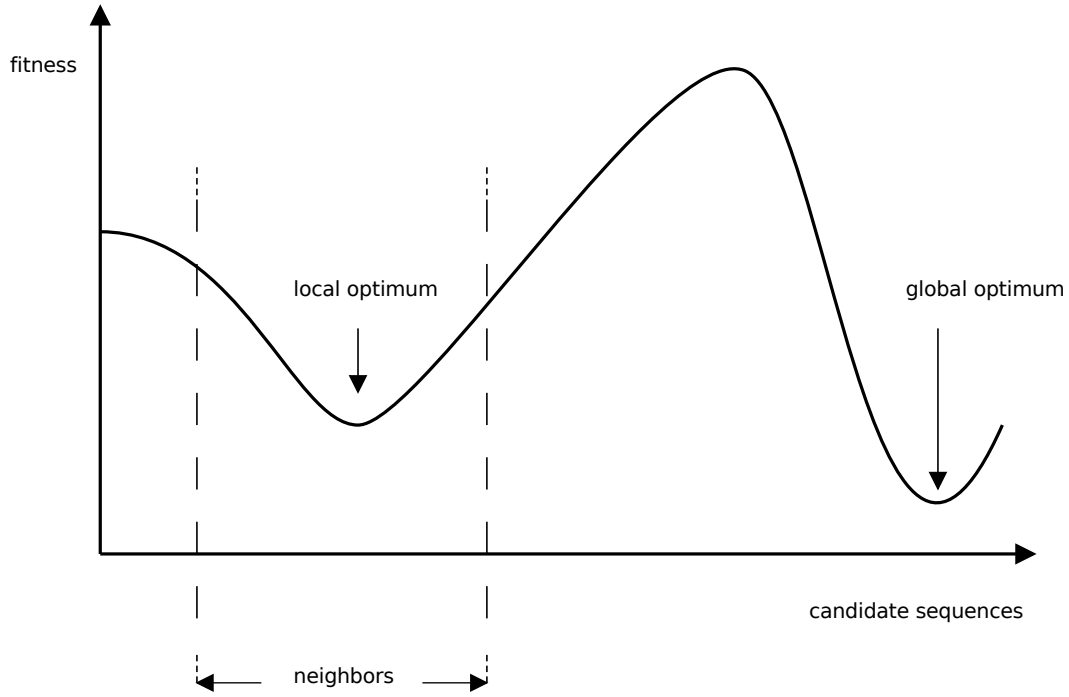

Figure 6.: Course of evaluation function v over the candidate sequences

Almagor, Cooper, et al. examined small search spaces and stated that 80% of the local optima are within 5 to 10% of the optimal solution [8, p. 1]. Furthermore, they pointed out that a search technique like multiple hill climbing runs should find good solutions because a search space contains a large number of local optima [8, p. 4]. Consequently, it is not fatal if a hill climbing algorithm gets stuck in a local optimum because this local optimum is likely near to the optimal solution. Hill climbing algorithms have already been compared with other search techniques and, indeed, a hill climber is able to find good solutions that are close to the optimum [8], [14]. So the hill climbing algorithm described above should be able to find good solutions that are close to the optimal one and that are better than *-polly-canonicalize*.

3.2.2 *Greedy Algorithm*

Greedy algorithms are often used for optimization problems because they obey a simple paradigm [20, p. 324]. A greedy algorithm makes the next step based on locally available information. Therefore, the step is chosen that promises the highest profit if a solution for a maximization problem is looked for or the step that promises the lowest costs if a solution for a minimization problem is looked for. The idea behind this is to reach the optimal solution if the best possible step is made at each point in time.

Kulkarni et al. have compared the results of a greedy algorithm with the results of other heuristic approaches and have come to the decision that a greedy algorithm provides efficient/good solutions, but they are not as good as the ones provided by other techniques like genetic algorithms or hill climbing algorithms [14]. Nevertheless, the custom sequences found by the greedy algorithm described in [14, p. 9] are still efficient ones and are in most cases better than fixed optimization sequences. Hence, this greedy algorithm is adapted to the problem described in Section 3.1.

Assume a custom optimization sequence of length $m \in \mathbb{N}$ for a program p is wanted and there are $n \in \mathbb{N}$ optimization passes available, so $|O| = n$. The greedy algorithm starts with the empty sequence as initial base sequence b. In the first step the algorithm generates all n sequences of length one , so all sequences $s \in S_1$. The sequences generated out of the base sequence are called its *child sequences*. The algorithm evaluates all child sequences according to the familiar evaluation function v and chooses the sequences that have the lowest evaluation value, so all child sequences s with $v(p, s) \leqslant v(p, s')$ for all $s' \in S_1$, and selects randomly one of them as new base sequence b. In the next step the algorithm creates for each optimization $o \in O$ two new sequences s_1 and s_2 by appending o to the base sequence, $s_1 = b \| o$, and by prepending o to the base sequence, $s_2 = o \| b$. That way, $2n$ new sequences are generated. The sequences are again evaluated and the new base sequence is selected like in the first step. This process is repeated till the greedy algorithm gets to a base sequence of length m. This base sequence represents the found custom optimization sequence. Like proposed in [14, p. 9] the algorithm is repeated 100 times to reduce the noise that may be caused by the random point at the selection of the next base sequence. Algorithm 2 shows the presented greedy algorithm as pseudo-code.

Algorithm 2 Greedy Algorithm

```
1: procedure GREEDY(p, O, m)
2:     ▷ LLVM-IR program p, available optimizations O, desired sequence length m
3:     solutions ← []                ▷ Found solutions are at the beginning an empty list
4:     iterations ← 0
5:
6:     while iterations < 100 do
7:         base ← ()                          ▷ Start with an empty base sequence
8:
9:         repeat
10:            ▷ Generate child sequences by appending and prepending
11:            children ← ∅
12:            for all o ∈ O do
13:                child_a ← base‖o
14:                child_p ← o‖base
15:                children ← children ∪ {child_a, child_p}
16:            end for
17:
18:            ▷ Select the children with the lowest evaluation value
19:            best ← ∅
20:            for all child ∈ children do
21:                if ν(p, child) ⩽ ν(p, child′) for all child′ ∈ children then
22:                    best ← best ∪ {child}
23:                end if
24:            end for
25:
26:            ▷ Select new base sequence randomly
27:            base ← random child from best
28:        until length(base) == m
29:
30:        solutions.append(base)
31:        iterations++
32:    end while
33:
34:    sort solutions according to fitness from high to low
35:    solution ← return last element in list solutions
36:    return solution                ▷ The best solution found in 100 iterations
37: end procedure
```

3.2.3 *Genetic Algorithms*

John Holland from the University of Michigan invented genetic algorithms in the 1960s [17, p. 3]. Genetic algorithms are used, among other things, for finding solutions for optimization problems. The central idea behind genetic algorithms is that they take nature as example and imitate the genetic evolution [20, p. 331]. Melanie Mitchell explains with the following words why evolution is a good approach to optimization problems [17, p. 4] :

> "Evolution is, in effect, a method of searching among an enormous number of possibilities for 'solutions'."

Researchers have used genetic algorithms to generate custom optimization sequences whether optimizing for program speed, code size, or other metrics [8], [9], [14], [18]. Their experiments have shown that in the most cases the sequences generated by genetic algorithms provide better results than fixed optimization sequences. These results are therefore so impressive because genetic algorithms search only a portion of the search space for possible solutions. Cooper et al. have described a genetic algorithm that produces optimization sequences which aim to minimize the code size of a provided file. Their results are positive without exception. Hence, genetic algorithms are used in this thesis, in order to generate custom preoptimization sequences for Polly.

Before a concrete genetic algorithm can be described, it is necessary to define some basic terms [17, p. 3ff.].

Definition 3.2.1 (Gene and Gene Pool) *A gene* g *is a symbol that encodes some information. In this thesis, the set of available genes is called the gene pool* G.

Definition 3.2.2 (Chromosome) *A chromosome* c *consists of* n *genes, with* $n \in \mathbb{N}$. *In this thesis a chromosome* c *is interpreted as a sequence of genes with length* n, *so* $c = (g_1, g_2, ..., g_n)$ *with* $g_1, ..., g_n \in G$ *and* $n \in \mathbb{N}$.

Definition 3.2.3 (Locus) *Let* $c = (g_1, ..., g_n)$ *be a chromosome of length* $n \in \mathbb{N}$ *with* $g_1, ..., g_n \in G$. *The position of a gene on a chromosome is called its locus. In other words, the locus of a gene* g_l *in the sequence* c *is its index* l, *with* $1 \leqslant l \leqslant n$.

Definition 3.2.4 (Population) *A population* P *is a set of chromosomes that have the same length* $n \in \mathbb{N}$.

Definition 3.2.5 (Fitness function and fitness value) *Genetic algorithms include a fitness function* $f : G^n = \underbrace{G \times \cdots \times G}_{n \text{ times}} \to \mathbb{N}$, *with* $n \in \mathbb{N}$, *that assigns a chromosome* c *to its fitness value or short fitness* $f(c)$. *Chromosomes can be compared with respect to their fitness.*

A genetic algorithm moves from a population to a new one using operations called *selection*, *crossover*, and *mutation* [17, p. 7ff.]. The move from one population to a new one is called *generation*.

SELECTION This operator selects the chromosomes of the population P that are used for the reproduction according to their fitness value. Normally, the chromosomes with better fitness values are selected more likely because it is assumed that good chromosomes produce good offsprings.

CROSSOVER Assume there are two chromosomes $c_1 = (a_1, a_2, ..., a_n)$ and $c_2 = (b_1, b_2, ..., b_n)$. The crossover operation creates two offsprings by combining the genes of c_1 before the locus l with genes of c_2 after the locus l and vice versa. Consequently, the resulting offsprings are $o_1 = (a_1, ..., a_l, b_{l+1}, ..., b_n)$ and $o_2 = (b_1, ..., b_l, a_{l+1}, ..., a_n)$.

MUTATION The mutation exchanges a gene of a chromosome with a randomly chosen gene of the gene pool. A mutation can happen at each locus. For example, assume there is a chromosome $c = (a, b, c)$. A mutation at the second locus exchanges the gene b with gene d. The result of the mutation is the chromosome $c' = (a, d, c)$.

In the context of the problem defined in Section 3.1, the gene pool is the set of available optimizations, so $G = O$, while each gene represents an optimization. Consequently, a chromosome is an optimization sequence. Therefore, each chromosome c is a candidate solution and element of the search space S_m. The evaluation function v is the fitness function. Consequently, there is an own fitness function f for each LLVM-IR program p that should be optimized because the value of v for a sequence depends on p. It should be noted that in this problem, the chromosome with the lowest fitness value is the best.

The first genetic algorithm that is used in this thesis is adapted from [9] with adjustments according to the domain of Polly. This genetic algorithm starts with a population P of 20 chromosomes of size n. These chromosomes are randomly chosen from the set G^n, where G denotes the gene pool of Definition 3.2.1. The first step is now to calculate the fitness values $f(c)$ for each chromosome in the population P. The chromosomes are then sorted according to their fitness value from high to low. In the next step, P is divided into an upper half U and a lower half L. Due to the sorting, U contains the chromosomes with lower fitness values and L the ones with higher fitness values. The first element from L, which is the one with the highest fitness value and therefore the weakest one, is removed. Afterwards three further, random chosen, chromosomes from L are deleted. To fill the four arisen vacancies two chromosomes $c_1 = (a_1, ..., a_n)$ and $c_2 = (b_1, ..., b_n)$ from U are *selected* randomly for the reproduction. The reproduction happens by *crossover*. The first half of c_1 and the second half of c_2 are concatenated and vice versa, to produce two offsprings $o_1 = (a_1, ..., a_{n/2}, b_{n/2+1}, ..., b_n)$ and $o_2 = (b_1, ..., b_{n/2}, a_{n/2+1}, ..., a_n)$. This process is repeated to receive four offsprings all in all. The next step is the *mutation*. Mutation can happen to each of the chromosomes except the new offsprings and the best chromosome which is the last chromosome from U. A mutation can occur at each *locus* of a chromosome with a probability of 5% if the chromosome is from U and with a probability of 10% if the chromosome is from L. The new population P includes the chromosomes from U, L, and the four new offsprings o_1, o_2, o_3, o_4. In the last step, duplicates are removed from the population and are replaced by new randomly generated chromosomes. This whole procedure from one population to a new population is

called a *generation*. The genetic algorithm performs several generations and in the end it provides the best chromosome of the last population as result. The genetic algorithm described above is listed in Algorithm 3 as pseudo-code.

Algorithm 3 Genetic Algorithm - Variant 1

```
1:  procedure GENETIC1(G, n, g)
2:      ▷ Gene pool G, chromosome size n, number of generations g
3:      ▷ Start with a population of 20 randomly chosen chromosomes of size n
4:      P ← []
5:      while size of P < 20 do
6:          c ← new chromosome
7:          c.genes ← random gene sequence (g_1, ..., g_n) with g_i ∈ G for 1 ⩽ i ⩽ n
8:          insert c into P
9:      end while
10:
11:     best ← null                                  ▷ Keep track of the best chromosome
12:     i ← 0
13:     while i < g do
14:         ▷ Calculate fitness of chromosomes and sort them according to fitness
15:         for all c ∈ P do
16:             c.fitness ← f(c)
17:         end for
18:
19:         sort P according to the chromosomes fitness from high to low
20:         L ← lower half of P
21:         U ← upper half of P
22:         remove first element from L              ▷ This is the weakest chromosome
23:         remove 3 more randomly chosen chromosomes from L
24:
25:         ▷ Perform Crossover
26:         c_1 ← random chromosome from U
27:         c_2 ← random chromosome from U
28:         o_1, o_2, o_3, o_4 ← new chromosomes
29:         o_1.genes ← (c_1.genes[1], ..., c_1.genes[n/2], c_2.genes[n/2 + 1], ..., c_2.genes[n])
30:         o_2.genes ← (c_2.genes[1], ..., c_2.genes[n/2], c_1.genes[n/2 + 1], ..., c_1.genes[n])
31:         o_3.genes ← (c_1.genes[1], ..., c_1.genes[n/2], c_2.genes[n/2 + 1], ..., c_2.genes[n])
32:         o_4.genes ← (c_2.genes[1], ..., c_2.genes[n/2], c_1.genes[n/2 + 1], ..., c_1.genes[n])
```

```
33:         ▷ Perform Mutation
34:         best ← select and remove best chromosome of U
35:         for all c ∈ L ∪ U do
36:             j ← 1
37:             while j ⩽ n do
38:                 if c ∈ L then
39:                     c.genes[j] ← random g ∈ G with probability 0.1
40:                 else
41:                     c.genes[j] ← random g ∈ G with probability 0.05
42:                 end if
43:                 j++
44:             end while
45:         end for
46:
47:         P ← L + U + [best, o1, o2, o3, o3]
48:         i++
49:         if i < g then
50:             remove duplicates from P
51:             fill arisen vacancies in P with randomly generated chromosomes
52:         end if
53:     end while
54:     return best                ▷ The best chromosome found in the specified generations
55: end procedure
```

Cooper et al. examined in their paper [8] different heuristic approaches to the phase-ordering problem and proposed some changes to their genetic algorithm presented in [9]. The changed genetic algorithm starts with a population P of 50 chromosomes of size n. The fitness value $f(c)$ is calculated for all chromosomes and the chromosomes are again sorted according to their fitness from high to low. The 10% of the chromosomes that have the lowest fitness values form the set B of the best chromosomes of this population. All chromosomes from $P \setminus B$ are deleted. The arisen vacancies are again filled via crossover. Two chromosomes $c_1 = (a_1, \ldots, a_n)$ and $c_2 = (b_1, \ldots, b_n)$ are randomly chosen from B for reproduction. Crossover is performed with c_1 and c_2 as described previously to create four new chromosomes. The last two steps, selection of two random chromosomes and crossover, are repeated until there are no more vacancies. All the new offsprings can be affected by mutation. Mutation proceeds like previously at every gene of a chromosome with a probability of 10%. If chromosomes occur, which already were part of a population at any time, then these chromosomes are mutated until this is no longer true. If all possible chromosomes occurred once in a population, the previous mutation loop is also canceled. The new population P contains the chromosomes from B and all new offsprings. In the last step, duplicates are removed from the population and are replaced by new randomly generated chromosomes. The genetic algorithm again performs several generations and provides the best chromosome of the last population as result. The latter algorithm is listed in Algorithm 4 as pseudo-code.

Algorithm 4 Genetic Algorithm - Variant 2

procedure GENETIC2(G, n, g)
  ▷ Gene pool G, chromosome size n, number of generations g
  ▷ Start with a population of 50 randomly chosen chromosomes of size n
  $P \leftarrow []$
  $best \leftarrow$ null ▷ Keep track of the best chromosome
  while size of $P < 50$ **do**
    $c \leftarrow$ new chromosome
    $c.genes \leftarrow$ random gene sequence $(g_1, ..., g_n)$ with $g_i \in G$ for $1 \leqslant i \leqslant n$
    insert c into P
  end while

  $i \leftarrow 0$
  while $i < g$ **do**
    ▷ Calculate fitness of chromosomes and sort them according to fitness
    for all $c \in P$ **do**
      $c.fitness \leftarrow f(c)$
    end for

    sort P according to the chromosomes fitness from high to low
    B ← select 10% of the best chromosomes from P
    $N \leftarrow []$

    ▷ Perform Crossover
    while size of N < 50− size of B **do**
      $c_1 \leftarrow$ random chromosome from B
      $c_2 \leftarrow$ random chromosome from B
      $o_1, o_2, o_3, o_4 \leftarrow$ new chromosomes
      $o_1.genes \leftarrow (c_1.genes[1], ..., c_1.genes[n/2], c_2.genes[n/2+1], ..., c_2.genes[n])$
      $o_2.genes \leftarrow (c_2.genes[1], ..., c_2.genes[n/2], c_1.genes[n/2+1], ..., c_1.genes[n])$
      $o_3.genes \leftarrow (c_1.genes[1], ..., c_1.genes[n/2], c_2.genes[n/2+1], ..., c_2.genes[n])$
      $o_4.genes \leftarrow (c_2.genes[1], ..., c_2.genes[n/2], c_1.genes[n/2+1], ..., c_1.genes[n])$
      insert o_1 into N
      if size of N < 50− size of B **then**
        insert o_2 into N
      end if
      if size of N < 50− size of B **then**
        insert o_3 into N
      end if
      if size of N < 50− size of B **then**
        insert o_4 into N
      end if
    end while

```
43:         ▷ Perform Mutation
44:         for all c ∈ N do
45:             j ← 1
46:             while j ⩽ n do
47:                 c.genes[j] ← random g ∈ G with probability 0.1
48:                 j++
49:             end while
50:
51:             ▷ Perform mutation again if c was already part of any population
52:             while c ∈ P at any generation ∧ not tried all s ∈ G^n do
53:                 j ← 1
54:                 while j ⩽ n do
55:                     c.genes[j] ← random g ∈ G with probability 0.1
56:                     j++
57:                 end while
58:             end while
59:         end for
60:
61:         P ← B ∪ N
62:         i++
63:         if i < g then
64:             remove duplicates from P
65:             fill arisen vacancies in P with randomly generated chromosomes
66:         end if
67:     end while
68:
69:     ▷ Return the best chromosome found in the specified generations
70:     return best
71: end procedure
```

4

EXPERIMENTS

This chapter describes experiments that are performed in order to evaluate the presented heuristic approaches and to find out how good is *-polly-canonicalize* in comparison to the sequences generated by heuristic algorithms. Furthermore, the optimization passes of *-O3* and *-polly-canonicalize* are investigated with regard to their occurrence in generated preoptimization sequences to make a point which optimization passes are useful for preoptimization and which are not. Based on the gathered information, a new fixed preoptimization sequence is created.

4.1 EXPERIMENTAL FRAMEWORK

The first part of this chapter describes the environment in which the experiments are performed. The research in this thesis uses the development version of LLVM (git version hash 20850bb), *clang* (git version hash e66c78e), and Polly (git version hash 07aed96) for the first three experiments. For the other experiments a newer version of LLVM (git version hash 1bb48fa), *clang* (git version hash a9f8b07), and Polly (git version hash ed8e11c) is required to get more detailed statistical output from the SCoP detection. All optimizations are performed on the sample programs with the LLVM tool opt, which is used with integrated Polly to enable SCoP detection.

The following command is used in the experiments to get information about the results of the SCoP detection if a sequence *x* is used for preoptimization:

```
$ opt -strip-debug -load=/POLLY_PATH/LLVMPolly.so x -polly-detect -stats LLVM-
    IR_FILE
```

The variable `POLLY_PATH` represents the path where Polly is located. The variable `LLVM-IR_FILE` represents the LLVM-IR file for which the SCoP detection should be performed. Beside the preoptimization sequence *x*, the passes *-polly-detect* and *-stats* are used for the SCoP detection and to get information about the results of the detection. These passes have already been discussed in Section 2.5. The pass *-strip-debug* just causes *opt* to strip debug information before applying other optimizations and has no effect on Polly's SCoP detection [1].

4.1.1 *Sample passes*

Until now, the fixed sequence *-polly-canonicalize*, that is described in Section 2.5, has been used to prepare LLVM-IR code for the SCoP detection. It is assumed that its passes

increase the amount of code that can be detected as SCoPs. In addition, the *-O3* flag represents a sequence of passes that should have a positive effect on the quality of the LLVM-IR code. Consequently, the passes of *-O3* and the ones of *-polly-canonicalize* are used in the experiments for the creation of the custom preoptimization sequences. The passes of *-polly-canonicalize* have already been discussed in Section 2.5. The passes of *-O3* and a short description of what they do are listed in Appendix A.

4.1.2 *Sample programs*

Different LLVM-IR sample programs are used for the studies of optimization sequences. On the one hand, the sample programs are generated from open-source, real-world C/C++ programs. These programs have sizes from 500 to 600.000 lines of code and different application domains like compilation, compression, and multimedia processing. An overview of these programs is given in Table 4. On the other hand, sample programs are generated from the benchmark codes included in PolyBench[1]. The benchmark codes of PolyBench stand out due to their high number of contained SCoPs.

Application domain	Programs
Compilation	python, ruby, spidermonkey, tcc
Compression	7za, bzip2, gzip, xz
Database	leveldb, postgres, sqlite3
Encryption	ccrypt, openssl
Multimedia	povray, x264
Scientific	lammps, linpack
Simulation	crafty, lulesh, lulesh-omp
Verification	crocopat, minisat

Table 4.: Real-world programs used for the studies of optimization sequences

The selection of the real-world programs is based on the fact that they have already been used in previous experiments with Polly [21]. The versions of the programs listed in Table 4 are still the same as in [21] and are obtained from the pprof-study[2] git repository (git version hash e37abc7). Another reason for the selection of these programs as sample programs is that they represent a good variety and due to their diversity a detailed assessment of optimization sequences is possible. In summary, 66 sample programs are used for the experiments, 36 from pprof-study and 30 from PolyBench.

1 http://web.cse.ohio-state.edu/~pouchet/software/polybench/
2 https://github.com/simbuerg/pprof-study

4.2 EXPERIMENTAL SETUP

This section describes the performed experiments and discusses their results. For better understanding, some results are displayed graphically. Comparisons of fitness values of different sequences are visualized mostly using bar charts. In the bar charts for the fitness comparison, the height of a bar represents the fitness value of the corresponding sequence. Due to the problem definition in Section 3.1, lower fitness values are better than higher ones and so lower bars are better than higher ones. A value in a bar represents the height of this bar. The fitness value of a sequence for a specific sample program is derived from the statistics of the SCoP detection. For the sake of simplicity, the following notation is used in the experiments:

- The set O denotes the set of the available optimization passes. For the first experiments O contains the optimization passes of *-polly-canonicalize* and *-O3*. The set O is redefined if other optimization passes are used in an experiment.
- Let A be a set. The cartesian power of A is defined as: $A^n = \underbrace{A \times \cdots \times A}_{\text{n times}} = \{(a_1, \ldots, a_n) : a_i \in A \forall 1 \leqslant i \leqslant n\}$.
- The set $S_n = O^n$ denotes the set of preoptimization sequences of length $n \in \mathbb{N}$.
- The set P_{poly} denotes the set of PolyBench sample programs.
- The set P_{pprof} denotes the set of the sample programs derived from pprof-study.
- Let $v : (P_{poly} \cup P_{pprof}) \times S_n \to \mathbb{N}$ be the evaluation function presented in Section 3.1. The fitness of a sequence $s \in S_n$ for a certain sample program p refers to the value $v(p, s)$.

4.2.1 *Experiment 1 - Fitness of fixed sequences*

The idea behind the first experiment is to get a first impression which fitness values can be reached with already available fixed optimization sequences. Hence, the fitness value of the empty sequence, the one of *-polly-canonicalize*, and the one of *-O3* and *-polly-canonicalize* together are calculated for each sample program. The appliance of the empty sequence means that no optimization pass is used for preoptimization. A part of the results is listed below in Figure 7. A complete overview of the results is presented in Appendix B. The x-axis of a bar chart shows the different sequences and the y-axis shows the corresponding fitness value. The graphic reveals that there is no fixed sequence which every time yields better results than other sequences. For example, *-O3* and *-polly-canonicalize* together have a better fitness value as *-polly-canonicalize* alone for the program *atax*, but with regard to the program *7za* it's the other way round. Furthermore, the sequence *-polly-canonicalize* has a worse fitness value than the empty sequence for the sample program *cholesky*. Consequently, the usage of fixed preoptimization sequences can be worse than applying no pass for preoptimization.

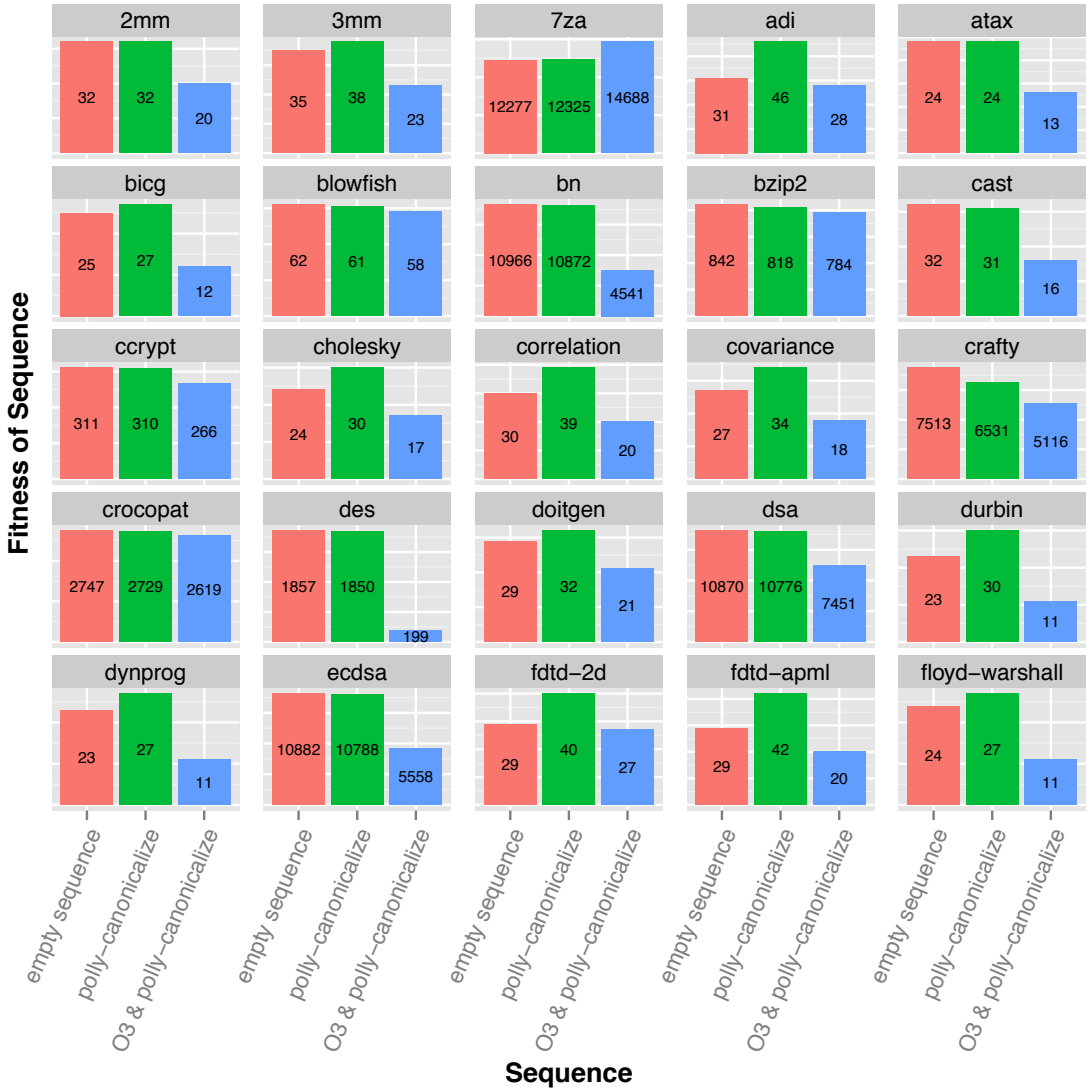

Figure 7.: Fitness comparison of fixed preoptimization sequences

4.2.2 *Experiment 2 - Fitness comparison of fixed and custom sequences*

In this experiment, the genetic algorithms described by Algorithm 3 and Algorithm 4 are used to generate custom preoptimization sequences for each sample program in P_{poly} and P_{pprof}. The genetic algorithm described by Algorithm 3 is referred to as "genetic 1" and the other genetic algorithm described by Algorithm 4 is referred to as "genetic 2" for the rest of this thesis. The following settings are used for this experiment:

- Genetic 1 uses a population size of 20 and calculates 200 generations.
- Genetic 2 uses a population size of 50 and calculates 50 generations.
- The gene pool for both genetic algorithms is the set O.

The first two settings are the settings recommended in [14]. Both heuristics search for sequences in S_{10} and S_{20}, so they run once with a chromosome length of 10 and once with a chromosome length of 20.

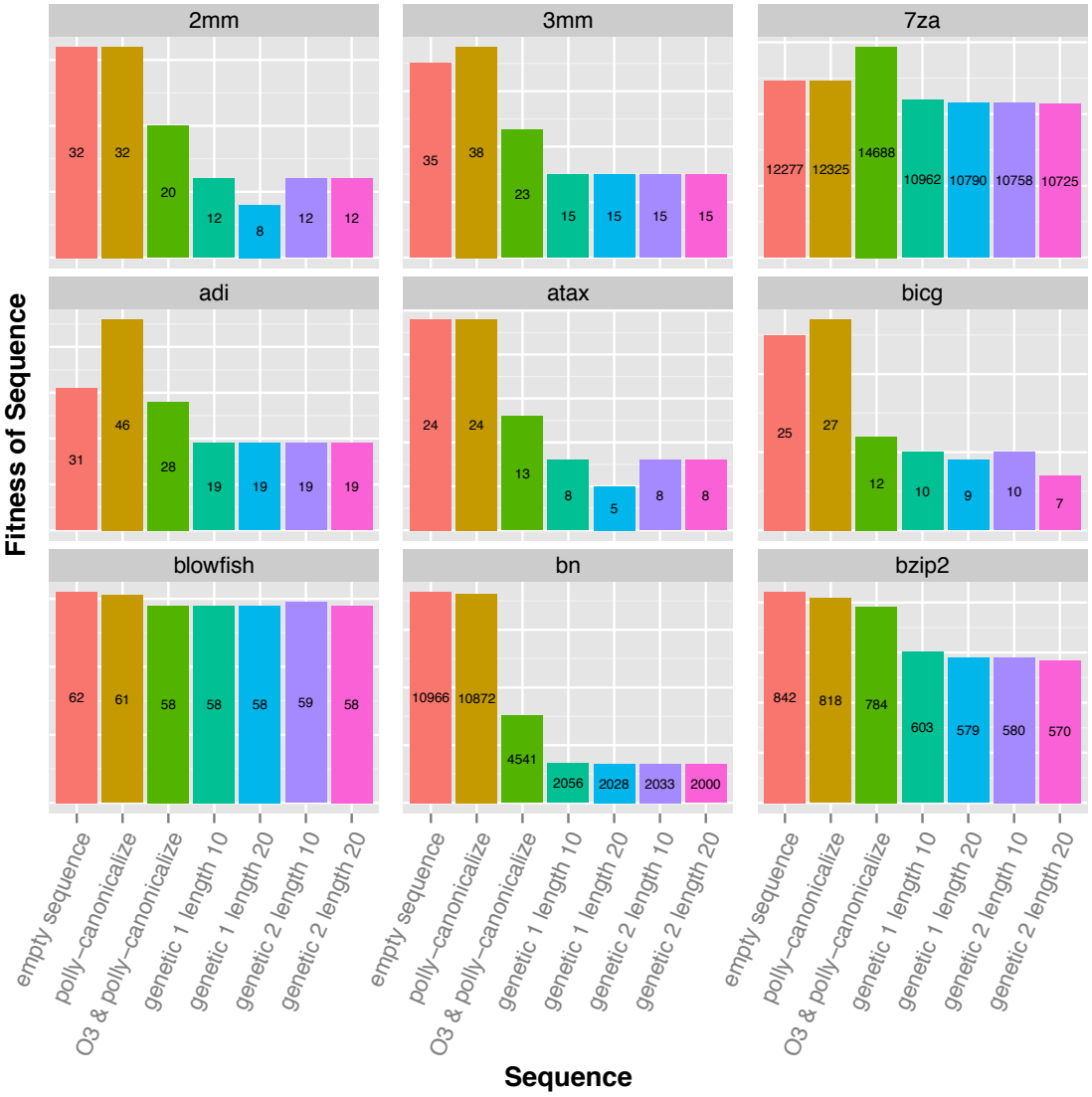

Figure 8.: Fitness comparison of fixed and custom preoptimization sequences

First, the structure of the generated preoptimization sequences is discussed. The generated sequences differ from program to program. The sequences of length 10 generated by genetic 1 are used as an example. The custom optimization sequence for the sample program *bzip2* of $\mathrm{P}_{\mathtt{pprof}}$ and the sample program *atax* of $\mathrm{P}_{\mathtt{poly}}$ are as follows:

bzip2 `-jump-threading -mem2reg -basicaa -gvn -prune-eh -sroa -simplifycfg -loops -polly-indvars -slp-vectorizer`

atax `-basicaa -no-aa -licm -barrier -globaldce -mem2reg -inline -sroa -globaldce -globaldce`

They partly contain different passes and have different orderings of the passes, but both sequences have also some common ground like they both contain the passes *-mem2reg* and *-basicaa*. So, one can hypothesize that the effect of a preoptimization sequence depends on the sample program that should be optimized. This is the reason why the sequences generated by the heuristic approaches deviate from each other. This observation sheds light why there is no fixed preoptimization sequence in Experiment 4.2.1 that outperforms the other sequences.

In the next step , the fitness values of the resulting custom preoptimization sequences are compared with the ones of the fixed preoptimization sequences. A part of the comparison's results is shown in Figure 8. The sequence denoted by "genetic 1 length 10" is the sequence of length 10 generated by genetic 1, "genetic 1 length 20" is the sequence of length 20 generated by genetic 1, "genetic 2 length 10" is the sequence of length 10 generated by genetic 2, and "genetic 2 length 20" is the sequence of length 20 generated by genetic 2. A complete overview of the results is presented in Appendix B.

The results are remarkable. The custom sequences have better fitness values for almost all sample programs. These results strengthen the findings of [8], [9], and [14] that heuristic approaches are able to generate custom sequences that improve the quality of the corresponding program better than a fixed sequence does if compiling for a certain metric. Furthermore, one can see that the length of the optimization sequence matters. As example, one can view the results for the sample program *bicg*. The longer custom sequences have better fitness values than the shorter ones.

4.2.3 *Experiment 3 - Optimization of generated custom sequences*

The quality of the results of genetic algorithms depends on the size of the search space because a genetic algorithm operates only on a subspace of the search space [8, p. 3]. So, the larger the search space the worse the results can be. Consequently, the quality of the results provided by genetic algorithms can be improved if the set of passes is reduced in a way that it only contains passes which help to increase the amount of detectable code. In the third experiment the best custom sequence, among the sequences generated by the genetic algorithms in the last experiment, is selected for each sample program and these best sequences are optimized. The optimization of a sequence is nothing more than an attempt to shorten the sequence in a way so that it only contains passes that contribute to the sequence's fitness value. The selection of the best custom sequences is done by the following approach:

1) For each program, select the custom sequences that have a lower fitness value than the best fixed sequence.

2) For each program, choose the shortest sequence from the selected custom sequences of step 1.

For example, Figure 8 shows that the best custom sequence for the sample program *bn* is the sequence of length 20 generated by genetic 2 because it is better than each fixed and than each other custom sequence.

The optimization or rather the shortening of the best custom sequences is done using the Algorithm 5 that is presented below.

Algorithm 5 Shortening

procedure SHORTEN(p, b)
  ▷ Sample program p
  ▷ Base sequence $b = (b_1, \ldots, b_{length(b)})$ that should be shortened
  $S \leftarrow \{b\}$ ▷ Keep track of current shorter sequences
  $R \leftarrow \{b\}$ ▷ Keep track of all shorter sequences
  while $S \neq \emptyset$ **do**
    ▷ Calculate all shorter sequences that have the same or a lower fitness value
    ▷ than b
    $S' \leftarrow \emptyset$
    for all $s \in S$ **do**
      $i \leftarrow 1$
      while $i \leqslant length(s)$ **do**
        $s' \leftarrow (s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_{length(s)})$
        if $\nu(p, s') \leqslant \nu(p, b)$ **then**
          $S' \leftarrow S' \cup \{s'\}$
        end if
        $i++$
      end while
    end for

    $S \leftarrow S'$
    $R \leftarrow R \cup S$
  end while

  ▷ Calculate the sequences with the lowest fitness value
  $L \leftarrow \{s \in R : \nu(p, s) \leqslant \nu(p, s') \forall s' \in R\}$
  ▷ Select the shortest sequences
  $L \leftarrow \{s \in L : length(s) \leqslant length(s') \forall s' \in L\}$
  ▷ If there are more shortest sequences, choose randomly one among them
  return random element of L
end procedure

The algorithm takes a sequence s as base sequence and generates all subsequences of s that contain one pass less than s. If one of these subsequences has the same or even a better fitness value than the base sequence, then the algorithm tries to shorten this sequence further. The algorithm remembers all generated subsequences with equal or better fitness value than the base sequence and if no subsequence can be shortened further, it selects the subsequences with the best fitness value from the set of remembered subsequences. In the next step the algorithm chooses the shortest sequence from the previously selected ones as result. If there is more than one shortest sequence, the algorithm chooses randomly one among them.

One observation of the sequence shortening is that there are generated sequences where a lot of passes could be removed. The sequence of length 20 generated by genetic 2 for the sample program *bzip2* is a good example of that. The passes highlighted with orange color are the passes which have been deleted.

original sequence	`-mem2reg -jump-threading -simplifycfg -basicaa -adce -loop-unswitch -ipsccp -instcombine -loop-idiom -gvn -jump-threading -ipsccp -ipsccp -sroa -simplifycfg -prune-eh -tailcallelim -jump-threading -polly-indvars -prune-eh`
shortened sequence	`-mem2reg -jump-threading -simplifycfg -basicaa -loop-unswitch -instcombine -gvn -ipsccp -sroa -simplifycfg -tailcallelim -jump-threading -polly-indvars -prune-eh`

It is evident that passes were omitted which occurred more than once like *ipsccp*, *-jump-threading*, or *-prune-eh*.

The next step is to examine which passes occur in the resulting shorter custom sequences. These are the passes that have a positive effect on the fitness value of a sequence. Figure 9 gives an overview of the relative frequency of the occurrence of a pass in a sequence.

You can derive from the graphic that the pass *-mem2reg* occurs in 32.2% of the best sequences. This is an astonishing result because it has been assumed that *-mem2reg* is necessary to detect any SCoP. Such a result was not anticipated at the beginning of the studies and, therefore, this result is further examined and discussed in the next experiment. Another interesting observation is that 11 of these 16 optimization passes are not part of *-polly-canonicalize*. This might be one reason why *-polly-canonicalize* is worse than the sequences generated by the genetic algorithms in Experiment 4.2.2.

The five passes that occur most frequently in the sequences are considered in detail now. The passes and interpretative attempts why they occur so many times in the best custom sequences are listed below. The passes are ordered from the most frequent one to the least frequent one. The description of the passes is derived from [1].

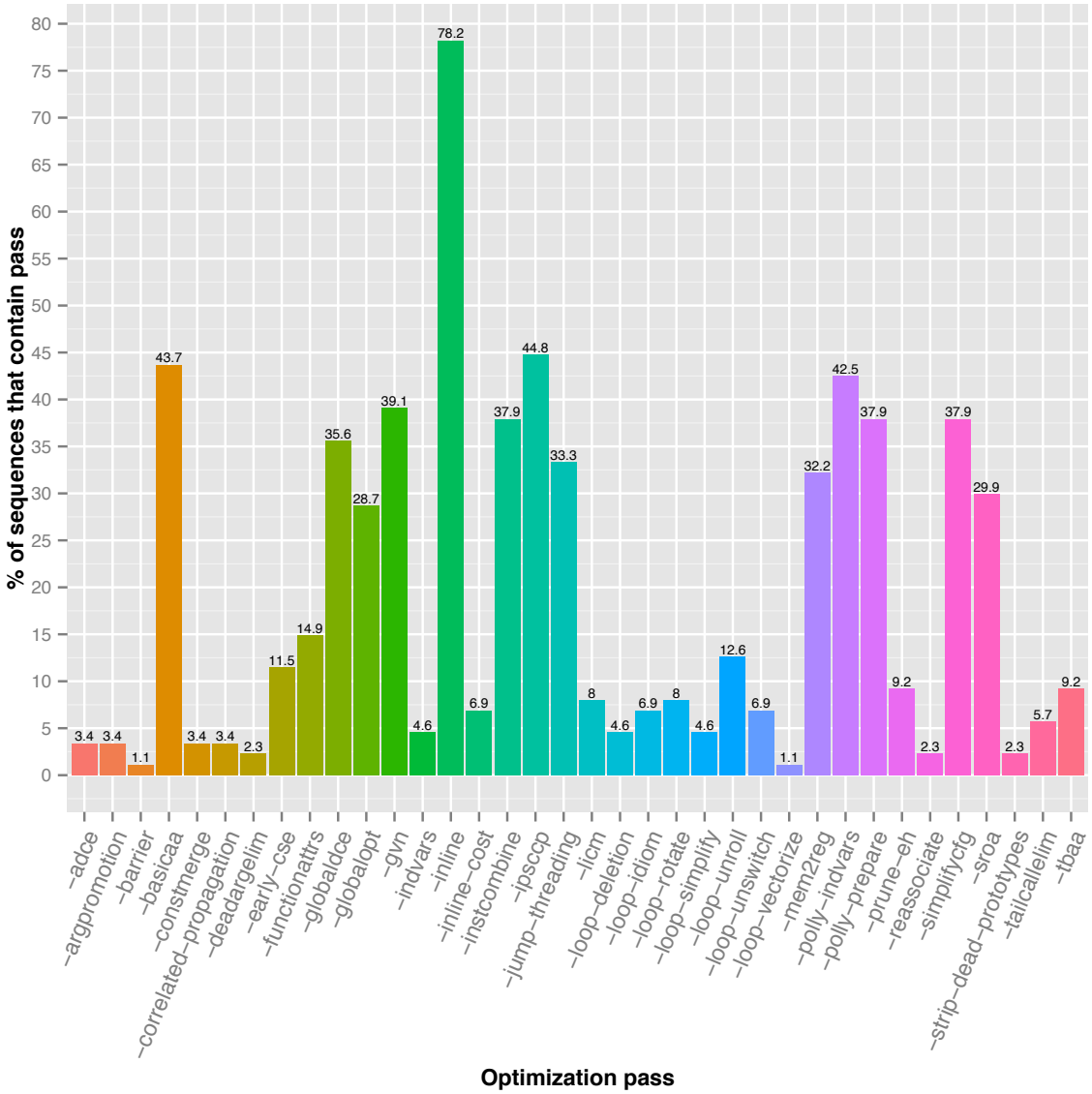

Figure 9.: Relative frequency of the pass occurrences

1) *-inline*: This pass inlines functions into callees. The course of action is bottom-up. Polly does not support function calls in SCoPs like discussed in Section 2.4. Consequently, the inlining of functions into its callees removes this constraint and a region that contained function calls can now be detected as SCoP.

2) *-ipsccp*: The interprocedural sparse conditional constant propagation performs dead code elimination and propagates constants over the program. In summary, the following actions are preformed. It assumes that values are constant unless it is proven otherwise and assumes that basic blocks of the CFG are dead unless it is proven otherwise. This pass proves values to be constant and then replaces them with constants. Finally, the pass proves conditional branches to be unconditional. The interprocedural substitution of values with constants and the dead code elimination are

the reasons why this pass occurs so many times in the custom sequences because it may transform non-affine expressions into affine ones and it may delete invalid basic blocks.

3) *-basicaa*: This pass performs a basic alias analysis. The analysis provides inter alia as result that there is no alias between two global constants. This pass has a positive effect on the SCoP detection because Polly requires the results of an alias analysis for the check if two base pointers refer to distinct memory spaces or are identical.

4) *-polly-indvars*: This pass is a variant of the pass *indvars* that is adjusted to the needs of Polly. It simplifies induction variables and transforms them into a canonical form. This pass has a positive effect on the SCoP detection because Polly detects only SCoPs where occurring induction variables match a canonical form.

5) *-gvn*: This pass assigns a value number to variables and expressions. It checks if variables and expressions are equivalent. Provably equivalent values and expressions get the same value number. Finally, this pass eliminates fully and partially redundant instructions using the assigned value numbers. Additionally, this pass eliminates redundant load instructions. Consequently, this pass occurs frequently because the elimination of redundant code reduces the number of regions that are no SCoPs.

4.2.4 *Experiment 4 - Effect of pass -mem2reg*

The pass *-mem2reg* has been assumed as a necessary preoptimization pass, but like observed previously it occurs only in 32.2% of the best custom sequences. In this experiment the effect of *-mem2reg* on the best custom sequences that do not contain it is examined. But first, *-mem2reg* is discussed in more detail. This pass promotes memory references to be register references. Hence, it looks at alloca instructions which only have loads and stores as uses. Consider the example C code shown in Listing 1. It presents a very simple program that declares two variables, calculates their sum, and it contains an if-statement.

```
int simpleSum() {
    int a = 8;
    int b = 2;
    int sum = a + b;

    if (sum > a) {
        b = 1;
    } else {
        b = 0;
    }

    return b;
}
```

Listing 1: Sample C code

```
define i32 @simpleSum() #0 {
entry:
  %a = alloca i32, align 4
  %b = alloca i32, align 4
  %sum = alloca i32, align 4
  store i32 8, i32* %a, align 4
  store i32 2, i32* %b, align 4
  %0 = load i32* %a, align 4
  %1 = load i32* %b, align 4
  %add = add nsw i32 %0, %1
  store i32 %add, i32* %sum, align 4
  %2 = load i32* %sum, align 4
  %3 = load i32* %a, align 4
  %cmp = icmp sgt i32 %2, %3
  br i1 %cmp, label %if.then, label %if.else

if.then:
  store i32 1, i32* %b, align 4
  br label %if.end

if.else:
  store i32 0, i32* %b, align 4
  br label %if.end

if.end:
  %4 = load i32* %b, align 4
  ret i32 %4
}
```

Listing 2: LLVM-IR version of code in Listing 1

Listing 2 shows the LLVM-IR version of the C code in Listing 1 and Listing 3 contains the LLVM-IR code after applying *-mem2reg*. After applying *-mem2reg* there are no more *alloca* instructions. The sum of a and b is calculated directly. Load and store instructions got replaced, for example by a PHI node in branch *if.end*.

```
define i32 @simpleSum() #0 {
entry:
  %add = add nsw i32 8, 2
  %cmp = icmp sgt i32 %add, 8
  br i1 %cmp, label %if.then, label %if.else

if.then:
  br label %if.end

if.else:
  br label %if.end

if.end:
  %b.0 = phi i32 [ 1, %if.then ], [ 0, %if.else ]
  ret i32 %b.0
}
```

Listing 3: LLVM-IR code of Listing 2 after applying *-mem2reg*

In the next step, the effect of *-mem2reg* on the best sequences that do not contain it is tested by once appending and once prepending *-mem2reg* to the sequence. Additionally, the effect of both, appending and prepending *-mem2reg*, is investigated. Then, the fitness values of the generated sequences are compared with the original sequence. A summary of the results is listed in Table 7:

# of the 66 best sequences that do not contain *-mem2reg*:	38
# of sequences to which the appending of *-mem2reg* has a positive effect:	0
# of sequences to which the appending of *-mem2reg* has a negative effect:	20
# of sequences to which the prepending of *-mem2reg* has a positive effect:	1
# of sequences to which the prepending of *-mem2reg* has a negative effect:	1
# of sequences to which both, appending and prepending of *-mem2reg*, has a positive effect:	0
# of sequences to which both, appending and prepending of *-mem2reg*, has a negative effect:	21

Table 7.: Effect of *-mem2reg*

The results show, first, that adding *-mem2reg* has a positive effect in only one of 38 cases and secondly, that there are some cases in which *-mem2reg* has a damaging effect on the fitness value. What is very clear here is that the appending of *-mem2reg* leads to a worsening of the fitness value in more than half of the cases. Appending and prepending *-mem2reg* to a sequence provides the worst results. The results for the custom sequences of the sample programs *7za*, *floyd-warshall*, and *lulesh* are considered in more detail to get a clearer picture of why *-mem2reg* has such effects. The column names of the Tables 8, 9, and 10 are explained in Appendix C. An overview of the whole results of this experiment is presented in Appendix D.

Tables 8, 9, and 10 show in the first row how the statistics of the SCoP detection changes after appending *-mem2reg* to the optimization sequence. A positive value indicates that the number has increased and a negative value that the number has decreased. The second row shows the results for the prepending of *-mem2reg* and the third row shows the results provided by appending and prepending *-mem2reg*.

Experiment	R	S	4	6	8	9	11	12	13	15	16	18	20	21	22	26
mem2reg appended	0	-90	-80	-5	-17	-1	-16	-41	-16	-25	-1023	-235	120	-355	1435	1435
mem2reg prepended	-3	0	4	-15	0	0	0	0	2	0	-6	0	0	0	0	0
mem2reg	-3	-90	-80	-5	-17	-1	-16	-41	-14	-25	-1025	-235	120	-355	1435	1435

Table 8.: Comparison of sequences with and without mem2reg for 7za

In Table 8 you can see that the prepending of *-mem2reg* has a positive effect on the fitness of the sequence because the number of regions (column R) decreased and the number of SCoPs (column S) stayed unrevised. On the other side, the appending of *-mem2reg* causes a worsening of the fitness value because the number of regions stayed unrevised and the number of SCoPs decreased. Additionally, one can observe that the number of PHI nodes in exit basic blocks (column 26) and the number of non canonical PHI nodes (column 20) has increased drastically. If *-mem2reg* is appended and prepended, the same trend can be observed. What a canonical PHI node looks like is explained by an example. A PHI node can be used for updating the induction variable of a loop like shown in Listing 4. The example in Listing 4 is derived from [1].

```
Loop:       ; Infinite loop that counts from 0 on up...
  %indvar = phi i32 [ 0, %LoopHeader ], [ %nextindvar, %Loop ]
  %nextindvar = add i32 %indvar, 1
  br label %Loop
```

Listing 4: Loop in LLVM-IR with PHI node

The PHI node in Listing 4 is canonical because it starts with value 0 and increases by one each step. Consequently, if a PHI node has not this form, it is not canonical.

Experiment	S	22	26
mem2reg appended	0	0	0
mem2reg prepended	-3	1	1
mem2reg	-3	1	1

Table 9.: Comparison of sequences with and without mem2reg for floyd-warshall

Table 9 shows the results for the sample program *floyd-warshall*. This is the only time that the prepending of *-mem2reg* has a negative effect on the fitness value. The number of SCoPs (column S) has decreased and for that the number of PHI nodes in exit basic blocks has increased. The appending of *-mem2reg* does not change the fitness of the sequence in this case.

Experiment	S	4	6	11	12	13	15	16	18	20	22	26
mem2reg appended	-8	-8	-3	-2	-3	-4	-1	-22	7	7	40	40
mem2reg prepended	0	0	-17	0	0	0	0	0	0	0	0	0
mem2reg	-8	-8	-3	-2	-3	-4	-1	-22	7	7	40	40

Table 10.: Comparison of sequences with and without mem2reg for lulesh

Table 10 shows the results for the sample program *lulesh*. Here one can observe the same trend as for the sample program *7za*. The appending of *-mem2reg* has a negative effect on the fitness value and the prepending has no effect.

In summary, the appending of *-mem2reg* to a sequence has a negative effect on its fitness in over half of the cases and the most common reasons for this are the increasing number of PHI nodes in exit basic blocks (column 26) and the increasing number of non canonical PHI nodes (column 22). On the other hand, the prepending of *-mem2reg* to an optimization sequence has no effect on its fitness value in the most cases. One might suspect that the prepending of *-mem2reg* provides not as bad results as the appending because the optimization passes that follow *-mem2reg* transform the inserted PHI nodes in a way they have no more negative effects on the SCoP detection. However, considering the results provided by appending and prepending *-mem2reg* to a sequence, one realizes that these results are as bad as the ones provided by just appending *-mem2reg*. More specifically, the results are almost identical. Consequently, it can be concluded that if *-mem2reg* is the last pass of the sequence, it inserts new PHI nodes regardless of whether *-mem2reg* has also been prepended or not. So, *-mem2reg* should always be one of the first optimization passes to run.

4.2.5 *Experiment 5 - Custom sequences with selected passes*

In this experiment it is investigated whether the genetic algorithms, genetic 1 and genetic 2, generate better sequences if the amount of used optimization passes is reduced or not. Like already noted in Experiment 4.2.3, the quality of the sequences generated by the genetic algorithms depends namely on the size of the search space. By reducing the amount of optimization passes, the search space is automatically reduced. Hence, only the most frequently occurring optimization passes, which have been presented in Experiment 4.2.3, are used in this experiment for the generation of the preoptimization sequences. Additionally, the hill climbing algorithm and the greedy algorithm are used to generate sequences. For this experiment only the sample programs in P_{poly} are used, to speed up the calculation. In this experiment the set O denotes the set of the frequent optimization passes presented in Experiment 4.2.3.

The following settings are used for this experiment:

- Genetic 1 uses a population size of 20 and calculates 200 generations.
- Genetic 2 uses a population size of 50 and calculates 50 generations.
- The gene pool for both genetic algorithms is the set O.
- The hill climbing algorithm and the greedy algorithm use the set O for the generation of sequences.
- All algorithms generate sequences of length 10.
- All algorithms generate sequences for the sample programs in P_{poly}.

A part of the experiment's results is shown in Figure 10. The x-axis shows again the different sequences. The first five sequences are the sequences already discussed in Experiment 4.2.2. The last four sequences are the sequences generated in this experiment. One can observe that the sequences generated in this experiment are at least as good as the previously generated optimization sequences of length 10. Furthermore, the custom preoptimization sequences from this experiment have in most cases a fitness value as good as the generated sequences of length 20 from Experiment 4.2.2. Consequently, the reduction of the used optimization passes does not really improve the genetic algorithms generation of preoptimization sequences, but the results of the genetic algorithms are as good as the previous ones. Another interesting result is that the sequences generated by the hill climbing algorithm are worse than the other custom preoptimization sequences. Kulkarni et al. discussed that their hill climbing algorithm finds solutions nearly as good as their genetic algorithm [14]. The solutions of the greedy algorithm are not as bad as the ones of the hill climbing algorithm, but they cannot keep up with the solutions of the genetic algorithms. A full list of the results is presented in Appendix E.

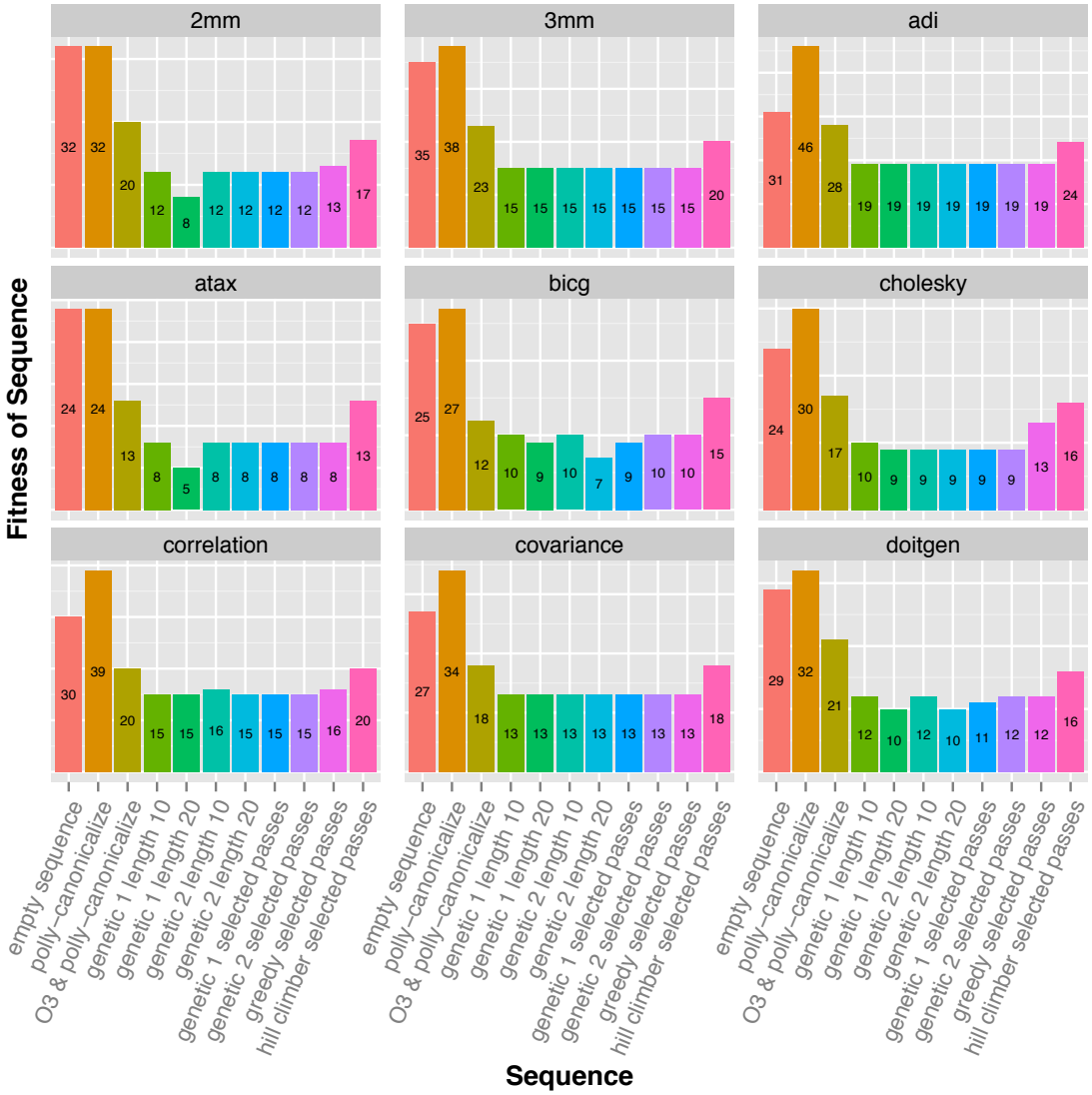

Figure 10.: Fitness comparison of custom sequences with selected optimization passes

4.2.6 *Experiment 6 - Construction of a new fixed preoptimization sequence*

This experiment is concerned with the construction of a new fixed preoptimization sequence that provides better results than the current fixed sequence represented by *-polly-canonicalize*. The 16 optimization passes presented in Experiment 4.2.3 are considered for this new preoptimization sequence. The new sequence should contain each of these optimization passes exactly once. Consequently, it will have a length of 16. For the sake of simplicity, the following notation is used in this experiment:

- The set O denotes the set of the 16 most frequent optimization passes that have been presented in Experiment 4.2.3.
- The set $S = \{(o_1, \ldots, o_{16}) \in O^{16} : i \neq j \implies o_i \neq o_j \forall 1 \leqslant i, j \leqslant 16\}$ denotes the set of candidate sequences.
- The set P_{small} denotes the set of the sample programs in P_{pprof} that have a file size less than 1 MB. Hence, P_{small} comprises 14 sample programs of P_{pprof}.

Creating a new fixed sequence still misses a partial ordering of the optimization passes. The ordering for the optimization passes can be obtained by examining pair-wise dependencies and/or restrictions for a single optimization pass. One restriction for the ordering can be derived from Experiment 4.2.4. The optimization pass *-mem2reg* should be applied first. The following restrictions can be derived from the documentation of the LLVM optimization passes [1]:

- The pass *-functionattrs* should run before the pass *-instcombine*. Whether the pass *-instcombine* can simplify library calls depends namely on the analysis results of the *-functionattrs* pass.
- The pass *-polly-indvars* should run before the pass *-loop-unroll* because the loop unrolling works best if the loops have been canonicalized.
- The pass *-globaldce* should run after the pass *-ipsccp* because it is possible that the pass *-ipsccp* makes definitions be dead.

Experiment 4.2.5 shows that the genetic algorithms are able to generate good custom sequences of length 10 with the set O of available optimization passes for the sample programs of P_{poly}. Hence, the genetic algorithms generate custom sequences with the same settings as in Experiment 4.2.5 now for all sample programs, so for the ones in P_{poly} and in P_{pprof}. The set of these custom sequences is denoted by G in this experiment and it holds that $G \subset O^{10}$. The custom sequences in G are used to obtain more ordering restrictions on the optimization passes. The next step examines, whether there are subsequences that occur more frequently in the custom sequences of G. A subsequence s_{sub} of a sequence $s = (o_1, \ldots, o_{10}) \in G$ is a sequence that contains just a part of the optimization passes of s and s_{sub} maintains the ordering of s. For example $s' = (o_1, o_3, o_9)$ is a subsequence of s, but $s'' = (o_2, o_5, o_3, o_7)$ is not a subsequence of s because o_3 must be before o_5. The most frequent subsequences of the sequences in

G indicate which optimization pass has to be applied before which other optimization pass. Table 11 shows the relative frequency of the different subsequences with a relative frequency of at least 20%. The occurring subsequences consist only of two optimization passes and can therefore be written as 2-tuples. Let $G_{sub} \subseteq O^2$ denote the set of the subsequences presented in Table 11.

Subsequence	Relative frequency of occurrence (in %)
(-inline, -inline)	39.4
(-inline, -basicaa)	36.4
(-basicaa, -polly-prepare)	34.8
(-polly-indvars, -polly-prepare)	31.8
(-inline, -globaldce)	28.8
(-basicaa, -inline)	28.8
(-inline, -instcombine)	28.8
(-inline, -gvn)	28.8
(-instcombine, -ipsccp)	28.0
(-gvn, -basicaa)	28.0
(-ipsccp, -basicaa)	28.0
(-instcombine, -jump-threading)	28.0
(-inline, -polly-prepare)	27.3
(-instcombine, -globaldce)	27.3
(-instcombine, -basicaa)	27.3
(-ipsccp, -jump-threading)	27.3
(-basicaa, -jump-threading)	27.3
(-gvn, -jump-threading)	27.3
(-instcombine, -polly-indvars)	26.5
(-inline, -polly-indvars)	26.5
(-gvn, -ipsccp)	25.8
(-instcombine, -inline)	25.8
(-basicaa, -polly-indvars)	25.8
(-globaldce, -ipsccp)	25.8
(-polly-indvars, -globaldce)	25.0
(-instcombine, -gvn)	24.2
(-inline, -globalopt)	24.2
(-instcombine, -polly-prepare)	24.2
(-instcombine, -simplifycfg)	24.2
(-basicaa, -ipsccp)	24.2
(-polly-indvars, -basicaa)	24.2
(-basicaa, -gvn)	23.5
(-inline, -jump-threading)	23.5
(-basicaa, -globaldce)	23.5
(-inline, -ipsccp)	23.5
(-gvn, -polly-indvars)	23.5
(-gvn, -polly-prepare)	22.7

(-inline, -functionattrs)	22.7
(-gvn, -simplifycfg)	22.7
(-globaldce, -inline)	22.7
(-simplifycfg, -polly-indvars)	22.0
(-gvn, -globaldce)	22.0
(-ipsccp, -polly-prepare)	22.0
(-inline, -simplifycfg)	22.0
(-inline, -sroa)	22.0
(-ipsccp, -polly-indvars)	22.0
(-ipsccp, -simplifycfg)	21.2
(-sroa, -inline)	21.2
(-simplifycfg, -polly-prepare)	21.2
(-jump-threading, -polly-indvars)	20.5
(-basicaa, -simplifycfg)	20.5

Table 11.: Subsequences that occur in the sequences of G with a relative frequency of at least 20%

The ordering of the optimization passes can now be derived from the frequency of the subsequences and the previously discussed restrictions. This makes it possible to define a strict partial order on the set of optimization passes O. The dependencies between optimization passes are determined as follows:

1. If a subsequence $g = (o, o) \in G_{sub}$ contains the same optimization pass twice, it is not considered because the new fixed sequence should contain each optimization pass exactly once and a strict partial order is irreflexive.

2. A dependency between two optimization passes is derived from each subsequence in G_{sub}, from the most frequent one to the least frequent one. Consequently, if a subsequence $g = (a, b) \in G_{sub}$ has already been taken into account, a later occurring subsequence $g' = (b, a) \in G_{sub}$ is not taken into consideration to maintain the asymmetry of the arising strict partial order.

Table 12 shows the predecessors for each optimization pass derived from the previously discussed restrictions and the occurring subsequences G_{sub}.

Optimization pass	Predecessors
-inline	*-mem2reg*
-sroa	*-mem2reg*, *-inline*
-basicaa	*-mem2reg*, *-inline*, *-ipsccp*, *-gvn*, *-instcombine*
-globaldce	*-mem2reg*, *-inline*, *-ipsccp*, *-gvn*, *-instcombine*, *-basicaa*, *-polly-indvars*
-polly-indvars	*-mem2reg*, *-inline*, *-ipsccp*, *-gvn*, *-instcombine*, *-basicaa*, *-simplifycfg*, *-jump-threading*

-instcombine	*-mem2reg*, *-inline*, *-functionattrs*
-mem2reg	
-loop-unroll	*-mem2reg*, *-polly-indvars*
-functionattrs	*-mem2reg*, *-inline*
-early-cse	*-mem2reg*
-polly-prepare	*-mem2reg*, *-inline*, *-ipsccp*, *-gvn*, *-instcombine*, *-basicaa*, *-simplifycfg*, *-polly-indvars*
-globalopt	*-mem2reg*, *-inline*
-simplifycfg	*-mem2reg*, *-inline*, *-ipsccp*, *-gvn*, *-instcombine*, *-basicaa*
-gvn	*-mem2reg*, *-inline*, *-instcombine*
-jump-threading	*-mem2reg*, *-inline -ipsccp*, *-gvn*, *-instcombine*, *-basicaa*
-ipsccp	*-mem2reg*, *-inline*, *-instcombine*, *-gvn*

Table 12.: Optimization passes and their predecessors

The knowledge about the ordering of optimization passes allows the definition of the following binary relation $R = O \times O$:

$$R = \{(\mathtt{mem2reg}, \mathtt{early-cse}), (\mathtt{mem2reg}, \mathtt{inline}), (\mathtt{inline}, \mathtt{sroa}), (\mathtt{inline}, \mathtt{globalopt}), (\mathtt{inline}, \mathtt{functionattrs}), (\mathtt{functionattrs}, \mathtt{instcombine}), (\mathtt{instcombine}, \mathtt{gvn}), (\mathtt{gvn}, \mathtt{ipsccp}), (\mathtt{ipsccp}, \mathtt{basicaa}), (\mathtt{basicaa}, \mathtt{simplifycfg}), (\mathtt{basicaa}, \mathtt{jump-threading}), (\mathtt{simplifycfg}, \mathtt{polly-indvars}), (\mathtt{jump-threading}, \mathtt{polly-indvars}), (\mathtt{polly-indvars}, \mathtt{polly-prepare}), (\mathtt{polly-indvars}, \mathtt{globaldce}), (\mathtt{polly-indvars}, \mathtt{loop-unroll})\}$$

By construction, the binary relation R is irreflexive and asymmetric. Let R^+ denote the transitive closure of R, then R^+ is irreflexive, asymmetric, and transitive. Consequently, R^+ defines a strict partial order on O and can be illustrated via a directed acyclic graph that is shown in Figure 11.

Each linear extension λ of the strict partial order R^+ represents a strict total order of the optimization passes in O. Hence, each linear extension λ of R^+ is a new preoptimization sequence because it specifies a 16-tuple of optimization passes. Therefore, the set $\Lambda(R^+)$ of all linear extensions of R^+ is considered now as the set of candidate sequences for the new fixed preoptimization sequence. The problem is now the calculation of all possible linear extensions of R^+. The problem of finding a single linear extension λ of R^+ is equivalent to the problem of finding a topological sorting of the directed acyclic graph shown in Figure 11. Varol and Rotem have presented an algorithm to generate all topological sorting arrangements of a directed acyclic graph [22]. Fortunately, there already exists a toolset for working with directed acyclic graphs in Python[3] and this toolset contains an implementation of Varol's and Rotem's algorithm. This implementation is used to generate all topological sorting arrangements of the graph shown in

3 `https://pypi.python.org/pypi/digraphtools/0.2.1`

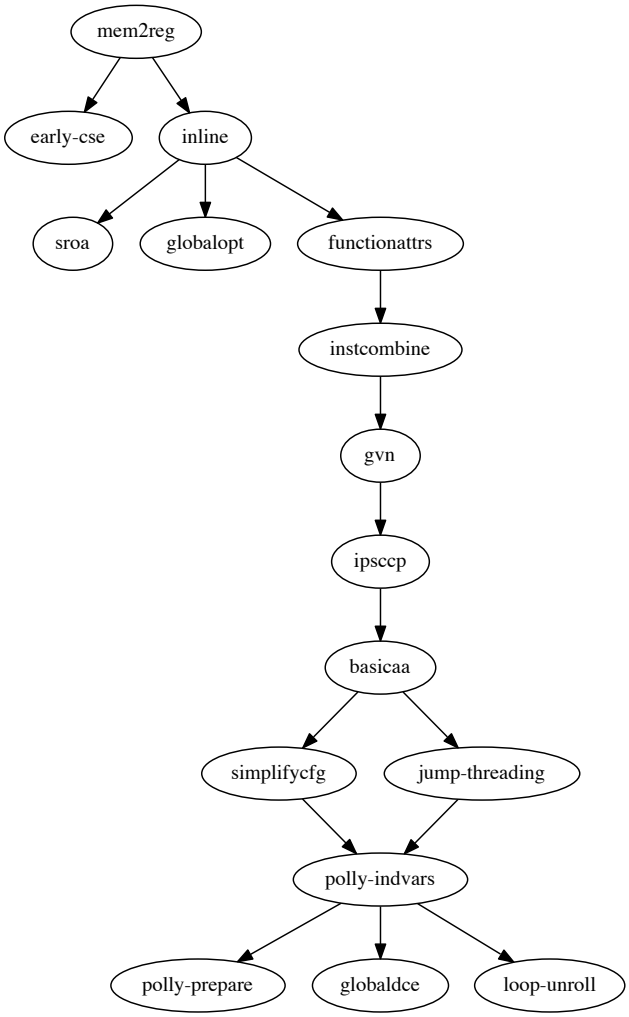

Figure 11.: Dependency graph of the selected optimization passes

Figure 11. The algorithm calculates 28079 preoptimization sequences that are valid topological sorting arrangements and so there are 28079 candidate sequences. The set of candidate sequences is denoted by C from now on.

Due to the high number of valid sequences, the following approach is used to shrink the number of possible new fixed sequences:

1. Let $P = P_{poly}$.
2. For each sample program $p \in P$ calculate the set $B_p = \{c \in C : \nu(p,c) \leqslant \nu(p,c')\forall c' \in C\} \subseteq C$. The result is a set B that contains for all $p \in P$ the set B_p. In other words, the set B contains for each program the set of candidate sequences that are best for this program.

3. For each candidate sequence $c \in C$ calculate the number n_c of elements of B that contain c, so $n_c = |\{B_p \in B : c \in B_p\}|$. Then calculate the set $C' = \{c \in C : n_c \geqslant n_{c'} \forall c' \in C\} \subseteq C$. The set C' contains now the candidate sequences of C that are best for most of the sample programs of P.

   The set C' contains now 21188 of the candidate sequences of C and each of these 21188 candidate sequences appears in all sets that are element of B. In other words, these candidate sequences are best for all of the 30 sample programs of P_{poly}.

4. Next, repeat step 2 and 3 with $P = P_{poly} \cup P_{small}$ and $C = C'$. The resulting set C' contains now only 72 candidate sequences and theses candidate sequences are among the best sequences for 40 of the 44 sample programs of P_{small} and P_{poly}.

5. Again, repeat step 2 and 3 with $P = P_{poly} \cup P_{pprof}$ and $C = C'$. As result, the set C' contains now only 2 candidate sequences and theses candidate sequences are among the best sequences for 55 of the 66 sample programs of P_{pprof} and P_{poly}.

6. In the last step, repeat step 2 and 3 with again $P = P_{poly} \cup P_{pprof}$ and $C = C'$. Consequently, C contains now only the 2 candidate sequences that remained after step 5. As final result, the set C' contains only 1 candidate sequence that is best for all of the 66 sample programs of P_{poly} and P_{pprof}.

The above steps generate one single candidate sequence. Consequently, this remaining candidate sequence is the wanted new fixed preoptimization sequence and is called *-polly-preopt* from now on. The sequence *-polly-preopt* looks like this:

***-polly-preopt* = (-mem2reg, -early-cse, -inline, -functionattrs, -instcombine, -globalopt, -sroa, -gvn, -ipsccp, -basicaa, -simplifycfg, -jump-threading, -polly-indvars, -loop-unroll, -globaldce, -polly-prepare)**

Figure 12 shows a part of the fitness comparison of the old fixed sequence *-polly-canonicalize* and the new sequence *-polly-preopt*. The x-axis shows the different sequences again. The y-axis shows the fitness value of the sequences. A complete overview of the results is presented in Appendix F.

Figure 12 shows that *-polly-preopt* is better than *-polly-canonicalize* for almost all sample programs. There are 10 sample programs, *7za*, *js*, *lammps*, *openssl*, *postgres*, *povray*, *python*, *ruby*, *x264*, and *xz*, for which the new sequence is not better than *-polly-canonicalize*. These 10 sample programs are of $P_{pprof} \setminus P_{small}$ and so, only the 72 candidate sequences that remained after step 4 have been evaluated because the size of these programs is too large to evaluate all 28079 sequences in adequate time. It might therefore be quite possible that there is a better sequence for these sample programs among the 28079 candidate sequences. If one takes a closer look at the results for the sample program *7za*, one can observe that the main reason why the new sequence is worse than *-polly-canonicalize* is the higher number of regions in the sample program. This is also the main reason why the new sequence is worse than *-polly-canonicalize* for the other 9 sample programs. The interesting thing about this is that the custom sequence generated by the algorithm genetic 2 for the sample program *7za*, called *genetic-7za*, contains just 10 optimization

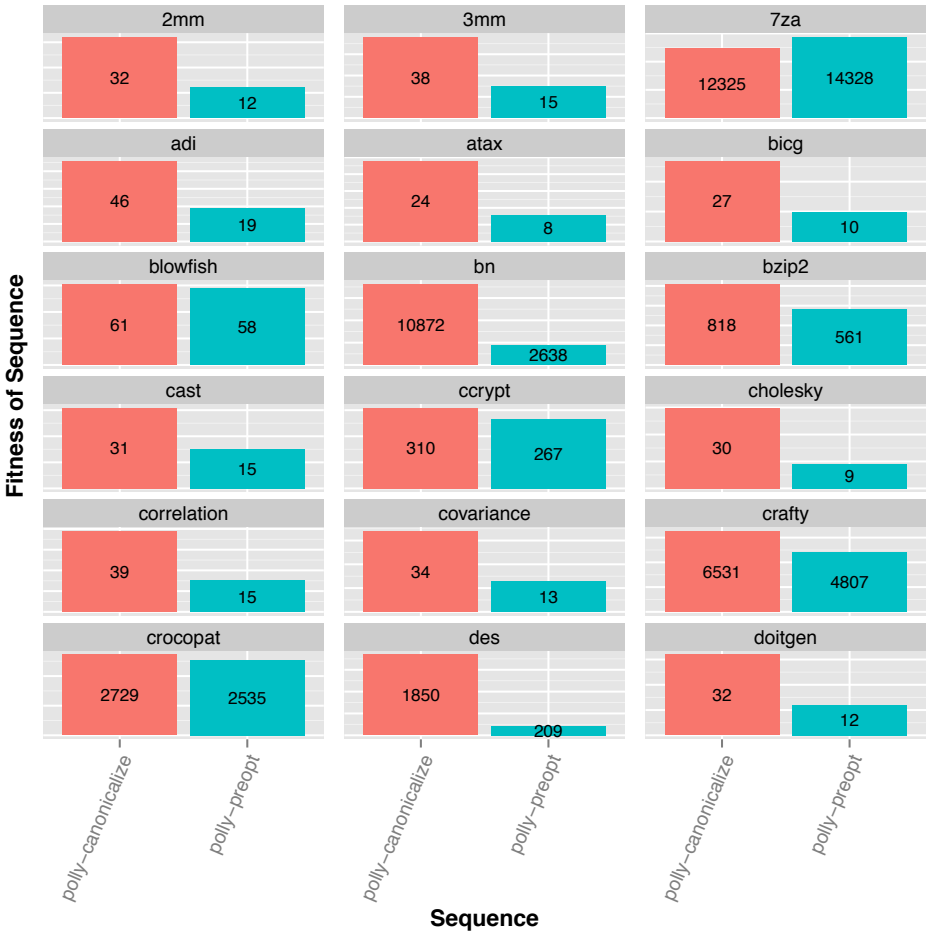

Figure 12.: Fitness comparison of *-polly-canonicalize* and *-polly-preopt*

passes that are all from the set O and *genetic-7za* is better than *-polly-canonicalize*. This custom sequence for *7za* looks like this:

genetic-7za = (-ipsccp, -instcombine, -polly-indvars, -loop-unroll, -globalopt, -globaldce, -gvn, -polly-prepare, -basicaa, -ipsccp)

One can observe immediately that the custom sequence does not contain the passes *-mem2reg*, *-early-cse*, *-inline*, *-functionattrs*, *-sroa*, *-simplifycfg*, and *-jump-threading*, that are present in *-polly-preopt*. Hence, it is checked how the results of Polly's SCoP detection for *-polly-preopt* change if one omits these optimization passes. Table 13 shows the test results. The column names of Table 13 are abbreviations that are explained in Appendix C. The results show that the fitness value (difference of entry in column R and entry in column S) of the new optimization sequence for the sample program *7za* is better if the optimization pass *-inline* is omitted. This pass performs inlining of functions into callees [1], but apparently this leads just to an increase of the number of regions what makes the fitness value worse.

Omitted pass	R	S	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
all	11071	168	1	1	0	269	0	16	0	123	3	0	36	91	114	240	64	2398	0	766	0	0	766	23	0	2	0	21	0
-early-cse	14511	185	1	1	0	296	0	16	0	158	3	0	25	94	110	309	75	2839	0	769	0	0	769	26	0	1	0	25	0
-functionattrs	14623	186	1	1	0	295	0	16	0	158	3	0	25	93	110	309	72	2837	0	769	0	0	769	26	0	1	0	25	0
-inline	11115	171	1	1	0	272	0	19	0	124	3	0	36	90	112	252	96	2439	0	768	0	0	768	21	0	1	0	20	0
-jump-threading	14835	186	1	1	0	294	0	13	0	160	3	0	25	93	112	325	75	2989	0	808	0	0	808	26	0	1	0	25	0
-mem2reg	14508	186	1	1	0	295	0	16	0	158	3	0	25	93	111	309	69	2839	0	770	0	0	770	26	0	1	0	25	0
-simplifycfg	14567	186	1	1	0	295	0	16	0	158	3	0	25	93	110	309	75	2841	0	769	0	0	769	26	0	1	0	25	0
-sroa	14636	186	1	1	0	295	0	16	0	158	3	0	25	93	111	312	75	2849	0	778	0	0	778	26	0	1	0	25	0

Table 13.: Changes due to omitted optimization passes

In the next step *-polly-preopt* without *-inline* is compared with *-polly-preopt*, and *-polly-canonicalize*, and with the sequences generated by the genetic algorithms at the beginning of this experiment. A part of the results is depicted in Figure 13. The x-axis shows again the different sequences. The y-axis shows the fitness value of the sequences. A complete overview of the results is presented in Appendix F.

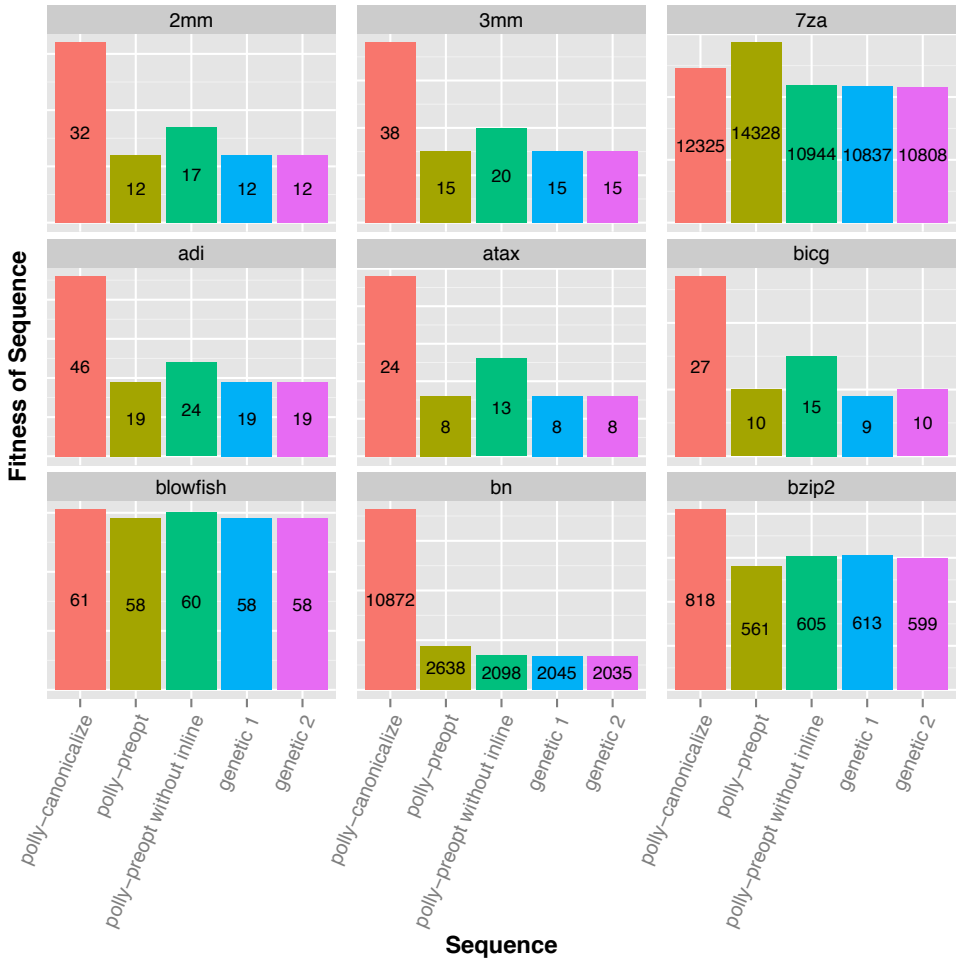

Figure 13.: Fitness comparison of *-polly-canonicalize* and the two new sequences

Figure 13 shows that *-polly-preopt* without *-inline* is worse than *-polly-preopt* in some cases, for example for *atax*, but there are also sample programs for which the new sequence without *-inline* is better, for example for *bn* and *7za*, because the omission of *-inline* decreases the number of regions in these sample programs. The reason why *-polly-preopt* without *-inline* is worse than *-polly-preopt* with *-inline* for some sample programs, is that the omission of *-inline* increases the number of regions in these sample programs. Consequently, one can conclude that *-inline* can have a positive effect on some sample programs, but can also have a negative effect. But the most astonishing result is that *-polly-preopt* without *-inline* is now better than *-polly-canonicalize* for all of the 66 sample programs. Furthermore, the fitness values of *-polly-preopt* without *-inline* are closer to the fitness values of the sequences generated by the genetic algorithms than the fitness values of *-polly-canonicalize* are.

Figure 14 shows the fitness of the new sequences *-polly-preopt* and *-polly-preopt* without *-inline* in comparison to the fitness of *-polly-canonicalize*. The x-Axis shows the different sample programs. The y-Axis shows the percentage by which the fitness of *-polly-preopt* and *-polly-preopt* without *-inline* is better or worse than the fitness of *-polly-canonicalize*. Consequently, a positive value on the y-axis indicates that the new sequence is better for this sample program than *-polly-canonicalize* and a negative value indicates that the new sequence is worse than *-polly-canonicalize*.

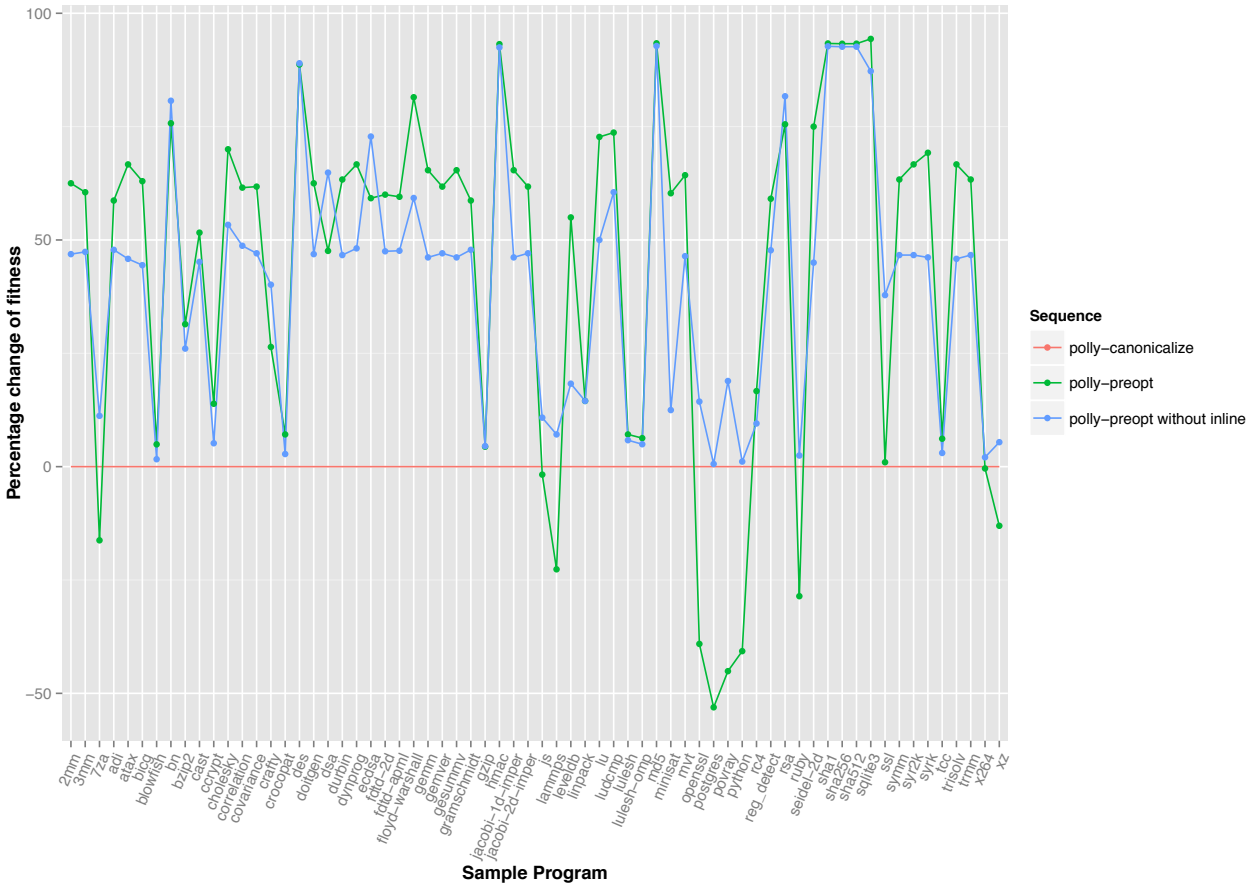

Figure 14.: Fitness change of new sequences compared to *-polly-canonicalize*

The test results show that it is possible to create a fixed sequence that provides results that are quite good in comparison with the sequences generated by heuristic approaches. Furthermore, the created sequence *-polly-preopt* without *-inline* provides better results than *-polly-canonicalize* for all sample programs and, hence, it is recommended as new preoptimization sequence for Polly.

***-polly-preopt* without *-inline* = (-mem2reg, -early-cse, -functionattrs, -instcombine, -globalopt, -sroa, -gvn, -ipsccp, -basicaa, -simplifycfg, -jump-threading, -polly-indvars, -loop-unroll, -globaldce, -polly-prepare)**

5

CONCLUSION

Polly is a plugin for the LLVM compiler framework that enables polyhedral optimization on LLVM-IR. Polly can only optimize parts of the LLVM-IR code that are detected as Static Control Parts (SCoPs). Hence, LLVM-IR code is prepared for Polly's SCoP detection with a preoptimization sequence that aims to increase the amount of code that can be detected as SCoPs.

This thesis examined the quality of the currently used fixed preoptimization sequence and investigated whether preoptimization sequences generated by heuristic approaches produce better results than current fixed sequences or not. The performed experiments showed that the discussed heuristics provide preoptimization sequences that outperform fixed sequences such as *-polly-canonicalize*. Furthermore, one was able to observe that custom preoptimization sequences depend strongly on the program or rather the LLVM-IR code that should be optimized and that there is no single sequence length which is best for all sample programs. The position of some optimization passes, like *-mem2reg*, in a sequence influences the fitness of this sequence. In summary, the fitness of a preoptimization sequence depends on the LLVM-IR code that should be optimized, on the optimization passes that are used, on the length of the sequence, and on the ordering of the optimization passes.

The last experiment used sequences found by heuristic approaches to extract a partial order on optimization passes. The possible topological sorting arrangements were deduced from this partial order and were evaluated to find a new preoptimization sequence for Polly. The obtained sequence was adjusted to the needs of Polly and a new fixed preoptimization sequence was formed that provides better results than *-polly-canonicalize* for all tested sample programs.

In conclusion, the thesis points out that heuristic approaches provide good preoptimization sequences for Polly. Furthermore, the thesis shows that one can use heuristic approaches to gather information about which optimization passes are useful when compiling for a certain metric and that this knowledge can be used to build better fixed preoptimization sequences.

OUTLINE The experiments just examined the fitness of sequences provided by heuristic approaches that work with a fixed predefined sequence length, but as pointed out previously the length of a sequence has an influence on its fitness. Consequently, more complex heuristic approaches that work with a variable sequence length can be examined. A remaining task is also to study the interaction between two optimization passes

in more detail. Currently, one can only say that a preoptimization sequence a has a better fitness than a preoptimization sequence b, but it is not possible to determine the contribution of a single optimization pass to the fitness of a preoptimization sequence. Hence, a metric is wanted that enables the assessment of single optimization passes. Finally, in the last experiment the set of the 28079 candidate sequences was shrunk by studying just the sequences that are best for the sample programs derived from PolyBench. To really make sure that the new fixed preoptimization sequence is the best among these 28079 sequences, one would have to evaluate each of these sequences for each of the sample programs.

Part III

APPENDIX

A

SELECTED PASSES OF THE -*O3* OPTIMIZATION SEQUENCE

This chapter presents the passes of the -*O3* optimization sequence that are used in the experiments discussed in Section 4.2. The passes and a short description of what they do are listed in Table 14. The description of the passes is derived from [1].

Passes of -*O3*

Pass name	Pass description
-adce	Performs an aggressive dead code elimination. This pass assumes that values are dead until proven otherwise.
-argpromotion	Promotes pointer arguments to scalar value arguments. Checks for each internal function with pointer arguments wether the pointer arguments are only loaded and if so, they will be replaced by the value. The pass uses the results of an alias analysis for this check.
-basicaa	Performs a basic alias analysis.
-constmerge	Merges duplicate global constants together into a single global constant.
-correlated-propagation	Performs value propagation.
-deadargelim	Removes dead arguments and dead return values from internal functions.
-dse	Eliminates dead stores that are basic-block local.
-early-cse	Performs a simple dominator tree walk that eliminates trivially redundant instructions.
-indvars	Transforms induction variables into a simpler canonical form.
-inline	Performs bottom-up inlining of functions into callees.
-ipsccp	Performs interprocedural sparse conditional constant propagation.
-licm	Tries to remove as much code of a loop body as possible.
-loop-deletion	Deletes loops with finite computable trip counts that have no side effects and do not contribute to the computation of the function's return value.

-tbaa	Performs a type based alias analysis. For this purpose, metadata is added to the LLVM-IR to describe a type system of a higher level language.
-barrier	A no-op barrier pass to allow manipulation of the implicitly nesting pass manager.
-basiccg	Constructs a call graph.
-block-freq	Performs a block frequency analysis.
-branch-prob	Performs a branch probability analysis.
-functionattrs	Interprocedural pass that walks the call graph looking for functions which do not access or only read non local memory and marking them readnon/readonly. In addition, it marks function arguments of type pointer nocapture if the pointer is only dereferenced and not returned from the function or stored in a global.
-globaldce	Eliminates unreachable internal globals from the program.
-globalopt	Transforms global variables that never have their address taken into global constants and deletes variables only stored to and more.
-gvn	Performs global value numbering to eliminate fully and partially redundant instructions.
-inline-cost	Performs an inline cost analysis.
-jump-threading	Looks at blocks that have multiple predecessors and multiple successors. If one or more of the predecessors of the block can be proven to always cause a jump to one of the successors, the edge from the predecessor is forwarded to the successor.
-lazy-value-info	Performs a lazy value information analysis.
-loop-unswitch	Transforms loops that contain branches on loop-invariant conditions to have multiple loops.
-loop-idiom	Transforms simple loops into a non-loop form.
-loop-unroll	Performs loop unrolling.
-loop-vectorize	Searches for loops that can be vectorized, transforms these loops, and generates vector codes.
-memcpyopt	Performs transformations related to eliminating memcpy calls or transforming sets of stores into memsets.
-memdep	Performs an analysis that determines for a given memory operation what preceding memory operation it depends on.
-no-aa	Performs an alias analysis that always returns may alias.
-prune-eh	Interprocedural pass that walks the call graph and turns LLVM-IR invoke instructions into call instructions if and only if the callee cannot throw an exception.

-sccp	Performs sparse conditional constant propagation and merging. Assumes values are constants unless proven otherwise and basic blocks are dead unless proven otherwise. Proves values to be constant and replaces them with constants. Proves conditional branches to be unconditional.
-slp-vectorizer	Detects consecutive stores that can be put together into vector-stores and attempts to construct a vectorizable tree using use-def chains. If a profitable tree was found, vectorization is performed on this tree.
-sroa	Scalar replacement of aggregates transformation. It tries to identify promotable elements of an aggregate alloca, and promotes them to registers. It will also try to convert uses of an element (or set of elements) of an alloca into a vector or bitfield-style integer scalar if appropriate.
-strip-dead-prototypes	Loops over all of the functions in the input module, looking for dead declarations and removes them. Dead declarations are declarations of functions for which no implementation is available.

Table 14.: Set of -*O3* passes used for the research

B

FITNESS COMPARISON OF OPTIMIZATION SEQUENCES

This chapter presents the results of the fitness comparison of preoptimization sequences discussed in experiments 1 and 2 in Section 4.2. Table 16 lists for every examined sample program the fitness values of the different sequences. The first column shows the sample program and the other columns show the fitness values of the different preoptimization sequences. The column names of Table 16 are explained in Table 15.

Column name	Preoptimization sequence
1	empty sequence (no optimization pass applied)
2	*-polly-canonicalize*
3	*-O3 -polly-canonicalize*
4	sequence of length 10 found by genetic algorithm 1
5	sequence of length 20 found by genetic algorithm 1
6	sequence of length 10 found by genetic algorithm 2
7	sequence of length 20 found by genetic algorithm 2

Table 15.: Column names and referred preoptimization sequences

Sample program	1	2	3	4	5	6	7
2mm	32	32	20	12	8	12	12
3mm	35	38	23	15	15	15	15
7z	12277	12325	14688	10962	10790	10758	10725
adi	31	46	28	19	19	19	19
atax	24	24	13	8	5	8	8
bicg	25	27	12	10	9	10	7
blowfish	62	61	58	58	58	59	58
bn	10966	10872	4541	2056	2028	2033	2000
bzip2	842	818	784	603	579	580	570
cast	32	31	16	15	14	14	14
ccrypt	311	310	266	269	262	267	261
cholesky	24	30	17	10	9	9	9

correlation	30	39	20	15	15	16	15
covariance	27	34	18	13	13	13	13
crafty	7513	6531	5116	4016	3893	3973	3860
crocopat	2747	2729	2619	2557	2525	2539	2554
des	1857	1850	199	201	198	198	198
doitgen	29	32	21	12	10	12	10
dsa	10870	10776	7451	3766	3731	3758	3698
durbin	23	30	11	11	11	11	11
dynprog	23	27	11	9	9	9	9
ecdsa	10882	10788	5558	2921	2889	2910	2884
fdtd-2d	29	40	27	16	16	16	16
fdtd-apml	29	42	20	17	17	17	17
floyd-warshall	24	27	11	5	5	5	5
gemm	27	26	15	9	9	9	9
gemver	26	34	20	13	12	13	13
gesummv	21	26	9	9	9	9	9
gramschmidt	34	46	24	19	19	20	19
gzip	722	683	464	602	600	600	599
hmac	10795	10701	1194	715	664	665	640
jacobi-1d-imper	21	26	13	9	9	9	9
jacobi-2d-imper	25	34	17	13	13	13	13
js	36155	36354	41819	32673	32416	32447	32306
lammps	32323	28766	36450	26773	26675	26668	26409
leveldb	2668	2694	1348	1364	1141	1206	1077
linpack	70	69	81	59	59	65	59
lu	24	22	13	5	5	5	5
ludcmp	28	38	18	10	10	10	15
lulesh	413	394	414	367	366	366	366
lulesh-omp	468	445	466	416	416	417	416
md5	10795	10701	1132	684	635	723	623
minisat	630	665	338	246	247	244	240
mvt	23	28	11	10	9	10	10
openssl	20279	20021	33279	17120	16936	17010	16900
postgres	50659	50697	84562	50317	50259	50261	50235
povray	12347	12277	17570	10144	9855	9963	9883
python	22922	22149	31344	22042	21924	21922	21867
rc4	43	42	35	35	35	35	35
reg_detect	30	44	23	18	18	18	18
rsa	10807	10713	3337	1932	1909	1935	1898
ruby	22336	22127	35373	21639	21559	21569	21548
seidel-2d	23	20	11	5	5	5	5

sha1	10799	10705	1135	689	635	718	620
sha256	10800	10706	1141	729	682	750	623
sha512	10800	10706	1140	732	662	676	631
sqlite3	2623	2654	138	193	133	208	124
ssl	14403	14238	17355	8866	8795	8805	8747
symm	25	30	14	11	11	11	11
syr2k	27	30	15	10	10	10	10
syrk	27	26	15	8	8	8	8
tcc	3833	3278	3785	2944	2908	3013	2883
trisolv	21	24	10	8	8	8	8
trmm	23	30	11	11	11	11	11
x264	7771	7564	7763	7406	7372	7386	7364
xz	1532	1464	1645	1393	1378	1397	1377

Table 16.: Fitness comparison of sequences of experiment 1 and 2 in section 4.2

C

ABBREVIATIONS FOR THE DETECTION STATISTICS

Statistics of a SCoP detection can be obtained if the pass *-stats* is used. Table 17 lists abbreviations that are used in the thesis for these statistics.

Abbreviation	Statistic
R	Number of regions
S	Weighted number of regions that are a valid SCoP
1	Number of bad regions for SCoP: CFG too complex
2	Number of bad regions for SCoP: Non branch instruction terminates basic block
3	Number of bad regions for SCoP: Not well structured condition in basic block
4	Number of bad regions for SCoP: Expression not affine
5	Number of bad regions for SCoP: Condition based on 'undef' value in basic block, 'undef' values are things without specified contents
6	Number of bad regions for SCoP: Condition in basic block neither constant nor an icmp (integer comparison) instruction
7	Number of bad regions for SCoP: Undefined operand in branch at basic block
8	Number of bad regions for SCoP: Non affine branch in basic block
9	Number of bad regions for SCoP: No base pointer
10	Number of bad regions for SCoP: Undefined base pointer
11	Number of bad regions for SCoP: Base address not invariant in current region
12	Number of bad regions for SCoP: Non affine access function
13	Number of bad regions for SCoP: Found base address alias
14	Number of bad regions for SCoP: Found invalid region entering edges
15	Number of bad regions for SCoP: Function call with side effects appeared
16	Number of bad regions for SCoP: Loop bounds can not be computed
17	Number of bad regions for SCoP: Loop not in -loop-simplify form
18	Number of bad regions for SCoP: Non canonical induction variable in loop
19	Number of bad regions for SCoP: SCEV of PHI node refers to SSA names in region

20	Number of bad regions for SCoP: Non canonical PHI node. A PHI node is canonical if it has the form `phi i32 [0, %predecessor1], [%nextvalue, %predecessor]`. The result type has not to be `i32`, it just has to be an integer.
21	Number of bad regions for SCoP: No canonical induction variable at loop header
22	Number of bad regions for SCoP: Others
23	Number of bad regions for SCoP: Found bad intToptr pointer, intToptr represents the cast from an integer to a pointer
24	Number of bad regions for SCoP: Alloca instruction
25	Number of bad regions for SCoP: Unknown instruction
26	Number of bad regions for SCoP: PHI node in exit basic block
27	Number of bad regions for SCoP: Region containing entry block of function is invalid

Table 17.: Abbreviations used in Tables 8, 9, and 10

D

EFFECT OF *-MEM2REG*

Table 18 presents the results of Experiment 4.2.4. The first column indicates the sample program. The second column shows four different experiments for each sample program:

- *original*: the statistics obtained for the original sequence without *-mem2reg*.
- *appended*: the statistics obtained for the original sequence with *-mem2reg* appended.
- *prepended*: the statistics obtained for the original sequence with *-mem2reg* prepended.
- *both*: the statistics obtained for the original sequence with *-mem2reg* appended and prepended.

The remaining columns list the statistical output of the experiment. The column names are abbreviations for the different statistical outputs that are explained in Appendix C.

Sample program	Experiment	R	S	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
7z	original	11009	178	1	1	0	272	0	15	0	124	3	0	37	93	97	243	91	2427	0	774	0	0	774	20	0	1	0	19	0
7z	appended	11009	88	1	1	0	192	0	10	0	107	2	0	21	52	81	243	66	1404	0	539	0	120	419	1455	0	1	0	1454	0
7z	prepended	11006	178	1	1	0	276	0	0	0	124	3	0	37	93	99	243	91	2421	0	774	0	0	774	20	0	1	0	19	0
7z	both	11006	88	1	1	0	192	0	10	0	107	2	0	21	52	83	243	66	1402	0	539	0	120	419	1455	0	1	0	1454	0
adi	original	19	0	0	0	0	4	0	0	0	4	0	0	0	0	9	0	0	0	0	4	0	0	4	0	0	0	0	0	0
adi	appended	19	0	0	0	0	4	0	0	0	4	0	0	0	0	9	0	0	0	0	4	0	0	4	0	0	0	0	0	0
adi	prepended	19	0	0	0	0	4	0	0	0	4	0	0	0	0	9	0	0	0	0	4	0	0	4	0	0	0	0	0	0
adi	both	19	0	0	0	0	4	0	0	0	4	0	0	0	0	9	0	0	0	0	4	0	0	4	0	0	0	0	0	0
blowfish	original	58	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	6	0	9	0	2	7	18	0	0	0	14	4
blowfish	appended	58	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	6	0	9	0	2	7	18	0	0	0	14	4
blowfish	prepended	58	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	6	0	9	0	2	7	18	0	0	0	14	4
blowfish	both	58	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	6	0	9	0	2	7	18	0	0	0	14	4
bn	original	2016	16	0	0	0	46	0	5	0	31	0	0	5	5	19	4	23	391	22	93	0	0	93	131	1	0	0	20	110
bn	appended	2016	3	0	0	0	26	0	5	0	16	0	0	3	2	10	4	16	125	22	48	0	14	34	487	1	0	0	376	110
bn	prepended	2016	16	0	0	0	46	0	0	0	31	0	0	5	5	19	4	23	391	22	93	0	0	93	131	1	0	0	20	110
bn	both	2016	3	0	0	0	26	0	5	0	16	0	0	3	2	10	4	16	125	22	48	0	14	34	487	1	0	0	376	110
cast	original	19	5	0	0	0	2	0	0	0	2	0	0	0	0	0	0	0	4	0	0	0	0	0	0	0	0	0	0	0
cast	appended	19	4	0	0	0	2	0	0	0	2	0	0	0	0	0	0	0	2	0	0	0	0	0	3	0	0	0	3	0
cast	prepended	19	5	0	0	0	2	0	0	0	2	0	0	0	0	0	0	0	4	0	0	0	0	0	0	0	0	0	0	0
cast	both	19	4	0	0	0	2	0	0	0	2	0	0	0	0	0	0	0	2	0	0	0	0	0	3	0	0	0	3	0
ccrypt	original	267	6	0	0	0	31	0	0	0	12	0	0	0	19	9	10	4	31	0	20	0	0	20	13	0	0	0	4	9
ccrypt	appended	267	6	0	0	0	31	0	0	0	12	0	0	0	19	6	10	2	16	0	14	0	4	10	39	0	0	0	30	9
ccrypt	prepended	267	6	0	0	0	31	0	0	0	12	0	0	0	19	9	10	4	31	0	20	0	0	20	13	0	0	0	4	9
ccrypt	both	267	6	0	0	0	31	0	0	0	12	0	0	0	19	6	10	2	16	0	14	0	4	10	39	0	0	0	30	9
covariance	original	15	2	0	0	0	4	0	0	0	4	0	0	0	0	6	0	0	0	0	2	0	0	2	1	0	0	0	1	0
covariance	appended	15	2	0	0	0	4	0	0	0	4	0	0	0	0	6	0	0	0	0	2	0	0	2	1	0	0	0	1	0
covariance	prepended	15	2	0	0	0	4	0	0	0	4	0	0	0	0	6	0	0	0	0	2	0	0	2	1	0	0	0	1	0
covariance	both	15	2	0	0	0	4	0	0	0	4	0	0	0	0	6	0	0	0	0	2	0	0	2	1	0	0	0	1	0

crocopat	original	2524	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	0	5	17	560	0	0	560	0	0	0	0	0	0
crocopat	appended	2524	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	0	3	17	134	0	0	134	428	0	0	0	428	0
crocopat	prepended	2524	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	0	5	17	560	0	0	560	0	0	0	0	0	0
crocopat	both	2524	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	0	3	17	134	0	0	134	428	0	0	0	428	0
des	original	198	0	0	0	0	14	0	0	0	2	0	0	0	12	0	0	13	0	0	17	0	1	16	50	0	0	0	37	13	
des	appended	198	0	0	0	0	14	0	0	0	2	0	0	0	12	0	0	13	0	0	17	0	1	16	50	0	0	0	37	13	
des	prepended	198	0	0	0	0	14	0	0	0	2	0	0	0	12	0	0	13	0	0	17	0	1	16	50	0	0	0	37	13	
des	both	198	0	0	0	0	14	0	0	0	2	0	0	0	12	0	0	13	0	0	17	0	1	16	50	0	0	0	37	13	
dsa	original	3725	27	0	0	0	50	0	7	0	31	0	0	7	5	29	2	28	501	26	116	0	0	116	201	1	0	0	59	141	
dsa	appended	3725	8	0	0	0	28	0	5	0	18	0	0	3	2	19	2	16	167	26	57	0	16	41	653	1	0	0	511	141	
dsa	prepended	3725	27	0	0	0	50	0	0	0	31	0	0	7	5	29	2	28	501	26	116	0	0	116	201	1	0	0	59	141	
dsa	both	3725	8	0	0	0	28	0	5	0	18	0	0	3	2	19	2	16	167	26	57	0	16	41	653	1	0	0	511	141	
ecdsa	original	2891	7	0	0	0	31	0	4	0	22	0	0	3	2	20	21	22	185	0	63	0	20	43	520	1	0	0	397	122	
ecdsa	appended	2891	7	0	0	0	31	0	4	0	22	0	0	3	2	20	21	22	185	0	63	0	20	43	520	1	0	0	397	122	
ecdsa	prepended	2891	7	0	0	0	31	0	0	0	22	0	0	3	2	20	21	22	185	0	63	0	20	43	520	1	0	0	397	122	
ecdsa	both	2891	7	0	0	0	31	0	4	0	22	0	0	3	2	20	21	22	185	0	63	0	20	43	520	1	0	0	397	122	
fdtd-2d	original	17	1	0	0	0	4	0	0	0	4	0	0	0	0	7	0	0	0	0	4	0	0	4	1	0	0	0	1	0	
fdtd-2d	appended	17	1	0	0	0	4	0	0	0	4	0	0	0	0	7	0	0	0	0	4	0	0	4	1	0	0	0	1	0	
fdtd-2d	prepended	17	1	0	0	0	4	0	0	0	4	0	0	0	0	7	0	0	0	0	4	0	0	4	1	0	0	0	1	0	
fdtd-2d	both	17	1	0	0	0	4	0	0	0	4	0	0	0	0	7	0	0	0	0	4	0	0	4	1	0	0	0	1	0	
fdtd-apml	original	17	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	15	0	0	15	0	0	0	0	0	0	
fdtd-apml	appended	17	0	0	0	0	5	0	0	0	5	0	0	0	0	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
fdtd-apml	prepended	17	0	0	0	0	5	0	0	0	5	0	0	0	0	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
fdtd-apml	both	17	0	0	0	0	5	0	0	0	5	0	0	0	0	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
floyd-warshall	original	11	6	0	0	0	5	0	0	0	5	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	
floyd-warshall	appended	11	6	0	0	0	5	0	0	0	5	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	
floyd-warshall	prepended	11	3	0	0	0	5	0	0	0	5	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0	2	0	
floyd-warshall	both	11	3	0	0	0	5	0	0	0	5	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0	2	0	
hmac	original	643	3	0	0	0	13	0	3	0	10	0	0	0	0	1	3	1	44	5	12	0	3	9	183	0	0	0	155	28	

hmac	appended	643	3	0	0	0	13	0	3	0	10	0	0	0	0	1	3	1	44	5	12	0	3	9	183	0	0	0	155	28
hmac	prepended	643	3	0	0	0	13	0	0	0	10	0	0	0	0	1	3	1	44	5	12	0	3	9	183	0	0	0	155	28
hmac	both	643	3	0	0	0	13	0	3	0	10	0	0	0	0	1	3	1	44	5	12	0	3	9	183	0	0	0	155	28
jacobi-1d-imper	original	9	0	0	0	0	3	0	0	0	3	0	0	0	0	1	0	0	0	0	3	0	0	3	0	0	0	0	0	0
jacobi-1d-imper	appended	9	0	0	0	0	3	0	0	0	3	0	0	0	0	1	0	0	0	0	3	0	0	3	0	0	0	0	0	0
jacobi-1d-imper	prepended	9	0	0	0	0	3	0	0	0	3	0	0	0	0	1	0	0	0	0	3	0	0	3	0	0	0	0	0	0
jacobi-1d-imper	both	9	0	0	0	0	3	0	0	0	3	0	0	0	0	1	0	0	0	0	3	0	0	3	0	0	0	0	0	0
jacobi-2d-imper	original	13	0	0	0	0	4	0	0	0	4	0	0	0	0	2	0	0	0	0	5	0	0	5	0	0	0	0	0	0
jacobi-2d-imper	appended	13	0	0	0	0	4	0	0	0	4	0	0	0	0	2	0	0	0	0	5	0	0	5	0	0	0	0	0	0
jacobi-2d-imper	prepended	13	0	0	0	0	4	0	0	0	4	0	0	0	0	2	0	0	0	0	5	0	0	5	0	0	0	0	0	0
jacobi-2d-imper	both	13	0	0	0	0	4	0	0	0	4	0	0	0	0	2	0	0	0	0	5	0	0	5	0	0	0	0	0	0
js	original	32565	259	3	3	0	846	0	57	0	701	0	0	51	37	117	302	49	3395	0	2880	0	0	2880	63	63	0	0	0	0
js	appended	32565	162	2	2	0	635	0	15	0	578	0	0	12	30	86	302	41	1300	0	1202	0	60	1142	4265	62	0	0	4203	0
js	prepended	32565	259	3	3	0	846	0	0	0	701	0	0	51	37	117	302	49	3395	0	2880	0	0	2880	63	63	0	0	0	0
js	both	32565	162	2	2	0	635	0	15	0	578	0	0	12	30	86	302	41	1300	0	1202	0	60	1142	4265	62	0	0	4203	0
lammps	original	27115	706	27	27	0	2856	0	371	0	1471	0	0	484	530	921	1362	162	5148	0	340	0	0	340	62	0	0	0	62	0
lammps	appended	27115	503	22	22	0	2279	0	344	0	1281	0	0	313	341	667	1362	109	4041	0	802	0	560	242	1742	0	0	0	1742	0
lammps	prepended	27115	706	27	27	0	2856	0	0	0	1471	0	0	484	530	921	1362	162	5148	0	340	0	0	340	62	0	0	0	62	0
lammps	both	27115	503	22	22	0	2279	0	344	0	1281	0	0	313	341	667	1362	109	4041	0	802	0	560	242	1742	0	0	0	1742	0
linpack	original	75	16	0	0	0	2	0	1	0	1	0	0	0	0	4	11	5	30	0	2	0	0	2	2	0	0	0	2	0
linpack	appended	75	10	0	0	0	2	0	1	0	1	0	0	0	0	4	11	1	21	0	9	0	8	1	8	0	0	0	8	0
linpack	prepended	75	16	0	0	0	2	0	0	0	1	0	0	0	0	4	11	5	30	0	2	0	0	2	2	0	0	0	2	0
linpack	both	75	10	0	0	0	2	0	1	0	1	0	0	0	0	4	11	1	21	0	9	0	8	1	8	0	0	0	8	0
lulesh	original	407	41	1	1	0	50	0	17	0	1	0	0	9	23	17	3	4	79	0	2	0	0	2	1	0	0	0	0	1
lulesh	appended	407	33	1	1	0	42	0	14	0	1	0	0	7	20	13	3	3	57	0	9	0	7	2	41	0	0	0	40	1
lulesh	prepended	407	41	1	1	0	50	0	0	0	1	0	0	9	23	17	3	4	79	0	2	0	0	2	1	0	0	0	0	1
lulesh	both	407	33	1	1	0	42	0	14	0	1	0	0	7	20	13	3	3	57	0	9	0	7	2	41	0	0	0	40	1
lulesh-omp	original	461	45	1	1	0	57	0	23	0	0	0	0	8	26	18	5	8	93	0	2	0	0	2	0	0	0	0	0	0
lulesh-omp	appended	461	37	1	1	0	48	0	20	0	0	0	0	6	22	13	5	5	68	0	11	0	9	2	45	0	0	0	45	0

lulesh-omp	prepended	461	45	1	1	0	57	0	0	0	0	0	0	8	26	18	5	8	93	0	2	0	0	2	0	0	0	0	0	0
lulesh-omp	both	461	37	1	1	0	48	0	20	0	0	0	0	6	22	13	5	5	68	0	11	0	9	2	45	0	0	0	45	0
md5	original	624	1	0	0	0	14	0	3	0	11	0	0	0	0	1	5	1	52	3	14	0	6	8	153	0	0	0	125	28
md5	appended	624	1	0	0	0	14	0	3	0	11	0	0	0	0	1	5	1	52	3	14	0	6	8	153	0	0	0	125	28
md5	prepended	624	1	0	0	0	14	0	0	0	11	0	0	0	0	1	5	1	52	3	14	0	6	8	153	0	0	0	125	28
md5	both	624	1	0	0	0	14	0	3	0	11	0	0	0	0	1	5	1	52	3	14	0	6	8	153	0	0	0	125	28
openssl	original	17012	112	10	8	0	260	0	33	0	143	0	0	36	48	89	119	104	2531	0	792	0	0	792	10	1	0	0	9	0
openssl	appended	17012	59	8	6	0	138	0	14	0	84	0	0	17	23	67	119	53	864	0	372	0	102	270	2362	1	0	0	2361	0
openssl	prepended	17012	112	10	8	0	260	0	0	0	143	0	0	36	48	89	119	104	2531	0	792	0	0	792	10	1	0	0	9	0
openssl	both	17012	59	8	6	0	138	0	14	0	84	0	0	17	23	67	119	53	864	0	372	0	102	270	2362	1	0	0	2361	0
postgres	original	50610	179	43	43	0	778	0	40	0	502	0	0	154	82	117	923	189	5317	0	6358	0	0	6358	21	16	0	0	5	0
postgres	appended	50610	87	32	32	0	460	0	17	0	320	0	0	98	25	105	923	124	2836	0	3911	0	195	3716	5435	7	0	0	5428	0
postgres	prepended	50610	179	43	43	0	778	0	0	0	502	0	0	154	82	117	923	189	5317	0	6358	0	0	6358	21	16	0	0	5	0
postgres	both	50610	87	32	32	0	460	0	17	0	320	0	0	98	25	105	923	124	2836	0	3911	0	195	3716	5435	7	0	0	5428	0
python	original	22086	219	8	7	0	336	0	59	0	172	0	0	68	37	114	93	25	2664	20	831	0	0	831	0	0	0	0	0	0
python	appended	22086	96	5	5	0	179	0	8	0	113	0	0	44	14	58	93	21	593	20	322	0	119	203	2914	0	0	0	2914	0
python	prepended	22086	219	8	7	0	336	0	0	0	172	0	0	68	37	114	93	25	2664	20	831	0	0	831	0	0	0	0	0	0
python	both	22086	96	5	5	0	179	0	8	0	113	0	0	44	14	58	93	21	593	20	322	0	119	203	2914	0	0	0	2914	0
rc4	original	35	0	0	0	0	1	0	1	0	0	0	0	0	0	0	3	5	0	0	5	0	0	5	11	0	0	0	10	1
rc4	appended	35	0	0	0	0	1	0	1	0	0	0	0	0	0	0	3	5	0	0	5	0	0	5	11	0	0	0	10	1
rc4	prepended	35	0	0	0	0	1	0	0	0	0	0	0	0	0	0	3	5	0	0	5	0	0	5	11	0	0	0	10	1
rc4	both	35	0	0	0	0	1	0	1	0	0	0	0	0	0	0	3	5	0	0	5	0	0	5	11	0	0	0	10	1
reg_detect	original	18	0	0	0	0	4	0	0	0	4	0	0	0	0	4	0	0	0	0	8	0	0	8	0	0	0	0	0	0
reg_detect	appended	18	0	0	0	0	4	0	0	0	4	0	0	0	0	4	0	0	0	0	8	0	0	8	0	0	0	0	0	0
reg_detect	prepended	18	0	0	0	0	4	0	0	0	4	0	0	0	0	4	0	0	0	0	8	0	0	8	0	0	0	0	0	0
reg_detect	both	18	0	0	0	0	4	0	0	0	4	0	0	0	0	4	0	0	0	0	8	0	0	8	0	0	0	0	0	0
rsa	original	1914	16	0	0	0	36	0	2	0	25	0	0	5	4	19	5	25	308	20	61	0	0	61	110	1	0	0	21	88
rsa	appended	1914	5	0	0	0	22	0	2	0	16	0	0	3	1	10	5	17	95	20	34	0	13	21	388	1	0	0	299	88
rsa	prepended	1914	16	0	0	0	36	0	0	0	25	0	0	5	4	19	5	25	308	20	61	0	0	61	110	1	0	0	21	88

rsa	both	1914	5	0	0	0	22	0	2	0	16	0	0	3	1	10	5	17	95	20	34	0	13	21	388	1	0	0	299	88
ruby	original	21789	242	5	4	0	255	0	68	0	162	0	0	7	18	101	223	102	1833	0	2002	0	0	2002	5	3	0	0	2	0
ruby	appended	21789	116	4	4	0	170	0	25	0	128	0	0	3	14	78	223	73	781	0	1058	0	104	954	2240	1	0	0	2239	0
ruby	prepended	21789	242	5	4	0	255	0	0	0	162	0	0	7	18	101	223	102	1833	0	2002	0	0	2002	5	3	0	0	2	0
ruby	both	21789	116	4	4	0	170	0	25	0	128	0	0	3	14	78	223	73	781	0	1058	0	104	954	2240	1	0	0	2239	0
sha1	original	633	13	1	0	0	26	0	5	0	21	0	0	0	0	6	3	6	172	5	28	0	0	28	0	0	0	0	0	0
sha1	appended	633	1	0	0	0	13	0	3	0	10	0	0	0	0	1	3	4	69	5	16	0	6	10	144	0	0	0	144	0
sha1	prepended	633	13	1	0	0	26	0	0	0	21	0	0	0	0	6	3	6	172	5	28	0	0	28	0	0	0	0	0	0
sha1	both	633	1	0	0	0	13	0	3	0	10	0	0	0	0	1	3	4	69	5	16	0	6	10	144	0	0	0	144	0
sha256	original	624	1	0	0	0	12	0	3	0	9	0	0	0	0	1	3	2	41	3	16	0	6	10	131	0	0	0	104	27
sha256	appended	624	1	0	0	0	12	0	3	0	9	0	0	0	0	1	3	2	41	3	16	0	6	10	131	0	0	0	104	27
sha256	prepended	624	1	0	0	0	12	0	0	0	9	0	0	0	0	1	3	2	41	3	16	0	6	10	131	0	0	0	104	27
sha256	both	624	1	0	0	0	12	0	3	0	9	0	0	0	0	1	3	2	41	3	16	0	6	10	131	0	0	0	104	27
sha512	original	646	15	2	1	0	26	0	3	0	23	0	0	0	0	8	4	5	133	2	21	0	0	21	37	0	0	0	8	29
sha512	appended	646	1	0	0	0	13	0	1	0	12	0	0	0	0	3	4	2	60	2	20	0	12	8	142	0	0	0	113	29
sha512	prepended	646	15	2	1	0	26	0	0	0	23	0	0	0	0	8	4	5	133	2	21	0	0	21	37	0	0	0	8	29
sha512	both	646	1	0	0	0	13	0	1	0	12	0	0	0	0	3	4	2	60	2	20	0	12	8	142	0	0	0	113	29
sqlite3	original	127	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	0	4	0	0	4	4	0	0	0	3	1
sqlite3	appended	127	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	0	4	0	0	4	4	0	0	0	3	1
sqlite3	prepended	127	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	0	4	0	0	4	4	0	0	0	3	1
sqlite3	both	127	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	0	4	0	0	4	4	0	0	0	3	1
ssl	original	8785	38	7	5	0	117	0	12	0	57	0	0	19	29	52	42	64	983	37	287	0	0	287	441	1	0	0	100	340
ssl	appended	8785	10	5	3	0	64	0	6	0	38	0	0	11	9	40	42	37	320	37	147	0	63	84	1358	1	0	0	1017	340
ssl	prepended	8785	38	7	5	0	117	0	0	0	57	0	0	19	29	52	42	64	983	37	287	0	0	287	441	1	0	0	100	340
ssl	both	8785	10	5	3	0	64	0	6	0	38	0	0	11	9	40	42	37	320	37	147	0	63	84	1358	1	0	0	1017	340
tcc	original	2916	34	1	1	0	25	0	2	0	16	0	0	7	0	4	405	9	293	0	1032	0	0	1032	2	0	0	0	2	0
tcc	appended	2916	4	1	1	0	24	0	1	0	16	0	0	7	0	13	405	5	193	0	980	0	7	973	166	0	0	0	166	0
tcc	prepended	2916	34	1	1	0	25	0	0	0	16	0	0	7	0	4	405	9	293	0	1032	0	0	1032	2	0	0	0	2	0
tcc	both	2916	4	1	1	0	24	0	1	0	16	0	0	7	0	13	405	5	193	0	980	0	7	973	166	0	0	0	166	0

x264	original	7515	151	18	15	0	542	0	37	0	278	9	0	63	155	86	215	78	1052	0	355	0	0	355	17	0	0	0	17	0
x264	appended	7515	86	14	12	0	419	0	30	0	241	9	0	49	90	32	215	50	543	0	349	0	178	171	786	0	0	0	786	0
x264	prepended	7515	151	18	15	0	542	0	0	0	278	9	0	63	155	86	215	78	1052	0	355	0	0	355	17	0	0	0	17	0
x264	both	7515	86	14	12	0	419	0	30	0	241	9	0	49	90	32	215	50	543	0	349	0	178	171	786	0	0	0	786	0
xz	original	1453	76	1	1	0	25	0	2	0	7	1	0	1	14	1	27	4	261	0	90	0	0	90	18	0	0	0	18	0
xz	appended	1453	57	0	0	0	19	0	2	0	7	1	0	0	9	0	27	3	96	0	49	0	9	40	239	0	0	0	239	0
xz	prepended	1453	76	1	1	0	25	0	0	0	7	1	0	1	14	1	27	4	261	0	90	0	0	90	18	0	0	0	18	0
xz	both	1453	57	0	0	0	19	0	2	0	7	1	0	0	9	0	27	3	96	0	49	0	9	40	239	0	0	0	239	0

Table 18.: Comparison of sequences with and without mem2reg

E

FITNESS COMPARISON OF OPTIMIZATION SEQUENCES

This chapter presents the results of the fitness comparison of optimization sequences applied in Experiment 4.2. Table 20 lists for every sample program the fitness values of the different sequences. The first column shows the sample program and the other columns show the fitness values of the different optimization sequences. The column names of Table 20 are explained in Table 19.

Column name	Optimization sequence
1	empty sequence (no optimization pass applied)
2	*-polly-canonicalize*
3	*-O3 -polly-canonicalize*
4	sequence of length 10 found by genetic algorithm 1
5	sequence of length 20 found by genetic algorithm 1
6	sequence of length 10 found by genetic algorithm 2
7	sequence of length 20 found by genetic algorithm 2
8	sequence of length 10 found by genetic algorithm 1 using the selected passes of Experiment 4.2.3
9	sequence of length 10 found by genetic algorithm 2 using the selected passes of Experiment 4.2.3
10	sequence of length 10 found by greedy algorithm using the selected passes of Experiment 4.2.3
11	sequence of length 10 found by hill climbing algorithm using the selected passes of Experiment 4.2.3

Table 19.: Assignment of column name to optimization sequence

Sample program	1	2	3	4	5	6	7	8	9	10	11
2mm	32	32	20	12	8	12	12	12	12	13	17
3mm	35	38	23	15	15	15	15	15	15	15	20
adi	31	46	28	19	19	19	19	19	19	19	24
atax	24	24	13	8	5	8	8	8	8	8	13

bicg	25	27	12	10	9	10	7	9	10	10	15
cholesky	24	30	17	10	9	9	9	9	9	13	16
correlation	30	39	20	15	15	16	15	15	15	16	20
covariance	27	34	18	13	13	13	13	13	13	13	18
doitgen	29	32	21	12	10	12	10	11	12	12	16
durbin	23	30	11	11	11	11	11	11	11	11	16
dynprog	23	27	11	9	9	9	9	8	9	9	14
fdtd-2d	29	40	27	16	16	16	16	16	16	16	21
fdtd-apml	29	42	20	17	17	17	17	17	17	17	22
floyd-warshall	24	27	11	5	5	5	5	5	5	10	11
gemm	27	26	15	9	9	9	9	9	9	10	14
gemver	26	34	20	13	12	13	13	13	13	13	18
gesummv	21	26	9	9	9	9	9	9	9	9	14
gramschmidt	34	46	24	19	19	20	19	19	19	20	24
jacobi-1d-imper	21	26	13	9	9	9	9	9	9	9	14
jacobi-2d-imper	25	34	17	13	13	13	13	13	13	13	18
lu	24	22	13	5	5	5	5	6	6	6	11
ludcmp	28	38	18	10	10	10	15	10	10	15	16
mvt	23	28	11	10	9	10	10	10	10	10	15
reg_detect	30	44	23	18	18	18	18	18	18	23	23
seidel-2d	23	20	11	5	5	5	5	5	5	5	11
symm	25	30	14	11	11	11	11	11	11	11	16
syr2k	27	30	15	10	10	10	10	10	10	11	16
syrk	27	26	15	8	8	8	8	8	8	9	14
trisolv	21	24	10	8	8	8	8	8	8	8	13
trmm	23	30	11	11	11	11	11	11	11	11	16

Table 20.: Fitness comparison of sequences of experiment 5 in section 4.2.5

F

FITNESS OF *-POLLY-CANONICALIZE* AND NEW FIXED PREOPTIMIZATION SEQUENCES

This chapter presents the results of the fitness comparison presented in Experiment 4.2.6. The fitness values of *-polly-canonicalize*, *-polly-preopt*, *-polly-preopt* without *-inline*, and the sequences generated by the genetic algorithms genetic 1 and genetic 2 are compared. Table 21 lists for every sample program the fitness values of the different sequences. The first column shows the sample program and the other columns show the fitness values of the different preoptimization sequences.

Sample program	polly-canonicalize	polly-preopt	polly-preopt without inline	genetic 1	genetic 2
2mm	32	12	17	12	12
3mm	38	15	20	15	15
7za	12325	14328	10944	10837	10808
adi	46	19	24	19	19
atax	24	8	13	8	8
bicg	27	10	15	9	10
blowfish	61	58	60	58	58
bn	10872	2638	2098	2045	2035
bzip2	818	561	605	613	599
cast	31	15	17	14	14
ccrypt	310	267	294	261	265
cholesky	30	9	14	9	9
correlation	39	15	20	15	15
covariance	34	13	18	13	13
crafty	6531	4807	3910	3948	3904
crocopat	2729	2535	2653	2525	2521
des	1850	209	204	198	198
doitgen	32	12	17	11	12
dsa	10776	5648	3788	3743	3740
durbin	30	11	16	11	11
dynprog	27	9	14	8	9
ecdsa	10788	4400	2933	2902	2902
fdtd-2d	40	16	21	16	16
fdtd-apml	42	17	22	17	17
floyd-warshall	27	5	11	5	5

gemm	26	9	14	9	9
gemver	34	13	18	13	13
gesummv	26	9	14	9	9
gramschmidt	46	19	24	19	19
gzip	683	653	652	611	582
hmac	10701	729	806	687	659
jacobi-1d-imper	26	9	14	9	9
jacobi-2d-imper	34	13	18	13	13
js	36354	36993	32424	32512	32443
lammps	28766	35280	26721	26706	26557
leveldb	2694	1213	2201	1223	1187
linpack	69	59	59	59	59
lu	22	6	11	6	6
ludcmp	38	10	15	10	10
lulesh	394	366	371	366	366
lulesh-omp	445	417	423	416	416
md5	10701	709	772	673	633
minisat	665	264	582	233	228
mvt	28	10	15	10	10
openssl	20021	27848	17148	17023	17037
postgres	50697	77609	50395	50321	50319
povray	12277	17815	9958	10034	9899
python	22149	31162	21901	21909	21897
rc4	42	35	38	35	35
reg_detect	44	18	23	18	18
rsa	10713	2624	1961	1918	1911
ruby	22126	28448	21583	21575	21566
seidel-2d	20	5	11	5	5
sha1	10705	713	780	674	642
sha256	10706	719	790	688	659
sha512	10706	718	789	674	653
sqlite3	2654	150	339	154	129
ssl	14238	14100	8851	8813	8808
symm	30	11	16	11	11
syr2k	30	10	16	10	10
syrk	26	8	14	8	8
tcc	3278	3076	3179	2906	2924
trisolv	24	8	13	8	8
trmm	30	11	16	11	11
x264	7564	7594	7408	7402	7389
xz	1464	1655	1385	1379	1376

Table 21.: Fitness comparison of experiment 6 in section 4.2.6

BIBLIOGRAPHY

[1] LLVM: Analysis and Transformations Passes, . URL `http://llvm.org/docs/Passes.html`. Last checked: 2014-08-06. (Cited on pages 15, 16, 35, 42, 47, 51, 57, and 65.)

[2] LLVM Command Guide, . URL `http://llvm.org/docs/CommandGuide/`. Last checked: 2014-08-06. (Cited on pages 7 and 8.)

[3] LLVM-IR: Language Reference. URL `http://llvm.org/docs/LangRef.html`. Last checked: 2014-08-06. (Cited on pages 7 and 12.)

[4] clang. URL `http://clang.llvm.org/`. Last checked: 2014-08-06. (Cited on page 4.)

[5] gcc, 2014. URL `https://gcc.gnu.org/`. Last checked: 2014-08-06. (Cited on page 4.)

[6] Polly, 2014. URL `http://polly.llvm.org/`. Last checked: 2014-08-06. (Cited on page 6.)

[7] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., 2 edition, 1986. (Cited on pages 3 and 4.)

[8] L. Almagor, Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven W. Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. Finding effective compilation sequences. In *Proceedings of the 2004 SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems (LCTES)*, volume 39, pages 231–239, New York, NY, USA, 2004. ACM. doi: 10.1145/998300.997196. (Cited on pages 5, 6, 23, 25, 28, 31, and 40.)

[9] Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. Optimizing for reduced code space using genetic algorithms, 1999. (Cited on pages 5, 23, 28, 29, 31, and 40.)

[10] Tobias Grosser. Enabling Polyhedral Optimizations in LLVM. Diploma Thesis, University of Passau, 2011. (Cited on pages 7, 10, and 14.)

[11] Tobias Grosser, Hongbin Zheng, Ragesh Aloor, Andreas Simbürger, Armin Größlinger, and Louis-Noël Pouchet. Polly - Polyhedral Optimization in LLVM. *CGO*, 2011. (Cited on pages ix, 9, and 12.)

[12] Tobias Grosser, Armin Größlinger, and Christian Lengauer. Polly - Performing Polyhedral Optimizations on a Low-Level Intermediate Representation. *Parallel Processing Letters*, 2012. (Cited on pages 9, 11, 12, 13, and 15.)

[13] Dick Grune. *Modern Compiler Design*. Worldwide Series in Computer Science. Wiley, 1 edition, 2001. (Cited on page 3.)

[14] Prasad A. Kulkarni, David B. Whalley, and Gary S. Tyson. Evaluating heuristic optimization phase order search algorithms. In *In Proceedings of the International Symposium on Code Generation and Optimization (CGO'07*, pages 157–169. IEEE Computer Society, 2007. (Cited on pages 5, 6, 22, 23, 25, 26, 28, 39, 40, and 49.)

[15] Chris Lattner. LLVM: An Infrastructur for Multi-Stage Optimization. Master's thesis, University of Portland, 2000. (Cited on page 7.)

[16] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis transformation, 2004. (Cited on page 7.)

[17] Melanie Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA, USA, 1998. ISBN 0262631857. (Cited on page 28.)

[18] Andy Nisbet. Gaps: A compiler framework for genetic algorithm (ga) optimised parallelisation. In Peter Sloot, Marian Bubak, and Bob Hertzberger, editors, *High-Performance Computing and Networking*, volume 1401 of *Lecture Notes in Computer Science*, pages 987–989. Springer Berlin Heidelberg, 1998. ISBN 978-3-540-64443-9. doi: 10.1007/BFb0037253. URL `http://dx.doi.org/10.1007/BFb0037253`. (Cited on page 28.)

[19] R.P.J Pinkers, P.M.W Knijnenburg, M. Haneda, and H.A.G Wijshoff. Analysis of computer options using orthogonal arrays. *Proceedings of the Eleventh International Workshop on Compilers for Parallel Computers*, pages 137–148, 2004. (Cited on pages 5 and 23.)

[20] Uwe Schöning. *Algorithmik*. Spektrum Akadem. Verl., 2001. ISBN 978-3-8274-1092-4. (Cited on pages 23, 26, and 28.)

[21] Andreas Simbürger, Sven Apel, Armin Größlinger, and Christian Lengauer. The potential of polyhedral optimization: An empirical study. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 508–518, November 2013. doi: 10.1109/ASE.2013.6693108. (Cited on page 36.)

[22] Yaakov L. Varol and Doron Rotem. An algorithm to generate all topological sorting arrangements. pages 83–84, 1981. (Cited on page 54.)

COLOPHON

This thesis was typeset with LaTeX 2ε using Hermann Zapf's *Palatino* and *Euler* type faces (Type 1 PostScript fonts *URW Palladio L* and *FPL* were used). The listings are typeset in *Bera Mono*, originally developed by Bitstream, Inc. as "Bitstream Vera". (Type 1 PostScript fonts were made available by Malte Rosenau and Ulrich Dirr.)

The typographic style was inspired by **?** 's genius as presented in *The Elements of Typographic Style* [**?**]. It is available for LaTeX via CTAN as "`classicthesis`".

NOTE: The custom size of the textblock was calculated using the directions given by Mr. Bringhurst (pages 26–29 and 175/176). 10 pt Palatino needs 133.21 pt for the string "abcdefghijklmnopqrstuvwxyz". This yields a good line length between 24–26 pc (288–312 pt). Using a "*double square textblock*" with a 1:2 ratio this results in a textblock of 312:624 pt (which includes the headline in this design). A good alternative would be the "*golden section textblock*" with a ratio of 1:1.62, here 312:505.44 pt. For comparison, `DIV9` of the `typearea` package results in a line length of 389 pt (32.4 pc), which is by far too long. However, this information will only be of interest for hardcore pseudo-typographers like me.

To make your own calculations, use the following commands and look up the corresponding lengths in the book:

```
\settowidth{\abcd}{abcdefghijklmnopqrstuvwxyz}
\the\abcd\ % prints the value of the length
```

Please see the file `classicthesis.sty` for some precalculated values for Palatino and Minion.

Final Version as of September 6, 2014 at 13:00.

DECLARATION

Hiermit versichere ich, dass ich diese Bachelorarbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind, sowie dass ich diese Bachelorarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

Passau, Germany, August 2014

Christoph Woller