eman ta zabal zazu
Universidad del País Vasco
Euskal Herriko Unibertsitatea
INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Degree in Computer Engineering

Computer science

End of degree work

A Combinatorial Approach for Profile Guided Optimization with Metaheuristics

Author

Mikel Malagón

2022

eman ta zabal zazu
Universidad del País Vasco
Euskal Herriko Unibertsitatea
INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Degree in Computer Engineering

Computer science

End of degree work

A Combinatorial Approach for Profile Guided Optimization with Metaheuristics

Author

Mikel Malagón

Directors

Josu Ceberio, Jose A. Pascual

Acknowledgments

First, I would like to thank Josu Ceberio and Jose A. Pascual, not only for being my directors during this work, but for serving as inspiration, for believing in me, and for being able to transmit their passion for science and research to me. I also thank Ekhiñe Irurozki for all the patience, kindness, and shared knowledge along these years. I am specially grateful to David Pello for lighting my passion for computer science.

I would also like to thank my family, from the youngest to the oldest member, and to the ones that are missing. This would not have been possible without you.

I am grateful to my lifelong friends, as well as to the ones that I met during my university degree, for gifting so many great moments. I specially thank Esteban, for accompanying me in the hardest climbing routes, as well as in the hardest university exams. Finally, I would like to thank Joana, for being as awesome as you are.

Mila esker denei.

Summary

Code optimization is one of the main tasks of modern compilers. Besides translating code from one programming language to another, or into an executable, nowadays, compilers are expected to perform numerous optimizations to the input code. In most of the cases, these process results in a highly optimized, high performance executable. One of such code optimization techniques is profile guided optimization, that distinguishes for using runtime data about the program to optimize, thus being able to more accurately optimize the different parts of the code. In this work, profile guided optimization is revisited and formalized under the combinatorial optimization paradigm. This allows to approach the mentioned code optimization problem with combinatorial optimization algorithms. Conducted experiments reveal that combinatorial optimization algorithms are a valid and realistic approach for profile guided code optimization. Moreover, being able to outperform widely used implementations of the same code optimization in some cases.

Contents

List of Figures

List of Tables

1. CHAPTER

Introduction

Compilers are computer programs that translate code from one programming language to another, while maintaining the semantics intact [Aho et al., 2007]. Usually, this translation occurs from a high level language, easier for humans to understand and reason about, to a lower level language, finally creating an executable program.

Although the main task of a compiler is translation, code optimization is also a crucial task that most compilers also have to handle. In the optimization process, code is carefully altered to improve some characteristic of the compilation result, such as runtime performance or storage size, while still guarantying the same functionality of the input code.

Although many code optimization techniques exist, in this work we approach *Profile Guided Optimization* (PGO) that makes use of profiling data during the code optimization phase [Pettis and Hansen, 1990]. The first step of this optimization technique is to compile the program in question without PGO, with the objective to execute it under some realistic workloads, while gathering profiling data. The profiling data usually contains information on the number of times the different code segments that compose the program have been executed. Then, once the profiling data has been collected, the program is compiled for a second time, but this time, profiling information is made available to the compiler. This approach enables the compiler to take decisions on how to optimize the code which are otherwise impossible, as by means of the profile data, the compiler is able to accurately determine which code segments are the most relevant for the optimization objective. However, all these optimization decisions are often conducted by generalist and

very conservative heuristics [Triantafyllis et al., 2003]. This fact causes compilers to miss many possible code optimization opportunities.

In response, the bast success of machine learning has motived numerous attempts for replacing these conservative heuristics with ML systems [Trofin et al., 2021]. However, the adoption of ML guided code optimizations in production systems presents some serious difficulties, such as, training and deployment cost of such ML models, non-determinism, or lack of proper training datasets and benchmarks [Wang and O'Boyle, 2018].

In this work, we take a different approach to improve the decision-making of PGO. We tackle PGO by formalizing it as a combinatorial optimization problem, with the objective of applying CO algorithms to solve such problem. This allows more sensible code optimization decisions while still maintaining the computational costs of such code optimizations within a reasonable margin. Moreover, formalizing PGOs as CO problems, opens the door to decades of CO research, heavily studied and production tested algorithms, as well as to future breakthroughs yet to come in the CO field.

To that end, the LLVM compiler infrastructure [Lattner and Adve, 2004] has been chosen to apply the theoretical concepts introduced in this work to real world use cases . This choice has been motivated by the relevance of LLVM in production environments, while still being in the spotlight of the research community[1]. In order to validate our proposal, we compare our CO based PGO to the one implemented in the well known clang compiler (part of the LLVM infrastructure). Concretely, we optimize the Lua and Python interpreters with our PGO approach and the one implemented in clang. Then, the optimized interpreters have been benchmarked in various computationally intensive workloads.

Results reveal that our CO based PGO approach is comparable to the one implemented in clang, broadly used in production. Moreover, our method is able to sometimes improve the results obtained by the clang compiler. However, it is worth mentioning that the aim of this work is not to reach a new state-of-the-art in code optimization. The goal of this work is to guide and motivate future works on formalizing code optimizations as CO problems, ultimately bringing decades of CO research to the field of code optimization and compilers.

[1] List of some academic research works from the official web page of the LLVM project: https://llvm.org/pubs

2. CHAPTER

Objectives and project management

This chapter describes the objectives that we aim to accomplish in this work, as well as details on the management of the project. Note that this is a research work, and implies some uncertainty that can can cause the objectives or the planned procedure to suffer some changes during the course of the project.

2.1 Objectives of the project

For the sake of guiding the development of this project, in the following lines we define a set of objectives:

1. Formalize the basic block ordering PGO as a combinatorial optimization problem, as well as analyzing the properties of the problem to get a better understanding of it.

2. Implement and analyze the behavior of various well known combinatorial optimization algorithms, finally determining which kind of algorithms are best suited for the problem we tackle in this work.

3. Understand and measure the possible benefits of our code optimization approach in real word benchmarks compared to widely used PGO implementations.

4. Develop an open source and easy to use toolkit for the usage of the proposed approach in other works.

2.2 Tasks

Based on the goals described in the previous section, in this section, we enumerate a set of consecutive tasks for the successful development of this work.

1. Develop a tool for extracting profiling data generated by LLVM.
   (a) Understand how is PGO approached in LLVM.
   (b) Determine how to parse profiling data from LLVM IR.
2. Formalize the basic block ordering code optimization as a combinatorial problem.
   (a) Determine which aspects of the basic block layout affect the runtime performance of a program.
   (b) Determine a suitable codification for the solutions.
   (c) Design an appropriate objective function.
   (d) Analyze different properties of the proposed formalization. For example, analyze if the objective function correlates with the actual real life performance.
3. Implement and/or design multiple optimization algorithms to approach the combinatorial problem formalized.
   (a) Design and implement a constructive algorithm.
   (b) Design and implement a vanilla local search and simulated annealing algorithm as an extension to the prior algorithm.
   (c) Design and implement an evolutionary approach, specifically, an estimation of distribution algorithm.
4. Compare how the implemented algorithms perform and behave under different optimization scenarios.
   (a) Generate test instances to run the algorithms on. The instances should come from real CPU intensive programs.

   (b) Determine which of the implemented CO algorithm is the best suited for the real-world application of this work.

5. Analyze how the proposed approach for basic block order optimization performs in real life benchmarks.

   (a) Generate, or collect, multiple benchmarks sufficiently representative of real-world CPU intensive workloads.

   (b) Compare our PGO approach to PGO implementations already used in industry.

   (c) Determine if our PGO approach is suitable for real-world usage.

2.3 Structure of the project

With the objective to facilitate the management and planning of the project, in this section, the tasks from the previous section have been grouped into multiple work packages, listed in the following lines.

1. **Planning and tracking**: This package includes all project management aspects of this work, also including the meetings to keep track of the progress of the project.

2. **Background**: Consist of all the work to collect the sufficient background knowledge to address the objectives and tasks presented in the previous sections. For example: reading literature on similar PGO approaches, or gathering technical knowledge on LLVM.

3. **Problem formalization**: Includes the process of formalizing the PGO code optimization as a combinatorial problem, as well as further experiments to understand and analyze the different characteristics of such formalization. This package is divided into two sub-packages:

   (a) Formalization of the PGO as a combinatorial problem, from the codification to be used, to the representation of the problem and the definition of the objective function.

   (b) Experiments to determine the characteristics that compose the proposed combinatorial problem formalization.

4. **Implementation**: This package groups all tasks related to the implantation of the project. Similar to the last package, this package also splits into multiple sub-packages.

   (a) Implementation of the necessary tooling for efficient profiling data extraction and processing, as well as combinatorial problem instance generation and solution evaluation.

   (b) Efficient implementation of multiple CO algorithms: a constructive algorithm, vanilla local search, simulated annealing, and an estimation of distribution algorithm.

5. **Experimentation**: Gathers all the tasks related to the experimentation phase of this work. Again, due to the diverse nature of this package, this group of tasks also splits into various sub-packages:

   - Experiments to determine which type of the CO algorithm is the most suitable for the real-world application of the PGO approach presented in this work.
   - Experiments to justify the usage of this PGO approach in real world scenarios, as well as to motivate future research and extensions to this work.

6. **Report & presentation**: This package includes the composition of this document, together with the development of the graphical support that will be used in the oral presentation of this work.

2.4 Time estimations

In the following lines, an estimated time distribution in terms of the work packages described in the previous section is provided. Note that this is only an estimation, and that the time distribution presented below might suffer some alterations in the course of the project. In fact, the scientific research nature of this work, makes the project even more susceptible to planning alterations.

1. **Planning and tracking**: 20 hours (includes meetings).
2. **Background**: 50 hours.
3. **Problem formalization**: 40 hours.

   (a) **Formalization** 20 hours.

   (b) **Experiments and analysis**: 20 hours.

4. **Implementation**: 100 hours.

   (a) **Profile data parsing and instance generation**: 70 hours.

   (b) **Algorithm implementation**: 30 hours.

5. **Experimentation**: 40 hours.

   - **Algorithm comparison**: 20 hours.
   - **Real-world benchmarking and comparison**: 20 hours.

6. **Report & presentation**: 80 hours.

Finally, in an effort to mitigate time distribution alterations, 20 hours have been reserved to work packages that might consume extra time than the one estimated. Therefore, summing the estimated time in each work package, and the extra 20 hours, the total time estimate to develop the project are 350 hours. The described time distribution estimation is illustrated in Figure 2.1.

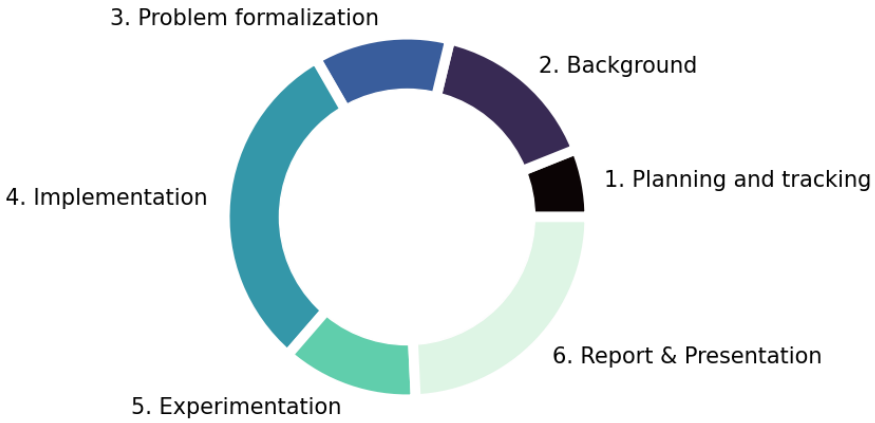


Figure 2.1: Time distribution estimation by work packages.

2.5 Resource management

In order to successfully develop this work and accomplish the objectives from Section 2.1, we estimate the usage of the following software and hardware tools:

- **Software**
  - **Overleaf**: An online and collaborative LaTeXeditor. In this work, it will be used to write all the documentation, while sharing it with the tutors.
  - **Inkscape**: Graphic design tool, that will be used to illustrate the diagrams of the documentation.
  - **Neovim**: Powerful text editor, designed for efficient coding, without getting in the way of the developer. Will be used to write all the code for the project.
  - **Rust**: We plan to use the Rust programming language and its ecosystem, to develop the majority of the software of this project. This allows us to develop very efficient code, while avoiding many security issues most of the systems programming languages are prone to have.
  - **Python**: The Python programming language and its widely used data processing ecosystem will be used to process and visualize all experimentation data.
  - **LLVM**: We plan to develop our project under the LLVM infrastructure, thus, we will use many of the tools that LLVM provides, including the clang compiler.
  - **Godbolt**: An online tool that allows to visualize and inspect the entrails of a variety of compilers, includes clang. This tool will be used to better understand LLVM IR and how source code is translated into it.
  - **Git & SourceHut**: We plan to extensively use Git, for software version control, and SourceHut (GitHub alternative), to host the Git repository.
  - **Perf & Valgrind**: Profiling tools. Both profiling tools will be used to extract relevant measurement of the optimized programs.
- **Hardware**:
  - **Own equipment**: Consists of a Lenovo Ideapad 330-15 ICH laptop, with an Intel Core i7-8750H CPU, 16GB of RAM memory and 248Gb of SSD storage. The operating system of the machine is Void Linux. It will be used to develop the entirety of the project and to run experiments with low hardware or time requirements.
  - **High-end server**: This machine will be used to conclude large scale experimentation (that usually requires not only powerful hardware, but also prolonged time) and intensive compilation. Composed of an Intel Xenon E5-1607

v3 CPU, 64Gb of RAM, and 2Tb of SSD storage. In this case, the operating system of the machine is Ubuntu 22.04 LTS.

Note that all the tooling that will be used to develop this work is open source. Moreover, all the software tools that we have planned to use are free at the time of this writing. Furthermore, this fact has a great impact in our project, as it also means that the tools that our work might depend on are open source, as this project itself. This will facilitate the integration of this project in other existing, or future, open source works, and hopefully, help to democratize technology.

2.6 Tracking and control

In this section, we revisit the goals, tasks, and estimations made in the beginning of the project, and expose the differences that occur throughout the development project. Note that this section has been written after the project has concluded.

2.6.1 Accomplished objectives

All the goals defined in the beginning of the problem have been accomplished. However, there have been some important changes:

- Due to the time constraints of the project, the experiments to analyze the characteristics of the formalized problem have been limited. We consider that these experiments would have been useful to extract more information about the problem we aim to solve, ultimately, enabling us to improve the proposed algorithms and conducted experiments.

- Again, due to the time limitations of the project, our approach has only been compared to other PGO proposals in two test cases: Python and Lua interpreters. Although, obtained conclusions are useful, conducting more experiments would allow us to extract more solid conclusions on the usage of our approach in real-world scenarios.

2.6.2 Time deviations

At the beginning of the project, we estimated some time to dedicate per work package (see Section 2.4), now that the project has concluded, in Table 2.1 time deviations that occurred during the course of the project are shown. In Figure 2.2 the information from Table 2.1 can be visualized.

Work package		Estimated	Real	Difference
Planning and tracking		20	24	+4
Background		50	62	+12
Problem formalization	Formalization	20	9	-11
	Experiments	20	8	-12
Implementation	Prof. data parsing and inst. gen.	70	52	-18
	Algorithm implementation	30	22	-8
Experimentation	Algorithm comparison	20	25	+5
	Real-world benchmarking	20	79	+59
Report and presentation		80	92	+12
TOTAL		**330 (350)**	**374**	**+44 (24)**

Table 2.1: Deviation to the estimated time per work package. Note that time is shown in hours.

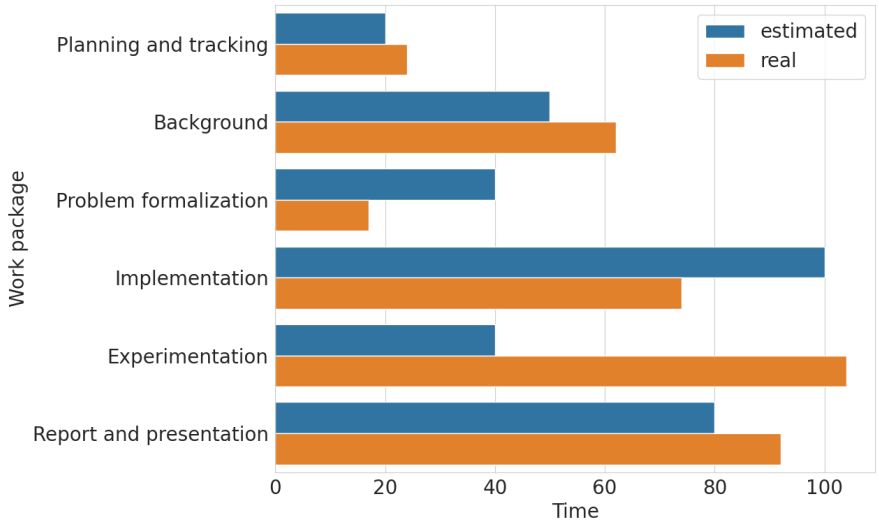


Figure 2.2: Time deviation for each work package.

As can be observed, the total time required to finish the project has been 374 hours, 44 hours more than the expected. However, the real deviation time has been reduced to 22

hours thanks to the extra 20 hours that we reserved for the work packages that might need it. Note that the greatest deviation time occurred in the *Real-world benchmarking* work package. The deviation was caused by multiple difficulties we encountered to compile Python and Lua while applying our PGO approach, as well as to some problems measuring cache misses. Finally, the time required to finish the *Report and presentation*, and *Background* packages also increased from then one we estimate, but this deviation was balanced by the time we save in the *Problem formalization* work package.

3. CHAPTER

Background

The aim of this chapter is to briefly introduce the reader to the most relevant topics for this work, with the objective to better understand the contributions of this work presented in coming chapters.

3.1 Compilers

As aforementioned, compilers are computer programs that translate code from one programming language to another [Aho et al., 2007], see Figure 3.1. Commonly, this translation is made from a higher level language to a lower level one, ultimately, generating an executable program. For example, from the C programming language to a `x86_64` executable binary.

In the early days of compilers, the source program was directly compiled into the target language. However, as the complexity of both, high-level programming languages, and low-level languages such as the instruction sets of modern CPUs increased, this approach soon became unscalable.

Nowadays, in order to mitigate this problem, compilation is divided in stages, see Figure 3.2. Instead of directly compiling the high-level source code to low-level code, the front end stage of the compiler, translates the input program into an intermediate representation (IR). Commonly, the IR is a machine independent assembly, and it is only for the internal use of the compiler, as can be seen in Figure 3.2. Then, the middle end stage takes the

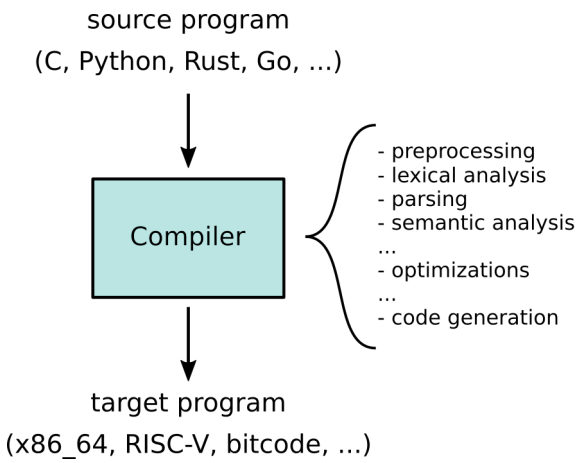


Figure 3.1: Typical compiler workflow. The source code is fed into the compiler, where after numerous transformation passes, it results in the target program.

generated IR and applies a series of transformations to it, generally, code optimizations. Finally, the back end stage is in charge of translating the optimized IR into the target language, usually, into a binary executable.

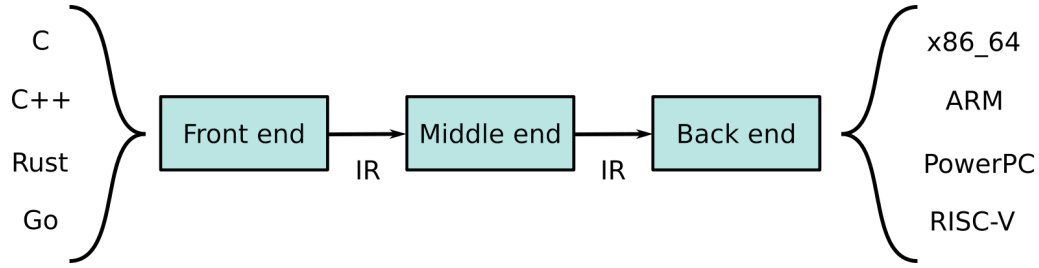


Figure 3.2: Architecture of a modern compiler. Each stage translates the higher level input into a lower level language representation.

An example of such process is illustrated in Figure 3.3, where the different representations that the same program takes during the compilation stages are shown. In the upper left block the source program is shown, in this case, a simple `add` function programmed in C. Then, in the green block, the IR of the C program can be seen, in this case, corresponds to the IR that LLVM uses. Finally, the blue block contains the output program in the target language, in this case, the same `add` function in `x86_64` assembly.

3.2 LLVM IR

In the workflow of a compiler, the vast majority of code optimizations occur in the middle end stage of the compilation process. Consequently, almost all code optimizations are

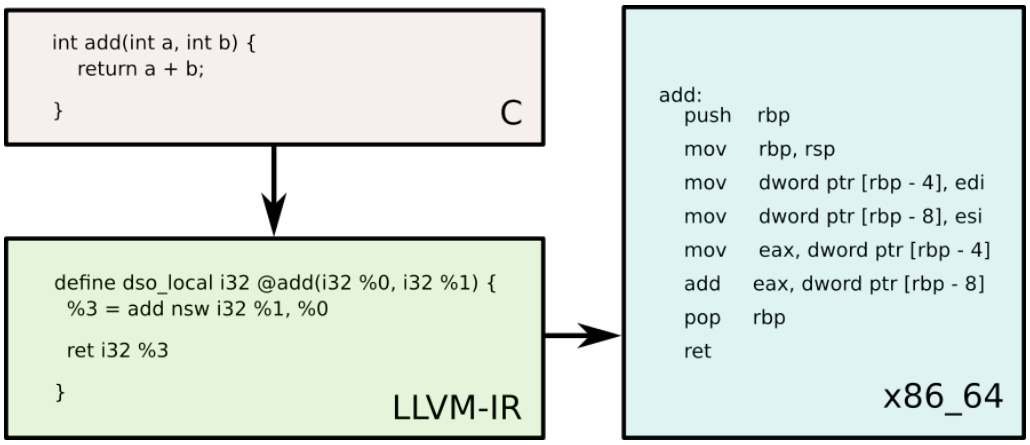


Figure 3.3: Different representations of the same program during compilation stages. The first block, in red, represents the input program (front-end stage). The next block, in green, shows the IR representation of the input program (middle-end stage). The last block, in blue, shows the assembly representation of the program (back-end stage).

performed over the IR of the program. In the specific case of this work, as we make use of LLVM, the IR at hand is LLVM IR[1], the intermediate representation of LLVM. Therefore, the following lines provide a brief introduction on LLVM IR, relevant to understand the contribution of this work.

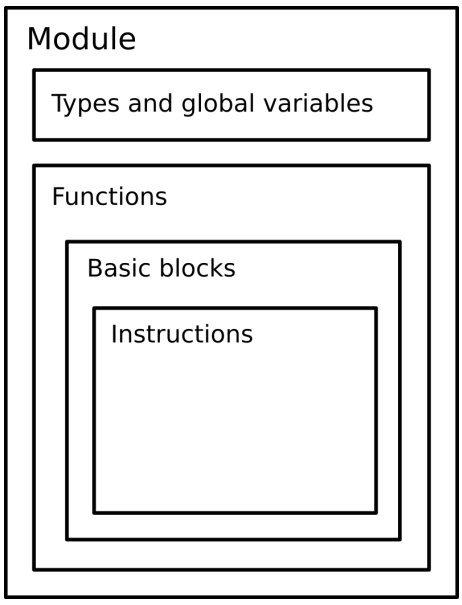


Figure 3.4: Hierarchical organization of LLVM IR programs.

Programs in LLVM IR are organized in a hierarchical structure (see Figure 3.4). The most abstract member of this hierarchical structure are modules, where each module cor-

[1]LLVM IR Reference manual: https://llvm.org/docs/LangRef.html

responds to a translation unit of the input program (sometimes translation units correspond to files in the source program, but this relation is not necessarily true). Modules themselves are composed of type declarations and global variables, for example, function declarations.

Descending into the hierarchical structure, the next abstraction layer correspond to functions. Functions in LLVM are very similar to functions in many mainstream programming languages, such as C or Java.

In turn, functions are composed of basic blocks, which consist of code segments that are sequentially executed (with no branching in between). These blocks have a single entry point and a single branch instruction, that is the last instruction of the basic block, referred to as the terminator (see Figure 3.6). Finally, basic blocks are composed of IR instructions, that represent the most basic unit of execution in LLVM IR.

With respect to this work, the most relevant member of the hierarchical structure of LLVM IR, is the basic block. In fact, the specific PGO we address in this work is the *basic block ordering optimization* described in the next section.

3.3 Profile guided optimization

Profile guided optimization is a type of code optimization performed by compilers to an input program [Pettis and Hansen, 1990]. This type of code optimization characterizes for using profiling data of the program to optimize. In turn, profiling data is the data collected in a process called *profiling*. Profiling is a type of dynamic analysis of a program, this means that the program is analyzed dynamically, at runtime (while execution). The data collected in this process, usually contains the frequency in which the different code segments that compose the analyzed program have been executed.

This type of dynamic analysis is done either by an external program that repeatedly analyzes the program at hand (this method is called *sampling profiling*, the most utilized tool for this job is `perf` [2]), or by the compiler itself, that injects extra code in the input program in order generate profile data in runtime [3] (this method is referred as *instrumentation profiling*).

[2] Link to the webpage of the perf tool: https://perf.wiki.kernel.org/index.php/Main_Page

[3] Guide on how to perform instrumentation profiling with clang: https://clang.llvm.org/docs/UsersManual.html#profiling-with-instrumentation

The advantages of exposing profile data to the compiler are numerous. For example, profile data provides the compiler with information on which code segments are the most relevant, in consequence, the compiler might apply more aggressive optimizations in such code segments.

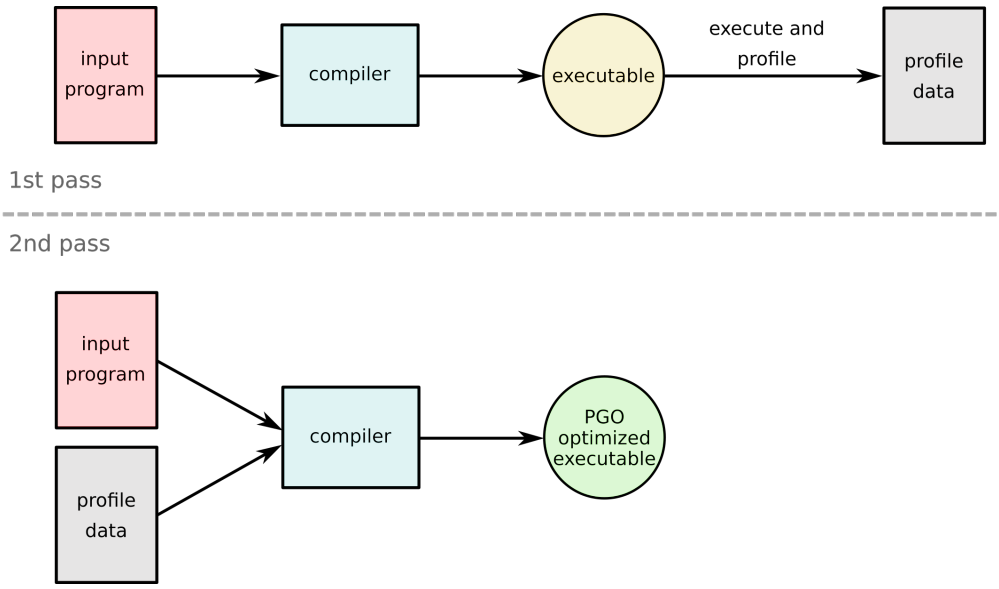


Figure 3.5: Typical workflow of PGO.

However, PGO comes with a noticeable cost. In order to apply PGO, first, profile data has to be collected, and to profile the program, first the program has to be compiled. With the aim to clarify the last statement, in Figure 3.5 the typical workflow of PGO is illustrated. In the first compilation pass (upper diagram), the program is compiled without PGO, as there is no profile data yet. Then, the compiled program is run in some benchmarks [4], while gathering profile data. Once the profile data is collected, a second compilation pass is performed (bottom diagram of Figure 3.5). In this pass, the compiler is fed with the original source code of the program, together with the profile data. Now, the compiler will be able to perform PGO, resulting in a highly optimized compilation result.

Finally, note that PGO does not perform just one optimization, in fact, some code optimizations that are usually done without profile data also benefit from profile data (for example, function inlining), as with profile data, optimizations can be more accurately done. As this work is a proof of concept for the introduction of CO in PGO, and the ob-

[4]Note that these benchmarks should mimic the real workload that the program will handle once optimized. Else, the generated profile data will not correspond to real life scenarios, ultimately, causing the PGO optimized program to perform poorly.

jective is not to reach state-of-the-art performance, in this work we focus in a particular code optimization: *basic block ordering*.

3.3.1 Basic block ordering optimization

Although PGO can perform various types of code optimizations, in this work we focus on a particular code optimization, *basic block ordering*, [Pettis and Hansen, 1990]. As its name suggests, basic block ordering is a code optimization over the basic blocks that constitute a function.

More precisely, the objective of this optimization is to order basic blocks such that blocks that frequently branch between each other, are located close in the layout of the final executable binary.

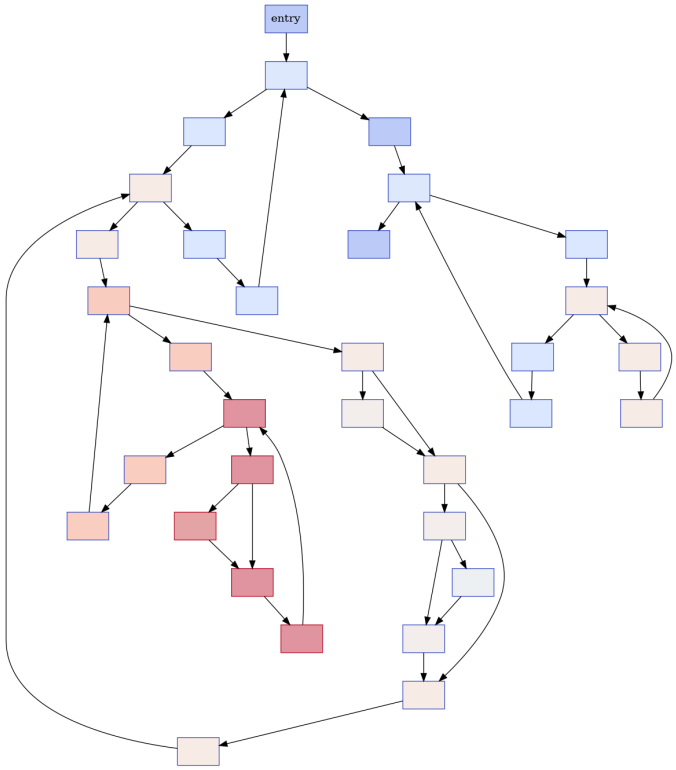


Figure 3.6: Basic block branch graph of a simple function. Basic blocks are represented as colored boxes, while the branches between them are illustrated with the black arrows. Note that the topmost block is the entry block of the function.

In Figure 3.6 the branch graph of a simple function is shown. In this figure, basic blocks are illustrated as squares, and branches are represented by arrows between blocks. Note that the entry block (the root node in the graph), can not be ordered, as this would change

the behavior of the function. Therefore, in the rest of the work and in the optimization itself, the entry block will be completely ignored and left unchanged.

By optimizing the basic block layout of the final executable program, the objective is to reduce cache and TLB misses, ultimately, resulting in better runtime performance. Note that this kind of code optimization barely affects small programs, as small programs can easily fit into modern CPUs cache memory. On the contrary, this optimization is relevant to programs of considerable size, such as, artificial intelligence applications, web browsers, compilers, or interpreters. For that reason, considering the rapid growth in size of modern software projects, this kind of code optimization is very relevant nowadays, [Lavaee et al., 2019, Panchenko et al., 2021].

3.4 Combinatorial optimization

Combinatorial optimization (CO) [Papadimitriou and Steiglitz, 1998] is a field that aims to solve combinatorial problems. In turn, in these problems, the objective is to find the optimal solution from a finite set of candidate solutions of the problem. More precisely, combinatorial problems are defined by a two element tuple, the set of all possible solutions, also known as *search space* (Ω) and the objective function, denoted as $f : \Omega \rightarrow \mathbb{R}$.

As aforementioned, the search space of a problem is the set of all possible solutions $\mathbf{x} \in \Omega$ to the problem. In other words, Ω is the set of solutions that satisfy all the constraints imposed by the combinatorial problem in hand. Although by definition, the size of this set is always finite, usually, the cardinality of Ω grows exponentially with the size of the instance. This fact makes many combinatorial problems extremely difficult or practically impossible to approach with classical techniques such as, *brute force* or *branch and bound*.

The other element of combinatorial problems is the objective function (also referred to as *fitness function*). This function is used to measure how optimal a solution is, and it is defined as a mapping between the search space Ω and $\mathbb{R}$, formally, $f : \Omega \rightarrow \mathbb{R}$.

Depending on the combinatorial problem at hand, the optimal solution, $\mathbf{x}^* \in \Omega$, is the solution that minimizes or maximizes such objective function. In a minimization problem, $\mathbf{x}^*$ is the solution that minimizes the objective function, therefore, $\mathbf{x}^* = \text{argmin}_{\mathbf{x}\in\Omega} f(\mathbf{x})$. On the contrary, in a maximization problem, $\mathbf{x}^* = \text{argmax}_{\mathbf{x}\in\Omega} f(\mathbf{x})$.

Finally, it is worth mentioning, that many CO problems belong to the family of *NP-Hard*

problems, [Arora and Barak, 2009]. These problems constitute a specially difficult to approach set of problems, of great interest for both academia and industry. Specifically, there is not a known algorithm to solve any of such problems in polynomial time, nor even for verifying if a solution is a global optima or not. Some of these well known problems are: the *Permutation Flow Shop Problem* (PFSP) [Allahverdi et al., 2008], the *Quadratic Assignment Problem* (QAP) [Koopmans and Beckmann, 1957], and the *Linear Ordering Problem* (LOP) [Ceberio et al., 2015].

4. CHAPTER

Formalizing profile guided optimization as a combinatorial problem

Although multiple PGOs exists, in this work, we focus on the *basic block ordering* optimization. The objective of this code optimization is to reorder all the basic blocks of a function, except the entry block (first block), such that blocks with high interaction are placed closer in the final executable.

As explained in Section 3.3.1, the entry block of a function can not be reordered. For the sake of clarity, in the rest of the paper, the entry block is omitted, and the preceding block of the entry block is considered to be the first block.

4.1 Codification

Let us consider a function with n basic blocks to order, where each block is different, and none of them can be repeated in the ordering. The codification for such ordering naturally falls into the permutation representation. Given a permutation σ, the i-th element of such permutation, σ_i, codifies that the σ_i-th block of the original function will be placed in the i-th position of the modified function. Note that under this definition, the identity permutation ($\sigma_i = i$, where $i = \{1,\ldots,n\}$) corresponds to the ordering of the blocks in the original function. In other words, the identity permutation is equal to applying no modifications to the original basic block layout.

Furthermore, all possible permutations of length n are valid solutions for the problem, and

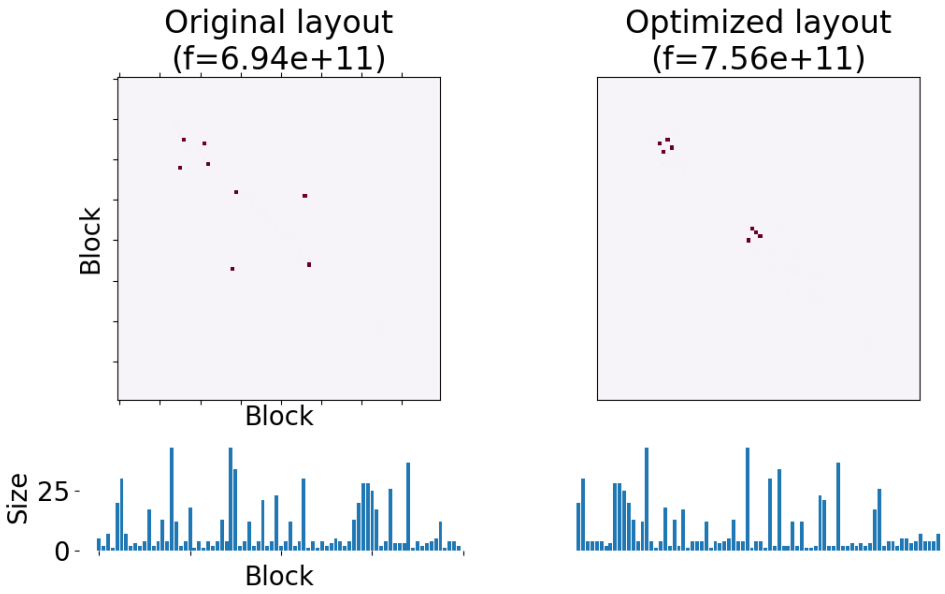


Figure 4.1: Comparison of the block layout of a low quality solution (left size figures) with the resulting block layout of a high quality solution. Upper figures show the interaction matrix (denoted as C) of the basic blocks, where darker colors mean higher interaction. On the other hand, the bottom figures represent the sizes of the basic blocks (the s vector).

thus, the search space of solutions is defined as the symmetric group of n items, $\mathbb{S}_n$. In consequence, the size of the set of all possible solutions grows exponentially with respect to the number of basic blocks to order, in fact, the cardinality of the search space is: $|\Omega| = n!$. This exponential relation justifies the usage of non-exact methods, heuristics, to approach this problem, as a relatively small function of 20 basic blocks has $20! = 2.43 \times 10^{18}$ possible orderings.

4.2 Objective function

Consider a matrix $C = [c_{i,j}]_{n \times n}$, where $c_{i,j} \in \mathbb{N}$ and the element $c_{i,j}$ corresponds to the number of times the basic block i branches to the basic block j. Let us also denote a vector $\boldsymbol{s}$, where $s_i \in \mathbb{N}^+$ is the size of the i-th basic block, determined by the number of instructions that compose the function. Accordingly, the objective function is defined as the following formula,

$$f(\sigma) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \left[(c_{\sigma_i,\sigma_j} + c_{\sigma_j,\sigma_i}) \left(\rho - \sum_{k=i}^{j} s_{\sigma_k} \right) \right] \tag{4.1}$$

where the term, $c_{\sigma_i,\sigma_j} + c_{\sigma_j,\sigma_i}$, denotes the amount of interaction between blocks σ_i and σ_j, the term $\sum_{k=i}^{j} s_{\sigma_k}$ refers to the distance between the same blocks in their final placement, and the term ρ denotes the total length of the input program, $\rho = \sum_{i=1}^{n} s_i$. Consequently, the objective of the optimization process is to find a solution σ^* that maximizes the objective function $f(\sigma)$,

$$\sigma^* = \underset{\sigma \in \mathbb{S}_n}{\operatorname{argmax}} f(\sigma) \tag{4.2}$$

where solutions that better fit $f(\sigma)$, are solutions that prioritize the proximity of blocks that branch between each other. Note that solutions with lower objective value than the identity permutation (that corresponds to the original block layout to optimize), are considered not beneficial for the optimization task, as they do not improve the original basic block layout. Therefore, the objective function value of the identity permutation is used as a baseline value in the experiments presented in Chapter 7.

In order to prove the validity of the proposed objective function, Figure 4.1 is presented. This figure shows the resulting block layout of a low quality solution (figures on the left) and the layout of a high quality solution (right side figures).

The upper heatmaps represent the C matrix, where $C_{i,j}$ is the interaction between the i-th and j-th block, and the bottom histograms represent the $\mathbf{s}$ vectors of both solutions. As can be seen in Figure 4.1, the solution with higher fitness (right), places high interaction blocks (yellow cells) closer, proving that the objective function models the real objective. On the other hand, the histograms in the bottom section represent the s size vectors, where the i-th bar represents the size of the σ_i-th block.

5. CHAPTER

Algorithms for approaching the proposed problem

In this work, with the objective of analyzing the behavior and performance of different CO algorithms under the presented PGO problem, four different CO algorithms have been tested: A constructive algorithm, two local search methods (vanilla local search [Pirlot, 1996] and simulated annealing [Bertsimas and Tsitsiklis, 1993]) and an Estimation of Distribution Algorithm [Lozano et al., 2006].

This algorithm selection is guided by the fact that code optimizations must occur fast and efficiently in real life, for example, in industry environments, where time requirements are crucial. However, besides implementing fast algorithms, the constructive algorithm and vanilla LS, in this work, we have also considered more powerful and computationally more expensive algorithms, the SA and an the EDA. Finally, as code optimization must occur fast, and real-life instances normally contain hundreds (or even thousands) of blocks to optimize, exact solver algorithms have been discarded for this work.

In the following sections, the already mentioned optimization algorithms are described, together with a pseudocode implementation and the list of hyperparameters [1] (used throughout the whole work).

[1] Note that the values of the presented hyperparameters are not a product of an extensive tuning process. This is a proof of concept work, thus, the objective is not to reach the stat-of-the-art. Therefore, the values of the hyperparameters might be far from optima.

5.1 Constructive Algorithm

The first algorithm that we propose for the CO problem introduced in this work is a constructive algorithm. Constructive algorithms are algorithms that start from an empty solution, and iteratively construct a good solution for the problem. These algorithms are problem specific, as they are only designed to perform well in a specific problem. In fact, constructive algorithms heavily rely on prior knowledge about the problem at hand, usually requiring expert knowledge on the problem. Although, (usually) constructive algorithms fail to reach the high-quality results of more complex heuristic algorithms, normally, constructive algorithms require much less computational resources than its counterparts, while reaching acceptable results.

Algorithm 1 Constructive algorithm

$s \leftarrow$ intialize empty list
$R \leftarrow$ blocks ranked by amount of interaction (higher to lower)
$front \leftarrow$ **false**
while s not complete **do**
  sample ρ from a categorical distribution, where, $p(\rho = i) = R_i^{\tau_1}$ and $\rho \in \{i | i \notin s,\ i = 1 \ldots, n\}$
  if front **then**
    push item ρ as the first element of s
  else
    push item ρ as the last element of s
  end if
  front $\leftarrow$ **not** front
  $U^\rho \leftarrow \{i | i \notin s, c_{\rho,i} \neq 0 i = 1, \ldots, n\}$
  while $|U^\rho| > 0$ **do**
    sample ν from a categorical distribution, where, $p(\nu = i) = R_i^{\tau_2}$ and $\nu \in \{i | i \notin s,\ i \in U^\rho\}$
    if front **then**
      push item ν as the first element of s
    else
      push item ν as the last element of s
    end if
    front $\leftarrow$ **not** front
    $U^\rho \leftarrow U^\rho \cap \{\nu\}$
  end while
end while

In the specific case of this work, time requirements are specially relevant (code optimiza-

tions must occur fast), while obtaining very high quality results is not particularly crucial, therefore, constructive algorithms are a great candidate for this specific scenario.

The constructive algorithm implemented in this work is presented in Algorithm 1. Note that the constructive algorithm is not deterministic, as it relies on multiple sampling steps from random distributions. However, reproducible results are possible if the random seed is fixed. In the following lines, Algorithm 1 is explained step by step.

The first step is to rank the code blocks by amount of interactions, this ranking is denoted R, where $R_i = j$, only if $\sum_{k=1}^{n} c_{i,k} > \sum_{k=1}^{n} c_{l,k}$, and $l = j+1,\ldots,n$ (note that $c_{i,j}$ denotes the amount of interaction of the i-th block with the j-th block, notation from Chapter 4). Before continuing to the next step, the partial solution s is initialized to an empty list.

Then, a parent item ρ has to be selected, where $\rho \in \{i | i \notin s, i = 1,\ldots,n\}$, and refers to a basic block of the function to optimize. The parent item is randomly sampled from a categorical distribution, where the probability of selecting a specific item x from such distribution is determined by the ranking of x in the R vector. Concretely, $p(x = i) = R_i^{\tau_1}$, where the exponent $\tau_1 \in \mathbb{N}$ is a hyperparameter that determines the likelihood of a high rank item to be selected, the effect of τ is illustrated in Figure 5.1.

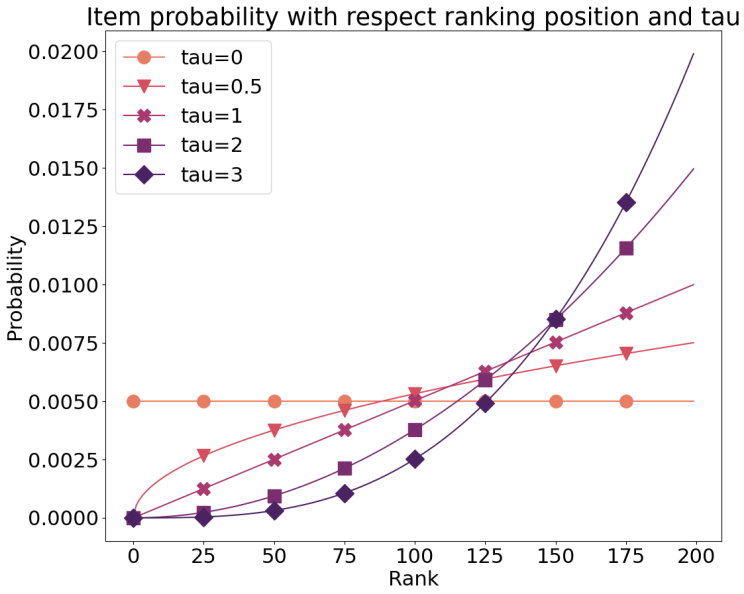


Figure 5.1: Effect of the τ parameter over the probability of items to be selected.

Once the parent item ρ is selected, it is added to the partial solution s, and the set of its children blocks, U^{ρ}, is computed from items that are not members of the partial solution

s, and have non-null interaction with the parent item ρ, formally, $U^{\rho} = \{i | i \notin s, c_{\rho,i} \neq 0, i = 1,\ldots,n\}$. Then, the following procedure is repeated until U^{ρ} is empty: randomly select a child $v \in U^{\rho}$ from a categorical distribution where $p(v = i) = R_i^{\tau_2}$, add v to s, and remove v from U^{ρ} After this procedure, if s is complete, the algorithm stops, else, the algorithm loops back to the parent item (ρ) selection step. Note that, as described in Algorithm 1, items are added to the partial solution s alternately, where an item is added to the front of s if the last item was added as the last element of s, and vice versa.

Finally, throughout the whole extension of this work, the values of both τ parameters of the constructive algorithm have been set to $\tau_1 = 2$ and $\tau_2 = 3$.

5.2 Local Search

With the objective of implementing a more advanced heuristic than the described constructive algorithm, while still remaining relatively fast and simple, in this section we introduce the well known local search (LS) algorithm [Pirlot, 1996].

Algorithm 2 Local search

```
s* ← radom solution
while stopping criterion not met do
    updated ← false
    for s in N(s*) do
        if f(s) > f(s*) then
            s* ← s
            updated ← true
            break
        end if
    end for
    if not update then
        break
    end if
end while
return s*
```

As described in Algorithm 2, the first step is to initialize the best solution found so far (s^*) to a randomly generated solution. Then, the neighborhood of s^*, $N(s^*)$ is traversed until a better solution than s^* is found. If there is no solution better than s^* in its neighborhood, the algorithm ends. Else, s^* gets updated with the new best solution and the algorithm loops back to the inspection of $N(s^*)$.

Note that the described LS algorithm always jumps to better solutions, and if there is no solution in $N(s^*)$ better than s^*, it stops. This early stopping mechanism, causes the algorithm to prefer exploiting over exploring. In consequence, this algorithm often suffers for premature convergence by getting trapped in a local optima.

Finally, although many neighborhood functions $N(s)$ have been proposed in the literature on the symmetric group $\mathbb{S}^n$, in this paper we consider the *adjacent swap* neighborhood function, chosen by its simplicity and very lower computational cost.

5.3 Simulated Annealing

The simulated annealing algorithm (SA) [Bertsimas and Tsitsiklis, 1993] is a well known algorithm that has been widely used in industry as well as in academia for decades. SA is an extension to the LS algorithm introduced in the previous section. As aforementioned, once LS finds a local optimum value, the algorithm stops. SA extends the LS algorithm with the aim to better explore the search space, and avoid getting trapped in a local optima. Concretely, with certain probability, SA is able to select solutions that are worse than the current solution, thus avoid premature convergence. In turn, this probability depends on a temperature value, that gets decreased through the optimization process, reducing the probability to select lower quality solutions as the optimization process advances.

The particular SA implemented in this work (see Algorithm 3) is very similar to the vanilla SA from [Bertsimas and Tsitsiklis, 1993], except some minor changes to adapt it to the basic block ordering problem presented in this work.

First, the best solution, s^* and the current solution, s, are initialized to some randomly generated solution. The value of the current temperature, T, is also initialized with some initial value T_0 Then, a random neighbor of s is selected, denoted as s', and its energy value is calculated, ΔE. If the energy value is positive, then, s' is better than s, and thus, s gets updated with s'. In this same step, if s' is better than s^*, s^* also gets updated with s'. In the other hand, if the energy of s' has negative value, s gets the value of s' with probability $e^{\frac{\Delta E}{T}}$. Lastly, if the equilibrium criterion is met, the temperature T gets decreased by some ratio δ and the algorithm returns to the s' generation step. Else, the algorithm returns to the same step without updating T, note that in case of meeting the stopping criterion, the algorithm stops its execution.

As the vanilla SA algorithm [Bertsimas and Tsitsiklis, 1993], assumes a minimization problem, the only notable change introduced to the vanilla SA in this work, has been its

Algorithm 3 Simulated Annealing

```
s* ← radom solution
s ← s*
T ← T_0
while stopping criterion not met do
    while equilibrium condition not met do
        s' ← random neighbor of s
        ΔE ← f(s') − f(s)
        if ΔE > 0 then
            s ← s'
            if f(s) > f(s*) then
                s* ← s
            end if
        else
            with probability e^(ΔE/T) do: s ← s
        end if
    end while
    T ← δT
end while
return s*
```

adaptation to a maximization problem. This adaptation only consists in a subtle change in the original probability term $e^{\frac{-\Delta E}{T}}$, that has been replaced with $e^{\frac{\Delta E}{T}}$.

Finally, similarly to the LS algorithm, the neighborhood function has also been set to the *adjacent swap* neighborhood function. On the other hand, the initial temperature has been set to, $T_0 = 10^5$, the temperature decay to $\delta = 0.95$, and the equilibrium condition has been set to 100 iterations.

5.4 Estimation of Distribution Algorithm

The Estimation of Distribution Algorithm (EDA) [Lozano et al., 2006], is the only evolutive algorithm considered in this work. Although being considerably more computationally expensive than the other introduced algorithms, EDAs have demonstrated great versatility for a very broad range of domains and CO problems [Ceberio et al., 2012].

Specifically, the implemented EDA in this paper is the Univariate Marginal Distribution Algorithm (UMDA) described in this work [Malagon et al., 2020], and firstly introduced in [Pelikan and Mühlenbein, 1999] by Pelikan and Mühlenbein.

Considering a set $\mathcal{X} = \{x^1, x^2, \ldots, x^m\}$ of solutions, then the univariate marginal distribution defines the probability of solution x based on the univariate marginal frequency of solutions in $\mathcal{X}$. This is computed counting the number of times a specific item appears in a concrete position in the solutions that compose $\mathcal{X}$. Therefore, for any solution $x \in \Omega$ its probability is computed as,

$$P(x) = \prod_{i=0}^{n-1} p(x_i = j) \tag{5.1}$$

where $p(x_i = j)$ is the probability of the item i to appear in the j-th position of x, that is derived from the univariate marginal frequency explained above. This is denoted as *first order marginal probability*.

Algorithm 4 Estimation of Distribution Algorithm

```
S ← population of radom solutions
while stopping creterion not met do
    U ← select the best k solutions from S
    compute the univariate marginal frequencies of the solutions in U
    generate a new set Q by sampling solutions following Equation 5.1
    S ← U ∪ Q
end while
return s*
```

Based on the explained probabilistic model, the adopted UMDA in this work is described in Algorithm 4. First, a population of random solutions, S is generated. Then, the best k number of solutions are selected from S, generating a set of *survivor* solutions, U. From this new set U, a univariate marginal probability distribution is leaned by computing the univariate marginal frequency of the solutions in U. Once the distribution is learned, a new set of solutions, Q, is sampled from such distribution by adapting the classical sampling method for integer vectors (as the classical method does not guarantee sampling permutations). The particular sampling method consist of: for each position $1 \leq i \leq n$, a value is randomly sampled in the range of x_i with probability $p(x_i)$, then, the probability of sampling the item x_i in any other position is negated, and the probabilities of the rest of the items get normalized. Finally, the population, denoted as S, is redefined as the union of the sets U and Q, $S = U \cup Q$.

In all the conducted experiments, the hyperparameters of the described EDA have remained the same. The population size has been set to 300 solutions, where, at each iteration, the best 100 solutions are selected by truncation to estimate the probability distribution, from which 200 new solutions are sampled.

6. CHAPTER

Implementation

This chapter gives relevant implementation details of the theoretic concepts described in the previous chapters, as well as the technical decisions made in the development stage of the project.

As mentioned in Chapter 3, the first step in PGO is to collect the profiling data. In this work, we opted to gather profiling data via instrumentation profiling. In instrumentation profiling, the compiler smartly injects extra code (mostly consisting of counters) to the output program such that when executed, it creates a file containing all the profiling data. In fact, in this work, as we used clang to generate the instrumented executables, it has only required to add an extra flag to the compilation command [1]. Note that in order to generate enough and accurate profiling data, the instrumented program might be run multiple times with different benchmarks, generating various profile data files that get merged via specific tools, in this case, provided by LLVM (specifically, `llvm-profdata`).

As already mentioned in Chapter 3, PGO operates over IR, in this case as we developed the project on top of LLVM, our PGO approach operates over LLVM IR. Therefore, as shown in the upper section of Figure 6.1, once the profile data is available, it is fed, together with the program to optimize, to the compiler. As a result, an annotated LLVM IR representation of the program is generated, containing the profiling information as metadata. This process is also handled by the clang compiler.

Then, the annotated LLVM IR, is parsed (with a custom tool developed for this project)

[1] Link to the followed instructions: https://clang.llvm.org/docs/UsersManual.html#profiling-with-instrumentation

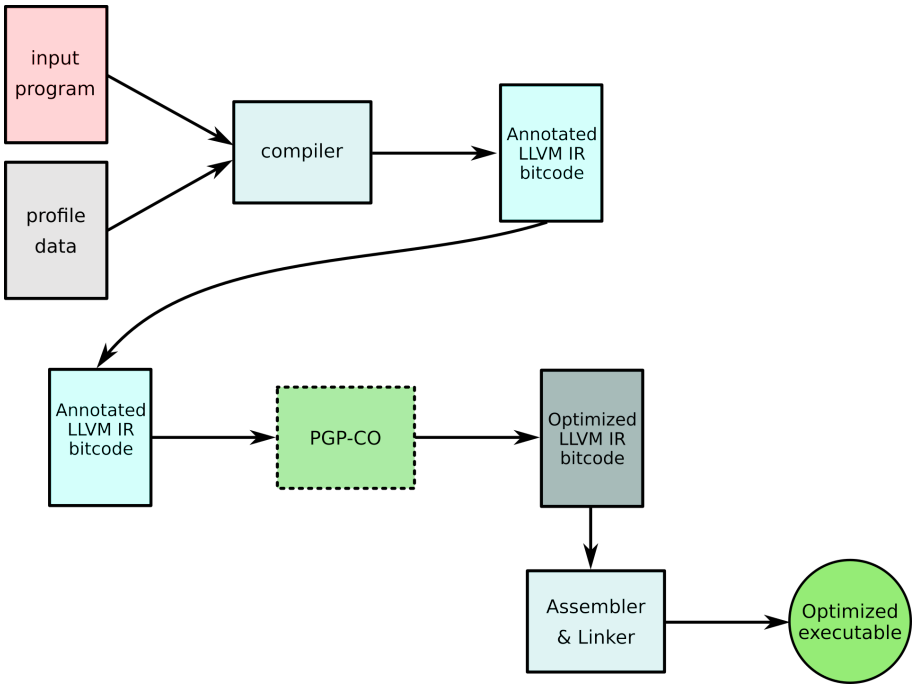


Figure 6.1: PGO process, as implemented in this work. Note that the diagram skips the step to collect the profiling data.

in order to generate an instance file, that contains the C matrix and the s vector (notation from Chapter 4) of the functions of interest in the input LLVM IR. With this instance file, a separate tool, also developed as part of this project, optimizes the LLVM IR with the CO algorithms from Chapter 5, generating a final, optimized LLVM IR file.

The final step of the workflow, see Figure 6.1, involves generating an actual executable file from the optimized LLVM IR. This particular job is accomplished by the assembler, that generates machine specific assembly from the LLVM IR file, and the linker that is used to link the executable to the needed libraries. In this case, the employed assembler and linker, are also part of the LLVM infrastructure.

6.1 Developed tools

In order to accomplish the different goals listed in Chapter 2, and to facilitate the integration of the ideas introduced in this work into other projects, a set of easy to use tools have been developed. In this section, a brief description of the different tools developed for this project is given.

All the tools described in the lines below have been developed with the Rust programming

language, a relatively new, systems programming language, with great focus on safety and runtime performance [Matsakis and Klock II, 2014]. However, the LLVM project is developed in C++, and there are no official bindings for Rust [2]. Although multiple LLVM interoperability libraries are available in the Rust ecosystem, at the time of this work, high or mid-level [3] LLVM libraries in Rust are not mature enough and lack some relevant features for this work, mostly related to LLVM IR metadata (used to parse profiling data). Therefore, for the sake of better interoperability with the C++ LLVM library, we have decided to use the `llvm_sys` [4] Rust library, that provides low-level, and almost feature-complete, bindings to the C API of LLVM.

Finally, note that, all the software developed for this project has been freely distributed under the *GPLv3* license [gpl, 2007] as an open source project, available here [5].

These are the developed tools during the course of the project:

- **LLVM IR inspector**: The purpose of this tool is to inspect the annotated LLVM IR (see Figure 6.1). Given the path to a LLVM IR bitcode file (binary representation of LLVM IR), this tool parses the bitcode and the metadata it might contain, and dumps information about the profiling data to the standard output. Dumped information contains, the C and s vector of each function, and a final summary on the profiling data, for example, the maximum number of basic blocks encountered, or the number of functions that contain profiling data. This tool has been developed for inspecting programs before optimizing them with the tools listed below.

- **Instance generator**: Given a LLVM IR bitcode file, this tool generates a JSON instance file containing a list of elements consisting of: name of the function, the C matrix, and the s vector (see Chapter 4). This tool provides a variety of flags to decide which functions of the input LLVM IR to include in the instance file, such as, only include the k number of most called functions.

- **Optimizer**: This is the most relevant tool developed for this work. Given a LLVM IR file, and an JSON instance file, this tool optimizes the LLVM IR file with the CO algorithms from Chapter 5. The user might also specify the algorithm to use, as well as the maximum optimization time per function. Note that if the objective function

[2] At the time of this work, besides C++, LLVM only has official support for C, in the form of an API that exposes most of the C++ library. See: `https://llvm.org/doxygen/group__LLVMC.html`

[3] With high or mid-level libraries, we refer to libraries that build elaborated abstractions on top of low-level features in order to provide an easier development experience.

[4] Repository of the library: `https://gitlab.com/taricorp/llvm-sys.rs`

[5] Link to the repository of the project: `https://git.sr.ht/~mikelma/pgo_co`

value (see Section 4.2) of the generated solution is worse than the objective value of the identity permutation, the solution is considered *harmful*, and the basic block layout will not be altered from the input to the output program.

7. CHAPTER

Experimentation

The aim of this chapter is to describe and analyze the set of experiments conducted throughout the project. First, the algorithms introduced in Chapter 5 are compared with the objective to determine which type of CO algorithm is the best suited for the optimization task we tackle in this work. The second part focuses on the practical aspect of this work, that is to compare our CO based PGO to the widely used PGO of the `clang` compiler.

7.1 Algorithm comparison

As already explained, we formalize the basic block ordering PGO as a CO problem to be able to apply CO algorithms to approach it. The literature on CO is vast [Blum and Roli, 2003], and the number of CO algorithms applicable to the mentioned problem is immense. Therefore, the objective of this section is to analyze the performance, cost, and behavior of the CO algorithms described in Chapter 5, and to determine which type of CO algorithms are best suited for approaching the CO problem introduced in this work.

7.1.1 Experimental setup

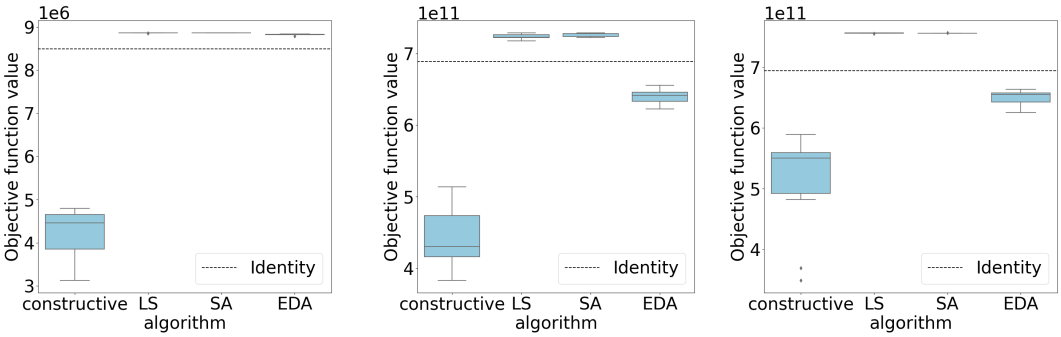
For the purpose of comparing the mentioned CO algorithms, three optimization instances, with different sizes, have been generated: `gol-main`, `gol-evolve` and `fluidSim`. All the three instances have been generated from CPU intensive programs. Specifically, the first

two instances have been extracted from a *Game Of Life* implementation, and the last one, from a fluid simulator [1], both programmed in the C programming language. The size, or the number of basic blocks, of the instances are, respectively: 20, 30, and 81.

Finally, all the experiments presented in this section have been executed in a machine with the following characteristics: *Intel i7-8750H* CPU at 2.20GHz, 16Gb of DDR4 RAM.

7.1.2 Experimental results

In this experiment, the performance of the proposed CO algorithms is compared in terms of the objective function value of the best solution found. Each algorithm has been run 10 times in each benchmark. Finally, in to model a real-world compilation scenario, the stopping criterion has been set to a maximum optimization time of 10 seconds. Note that the LS algorithm features an early stopping mechanism, that causes the algorithm to often stop before the maximum optimization time is reached.



(a) Instance: *gol-main* ($n = 20$) **(b)** Instance: *gol-evolve* ($n = 30$) **(c)** Instance: *fluidSim* ($n = 81$)

Figure 7.1: Comparison of the presented CO algorithms by means of the best solution found.

Results for the three optimization benchmarks are shown in Figure 7.1. In the figures, the results of each algorithm are shown, together with the baseline value, the objective function value of the identity permutation (equal to the original ordering of the basic blocks in the function to optimize, see Chapter 4).

Regarding the obtained results, in the smallest instance (see Figure 7.1a), `gol-main`, all algorithms show similar performance, except the constructive algorithm that is not able to improve the original ordering of the basic blocks. In the remaining, benchmarks results are practically identical, `gol-evolve` ($n = 30$) in Figure 7.1b, and `fluidSim` ($n = 81$) in

[1] Link to the repository: https://github.com/davidedc/Ascii-fluid-simulation-deobfuscated

Figure 7.1c. The quality of the solutions generated by the constructive algorithm continue to be under the baseline value, while the results of both local search based algorithms show the best performance. Conversely to the results observed in Figure 7.1a, the EDA performs under the baseline value in these bigger sized benchmarks.

Although it is clear that both local search algorithms, LS and SA, are the algorithms that best perform in the tested benchmarks, deciding which of the algorithms is the best performer is very difficult considering only the results shown in Figure 7.1.

Therefore, in order to decide which of the local search based algorithm performs the best, a statistical test, introduced in [Arza et al., 2022] has been conducted. This test is used to compare two random variables and determine which takes the best values. For the purpose of this test, LS and SA have been run 100 times in the `gol-evolve` instance and the objective value of the best solution found has been recorded. Then, the results of each algorithm have been considered the values taken by two random variables, that have be compared with the mentioned statistical test.

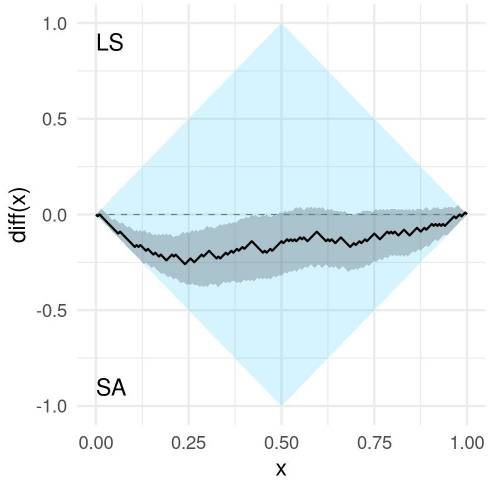


Figure 7.2: Cumulative difference plot of the results obtained by LS and SA in terms of the best solution found. Note that the gray contour refers to the confidence interval.

The obtained cumulative difference plot is shown in Figure 7.2. In this figure, the X axis represents quantiles, and the Y axis the difference between the obtained scores. Thus, on the left side of the plot the best results are compared ($x < 0.5$, quantiles above the median), and on the right side, the worst results ($x > 0.5$, quantiles below the median). For example (see Figure 7.2), in $x = 0.25$ the value of y is negative (the black line is curved towards SA), this means that the results obtained by SA in the 25% quantile are better than the ones from LS. As observed in Figure 7.2, the difference values are always

negative, towards SA, meaning that SA has greater probability to obtain better results than LS in all the quantiles. Moreover, the difference value in $x < 0.5$ is greater (more negative) than in $x > 0.5$, thus, SA has even higher probability to obtain better results than LS in high quality scores (upper quantiles).

7.1.3 Understanding the results

In this section, additional experiments are presented to justify the results seen in Figure 7.1. More precisely, the aim of these experiments is to answer the question: *Why the local search algorithms perform better than the EDA?*

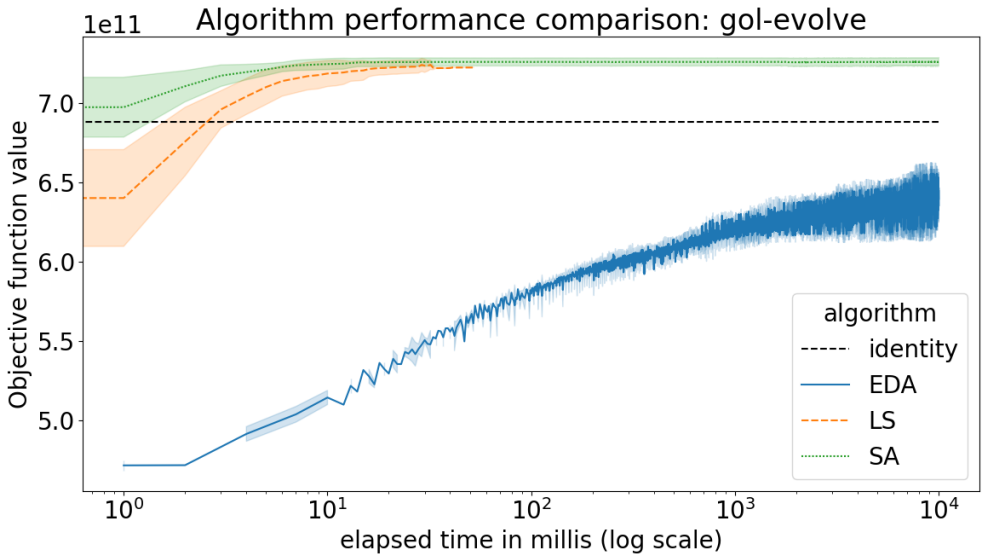


Figure 7.3: Evolution of the best solution found over time. Note that the X axis is shown in logarithmic scale. Contours refer to the standard deviation of the obtained values.

The first step has been to analyze the evolution of the best objective value along the optimization process. For this experiment, the algorithms have been run 10 times in the `gol-evolve` instance, results are shown in Figure 7.3. As can be seen, both local search methods rapidly converge to a high-quality solution, at around 100 milliseconds of elapsed time, while the EDA converges much slower and to a substantially lower quality solution (below the baseline value).

With the objective to understand the results of Figure 7.3, a new experiment has been conducted. This time, instead of focusing on the performance of the algorithms in terms of objective function values, we decided to visualize the number of evaluations that the

algorithms compute during the optimization process. Results are shown in Figure 7.4. As expected, SA is the fastest algorithm, evaluating considerably more solutions than its counterparts in the same time period. On the other hand, the vanilla LS algorithm and the EDA, evaluate a similar number of solutions per millisecond (both lines overlap in the first 100 milliseconds). This might be caused by the fact that LS compares all the solutions of the neighborhood, and that the sampling process of the EDA adds considerable overhead to the algorithm. However, the fact that LS and EDA have similar evaluation per time ratio, and that even then, LS obtains considerable better results than the EDA, suggests that the LS method is more efficient finding high quality results than the EDA in this specific CO problem.

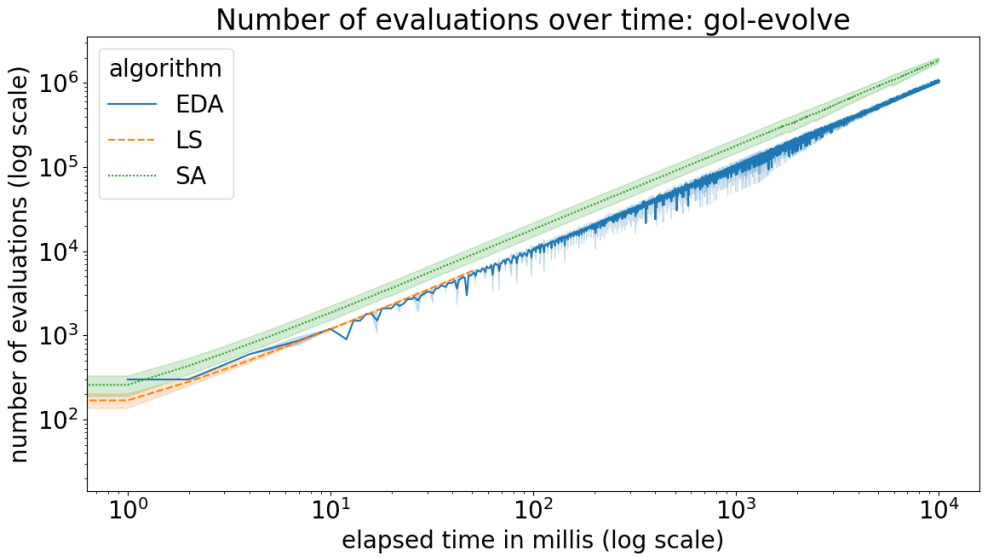


Figure 7.4: Number of evaluations over time. Note that both axes are shown in logarithmic scale.

In order to prove this hypothesis, results employed to generate Figures 7.3 and 7.4 have been used to create Figure 7.5. The figure illustrates the objective function value of the best solution found over the number of evaluations computed. As can be observed, SA is the algorithm with the best efficiency in terms of finding high-quality solutions to the problem at hand, closely followed by the vanilla LS, while the efficiency of the EDA is considerably lower.

Now that the results of all additional experiments have been exposed, in the following we provide some final thought on the obtained conclusions. The first experiment, shown in Figure 7.1, clearly demonstrates that the constructive algorithm is the algorithm that performs worst, followed by the EDA, and that both LS based algorithms are the ones that perform the best.

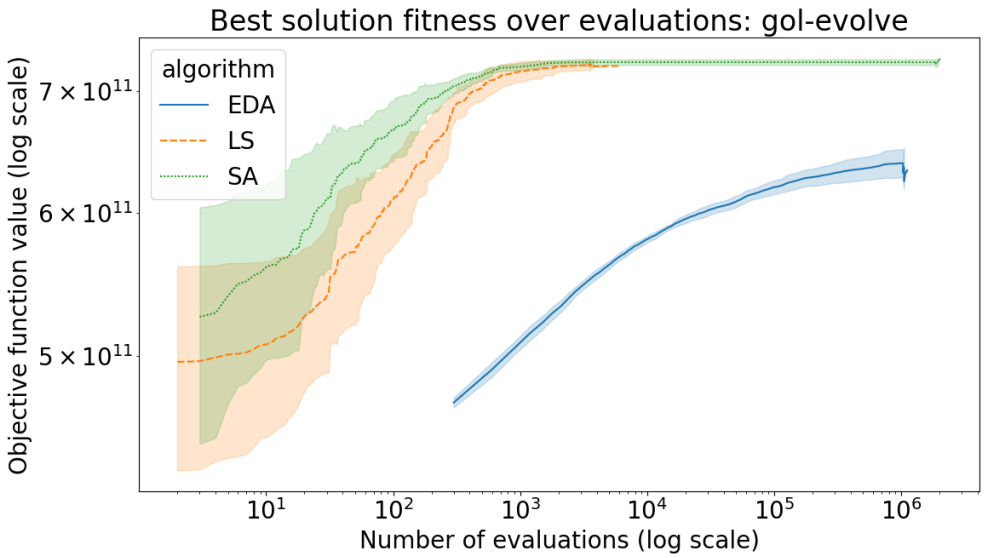


Figure 7.5: Objective function value of the best solution found over evaluations. Note that both axes are in logarithmic scale.

Next, in Figure 7.2, the rope between the vanilla LS and SA is solved, showing that the results of SA statistically dominate the results of vanilla LS. Later, in Section 7.1.3, obtained results are further explored, empirically demonstrating that LS and SA are the most efficient algorithms approaching the combinatorial problem in hand.

Finally, as mentioned in the first lines of this chapter, the main purpose of these experiments is to determine which algorithm is best suited to tackle the basic block ordering CO problem. At this point, the reader might assume that SA is the algorithm that best suits these job, however, SA converges slower than the vanilla LS, causing SA to spend more time optimizing its results. This extra time makes the optimization of a whole program (normally composed of hundreds or even thousands of functions) to be much slower compared with the vanilla LS. As the results obtained by the vanilla LS are very close to the ones obtained by SA (see Figure 7.1), and vanilla LS converges much faster than SA (see Figure 7.3), we consider vanilla LS to be the algorithm that best suits this optimization problem, for the practical reasons explained.

7.2 Real life benchmark comparison

In this section, our CO based PGO approach is compared to the broadly used PGO implementation of the clang compiler. More specifically, we compare both PGO approaches

by optimizing two very relevant real-world programs: the Lua and Python interpreters. Lua [Ierusalimschy, 2006] is a lightweight embeddable programming language, focused on efficiency and simplicity. On the other hand, Python [Van Rossum and Drake, 2009] is one of the most popular programming languages in the world (at the time of this work) [StackOverflow, 2021], and it is the *de-facto* programming language in artificial intelligence. Once both interpreters have been optimized by the different PGO approaches, we run the interpreters on a variety of CPU intensive benchmarks to measure relevant metrics for the code optimization we tackle in this work.

Although multiple interpreter implementations exist for Lua and Python, the ones chosen for this work have been: the original Lua v5.4.4 interpreter developed in PUC Rio[2] and the reference implantation of Python, the CPython[3] interpreter, maintained by the Python project itself.

As mentioned, the optimized interpreters have been run in multiple CPU intensive benchmarks. For the sake of reducing noise in the measurements of interest, the benchmarks used for both interpreters have been selected for their CPU intensive nature, and low I/O and system overhead. In the case of the Lua, benchmark programs have been extracted from an open source Lua benchmarking project [4]. On the other hand, Python benchmarks have been selected from a well known webpage on programming language benchmarking: *The Computer Language Benchmarks Game* [5].

7.2.1 Experimental setup

The procedure to generate a PGO optimized program with clang has been the one described in Section 3.3. On the other hand, the complete procedure to generate a PGO optimized program using our approach is explained in Chapter 6. With respect to the employed CO algorithm by our approach, as experiments performed in Section 7.1 determined that LS is the most suitable algorithm for this task, LS has been used to optimize the programs presented in the following experimentation. In addition, in order to fairly compare both PGO approaches, in both cases, the profile data used to optimize the programs has been exactly the same.

[2]Link to the mentioned Lua interpreter: http://www.lua.org/ftp/lua-5.4.4.tar.gz

[3]Link to the repository in the specific commit used in this work: https://github.com/python/cpython/tree/a365dd64c2a1f0d142540d5031003f24986f489f

[4]Link to the mentioned repository in the specific commit of its usage in this work: https://github.com/gligneul/Lua-Benchmarks/tree/e1b3f24c035799d92f3350a62bda557fba739d47

[5]The mentioned website: https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html

With the objective to measure the effects of the different PGO approaches, in this section, we have decided to measure the following metrics:

- Elapsed time in user space. Measures the time that the program spend in user space though the whole execution. This time measure does not consider the time that the process spends in kernel mode, that is irrelevant for the objective of these work. In this work, we make use of this measurement as a numerical quantification of the runtime performance.
- First-level (L1) instruction cache miss rate: Is the ratio between the number of L1 instruction cache misses and the total number of instruction cache accesses.
- Last-level (LL) instruction cache miss rate: Is the ratio between the number of LL instruction cache misses and the total number of instruction cache accesses.

In order to fairly measure elapsed time and minimize the inevitable noise introduced by the operating system, the optimized programs have been run 20 times in each benchmark. The specific tool used to measure elapsed time has been the well known `perf` Linux profiling tool. In the case of the reminding metrics, both have been measured using `cachegrind`[6] a cache and branch-prediction profiler. However, in the case of both cache measurements, benchmarks have been only run once, as `cachegrind` simulates the cache memory of a real machine, resulting in deterministic and noise-free results.

Finally, all the experiments we present in this section have been generated in a Linux machine with an Intel Xeon E5-1607 v3 CPU.

7.2.2 Experimental results

Experiment results are provided in Tables 7.1 and 7.2, where the average values from 20 repetitions are shown for benchmark and PGO approach pair. As a summary, in both tables the last row gives the average improvements per metric [7]. Finally, note that our approach is referred to as *pgo-co* in the rest of the experimentation.

Experimental results on the Lua benchmarks are shown in Table 7.1. Focusing on pure runtime performance (see elapsed time column), our PGO approach is able to produce slightly better results than the clang counterpart. With respect to the L1 instruction cache

[6] Homepage of the project: https://valgrind.org/docs/manual/cg-manual.html

[7] The improvement percentage for a metric x has been computed as, $100 \cdot ((x_{\text{clang}}/x_{\text{pgo-co}}) - 1)$

miss rates (second column), the results obtained by both approaches are similar: in half of the benchmarks, our PGO approach performs better, while in the other half, clang is able to produce better results. However, in the *mandel* benchmark, our approach is able to improve the L1 cache miss rate by orders of magnitude when compared to the result of its counterpart. On the other hand, the LL miss rate obtained by clang is almost systematically better than the one obtained by our approach.

The results of the Python benchmarks are shown in Table 7.2. When looking at the left-most column, the runtime performance obtained by both approach are even more similar than in Table 7.1. The only difference with the results obtained in the Lua benchmark, is that in this case, the Python interpreter optimized by the PGO of clang performs 0.19% better on average compared to the one optimized by our PGO proposal, thus, results are practically equivalent in average.

Nonetheless, when comparing the L1 instruction cache miss rates, our PGO approach reaches better results than its clang counterpart in almost every benchmark. In fact, our PGO, improves the L1 cache miss rate of the clang optimized executable by almost 14% in average. The most improvement occurs in the *spectral norm*, where the python interpreter optimized by our PGO almost reduces in half the cache misses of the interpreter optimized by the PGO implementation of clang.

Benchmark	Elapsed time (s)		L1 miss rate		LL miss rate	
	clang-pgo	pgo-co	clang-pgo	pgo-co	clang-pgo	pgo-co
ack	**8.36e-02**	8.37e-02	**3.65e-06**	4.16e-06	**1.88e-06**	3.34e-06
binary-trees	3.39e-03	**3.02e-03**	5.58e-04	**2.70e-04**	**1.33e-04**	2.02e-04
fannkuch-redux	2.72e-03	**2.58e-03**	**1.67e-04**	1.98e-04	**7.57e-05**	1.66e-04
fasta	2.30e-03	**2.07e-03**	1.41e-03	**9.26e-04**	**2.10e-04**	3.67e-04
fixpoint-fact	**2.28e-03**	2.36e-03	5.19e-04	**2.36e-04**	**1.54e-04**	1.95e-04
heapsort	**3.63e-02**	3.66e-02	**1.12e-05**	1.16e-05	**4.35e-06**	9.09e-06
mandel	2.02	**2.01**	2.02e-03	**2.34e-07**	**1.09e-07**	1.66e-07
n-body	4.10e-03	**3.98e-03**	2.43e-04	**1.96e-04**	**5.70e-05**	1.21e-04
queen	**4.95e-03**	5.84e-03	**8.30e-05**	8.92e-05	**4.08e-05**	7.39e-05
scimark	**2.38e01**	2.39e01	9.61e-06	**1.61e-06**	9.83e-07	**5.10e-07**
sieve	4.14e-02	**4.01e-02**	**7.71e-06**	1.06e-05	**3.74e-06**	8.31e-06
spectral-norm	**2.82e-02**	2.85e-02	**1.15e-05**	1.29e-05	**5.15e-06**	1.06e-05
Avg. improv.:	1.28%		72270.36%		-32.76%	

Table 7.1: Experimental results for all Lua benchmarks. The best result for each benchmark and measurement pair has been boldfaced, and the last row shows the average improvement of our approach with respect to the clang compiler.

Finally, when looking at the LL instruction cache miss rate (rightmost column in Table 7.2), our PGO implementation is able to outperform the PGO of clang in every benchmark except *pidigits*. However, the average improvement on LL cache misses is not as impressive as in the case of the L1 miss rate, in this case, our implementation improves the one in clang by 0.98% in average.

Benchmark	Elapsed time (s)		L1 miss rate		LL miss rate	
	clang-pgo	pgo-co	clang-pgo	pgo-co	clang-pgo	pgo-co
startup	8.39e-04	**7.59e-04**	1.33e-02	**1.28e-02**	3.27e-04	**3.22e-04**
n-body	1.06	**1.06**	1.55e-04	**1.49e-04**	3.46e-06	**3.45e-06**
fannkuchredux	**1.40e01**	1.54e01	**2.16e-03**	2.62e-03	2.26e-07	**2.24e-07**
spectral norm	**8.82e02**	8.88e02	6.57e-03	**3.67e-03**	3.56e-09	**3.49e-09**
pidigits	**3.41**	3.41	3.55e-04	**3.25e-04**	**8.35e-07**	8.36e-07
k-nucleotide	**2.11e-01**	2.16e-01	1.65e-02	**1.56e-02**	1.35e-04	**1.34e-04**
Avg. improv.:	-0.19%		13.99%		0.98%	

Table 7.2: Experimental results for Python benchmarks. The best result for each benchmark and measurement pair has been boldfaced, and the last row shows the average improvement of our approach with respect to the clang compiler.

8. CHAPTER

Conclusions & Future work

Code optimization is a hot topic in the compiler research community. However, in the last decades, the complexity of both, the software projects and the hardware that executes them has greatly increased. Consequently, the number of different code optimization scenarios that modern compilers have to face is vast. Most compilers approach this fact by applying conservative and generalist code optimization decisions, with the objective to address the maximum number of scenarios possible, though, causing to miss many code optimization opportunities.

With the objective to minimize the mentioned issue, Profile Guided Optimization (PGO), makes use of profiling data at compile time. Profiling is the process in which a program is dynamically inspected at runtime to collect information on how frequent different code sections get executed. With this information available, the program is compiled again, this time being able to exploit more code optimization opportunities. However, optimizing a program based on profiling data continues to be a greatly complex problem by itself, that is usually approached by simple heuristics, again, possibly missing code optimization opportunities.

In recent years, the multidisciplinary and continuously growing popularity of ML, has driven multiple attempts to replace the mentioned heuristics with more *intelligent* ML models. However, the learning and inference cost of such ML models, together with the need for vast datasets, have push back the introduction of these approaches in real-world environments.

In this work, we propose a new approach for PGO based on CO and metaheuristics. Con-

cretely, we take a specific PGO as a case of study, the *basic block order optimization*, and formalize it as a combinatorial problem. This code optimization consists of ordering chunks of code in such a way that the proximity between code blocks that frequently call each other is minimized, ultimately improving the number of cache misses, and thus, the runtime performance of the whole program.

Under this definition, the *basic block order* code optimization has been formalized as a permutation problem, where the number of basic blocks of the function to optimize, n, determines the number of possible solutions to the problem, specifically, $(n-1)!$. This causes the problem to quickly become intractable for exact solvers, justifying the need for heuristic algorithms.

In this work, several experiments have been run with the objective to empirically determine which types of CO algorithms are the best suited for the problem at hand. The tested algorithms have been: a constructive algorithm, vanilla local search, simulated annealing and a univariate marginal distribution algorithm (a type of EDA). Conducted experiments reveal that local search based algorithms (LS and SA) are the most performant algorithms under the strict time requirements that enforce compilers and code optimizations. Moreover, although SA reaches better results than LS, due to its fast convergency and good results, LS has been selected as the best suited algorithm for the mentioned optimization problem.

With the objective of validating our CO based PGO approach, in the second part of the experimentation, our proposal has been compared to the PGO implemented in the widely used clang compiler. Specifically, the Lua and Python interpreters have been optimized with the both PGO approaches and tested in a variety of CPU intensive benchmarks. Results reveal that our PGO approach is comparable to the one implemented in clang, even improving the L1 cache miss rate of its counterpart as far as 13.99% in some cases. Moreover, experiments demonstrate that the introduced PGO approach in this work is a realistic alternative to techniques used in industry grade compilers.

However, this work also presents its limitations and there are a number of possible improvements and open research lines that we think are interesting to develop in future works:

1. Extend the experimentation to more benchmarks and use cases. Extending the experimentation to other programs would help to reach more solid conclusions. Furthermore, this would allow us to better understand the behavior of our PGO proposal, and known when and why our approach performs better than others. How-

ever, due to the time constraints that this work has had, the number of test instances explored in this work has been limited. These are the lines which we think are most interesting to extend the experimentation towards:

   (a) Perform more repetitions on the already present benchmarks. This reduces the effect of the noise in the collected results, ultimately allowing to obtain more solid conclusions for the experimentation.

   (b) Test beyond interpreters. In this work, the real-life comparison of our PGO approach to the one implemented in clang has only been tested in one type of program: interpreters. It would be interesting to extend the experimentation to other types of programs sensible to runtime performance, such as: web browsers, OS kernels or artificial intelligence.

2. Improve hyperparameter values. Currently, the hyperparameters of the presented CO algorithms has not been almost tuned. Hyperparameter tuning could greatly improve the performance of the employed algorithms, possibly resulting in considerable improvements of the results obtained by our PGO approach.

3. Develop a more *globally aware* optimization approach. Currently, the basic block ordering optimization is performed in a per-function basis. However, the basic blocks of a function can also contain calls to other functions, that in turn, are composed of other basic blocks. The proposal is to consider the blocks of other functions when optimizing the layout of a specific function. This would help to further improve the cache locality of the code blocks, allowing a better optimization of the whole program.

4. Further investigate the characteristics of the proposed CO problem and its formalization. Knowing more about the specific problem we tackle would help to improve the algorithms we already present in this work, or even guide the development of new algorithms specific to address the problem at hand. For example, the constructive algorithm presented in this work could be greatly improved.

5. Develop a clang plugin. Currently, the whole project is developed in a standalone Rust project. Although, the project has been thought to facilitate the usage of our PGO proposal in existing codebases and other projects, developing a plugin for the clang compiler would considerably improve this integration.

6. Extend to other code optimizations. Finally, we think that the approach developed in this work is widely applicable, in other PGO optimizations (for example, in func-

tion ordering) as well as in other types of code optimization. By formalizing code optimizations as combinatorial problems, these task could be better understood, resulting in more advanced algorithms that could enhance already existent code optimizations.

Bibliography

[gpl, 2007] (2007). Gnu general public license, version 3. Last retrieved 2020-01-01.

[Aho et al., 2007] Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2007). *Compilers: principles, techniques, & tools*. Pearson Education India.

[Allahverdi et al., 2008] Allahverdi, A., Ng, C. T., Cheng, T. E., and Kovalyov, M. Y. (2008). A survey of scheduling problems with setup times or costs. *European journal of operational research*, 187(3):985–1032.

[Arora and Barak, 2009] Arora, S. and Barak, B. (2009). *Computational complexity: a modern approach*. Cambridge University Press.

[Arza et al., 2022] Arza, E., Ceberio, J., Irurozki, E., and Pérez, A. (2022). Comparing two samples through stochastic dominance: a graphical approach. *arXiv preprint arXiv:2203.07889*.

[Bertsimas and Tsitsiklis, 1993] Bertsimas, D. and Tsitsiklis, J. (1993). Simulated annealing. *Statistical science*, 8(1):10–15.

[Blum and Roli, 2003] Blum, C. and Roli, A. (2003). Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM computing surveys (CSUR)*, 35(3):268–308.

[Ceberio et al., 2012] Ceberio, J., Irurozki, E., Mendiburu, A., and Lozano, J. A. (2012). A review on estimation of distribution algorithms in permutation-based combinatorial optimization problems. *Progress in Artificial Intelligence*, 1(1):103–117.

[Ceberio et al., 2015] Ceberio, J., Mendiburu, A., and Lozano, J. A. (2015). The linear ordering problem revisited. *European Journal of Operational Research*, 241(3):686–696.

[Ierusalimschy, 2006] Ierusalimschy, R. (2006). *Programming in lua*. Roberto Ierusalimschy.

[Koopmans and Beckmann, 1957] Koopmans, T. C. and Beckmann, M. (1957). Assignment problems and the location of economic activities. *Econometrica: journal of the Econometric Society*, pages 53–76.

[Lattner and Adve, 2004] Lattner, C. and Adve, V. (2004). LLVM: A compilation framework for lifelong program analysis and transformation. pages 75–88, San Jose, CA, USA.

[Lavaee et al., 2019] Lavaee, R., Criswell, J., and Ding, C. (2019). Codestitcher: inter-procedural basic block layout optimization. In *Proceedings of the 28th International Conference on Compiler Construction*, pages 65–75.

[Lozano et al., 2006] Lozano, J. A., Larrañaga, P., Bengoetxea, E., and Inza, I. (2006). *Towards a new evolutionary computation: advances on estimation of distribution algorithms*, volume 192. Springer Science & Business Media.

[Malagon et al., 2020] Malagon, M., Irurozki, E., and Ceberio, J. (2020). Alternative representations for codifying solutions in permutation-based problems. In *2020 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8. IEEE.

[Matsakis and Klock II, 2014] Matsakis, N. D. and Klock II, F. S. (2014). The rust language. In *ACM SIGAda Ada Letters*, volume 34, pages 103–104. ACM.

[Panchenko et al., 2021] Panchenko, M., Auler, R., Sakka, L., and Ottoni, G. (2021). Lightning bolt: powerful, fast, and scalable binary optimization. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*, pages 119–130.

[Papadimitriou and Steiglitz, 1998] Papadimitriou, C. H. and Steiglitz, K. (1998). *Combinatorial optimization: algorithms and complexity*. Courier Corporation.

[Pelikan and Mühlenbein, 1999] Pelikan, M. and Mühlenbein, H. (1999). The bivariate marginal distribution algorithm. In *Advances in soft computing*, pages 521–535. Springer.

[Pettis and Hansen, 1990] Pettis, K. and Hansen, R. C. (1990). Profile guided code positioning. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 16–27.

[Pirlot, 1996] Pirlot, M. (1996). General local search methods. *European journal of operational research*, 92(3):493–511.

[StackOverflow, 2021] StackOverflow (2021). Stack overflow developer survey 2021.

[Triantafyllis et al., 2003] Triantafyllis, S., Vachharajani, M., Vachharajani, N., and August, D. (2003). Compiler optimization-space exploration. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.*, pages 204–215.

[Trofin et al., 2021] Trofin, M., Qian, Y., Brevdo, E., Lin, Z., Choromanski, K., and Li, D. (2021). Mlgo: a machine learning guided compiler optimizations framework.

[Van Rossum and Drake, 2009] Van Rossum, G. and Drake, F. L. (2009). *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA.

[Wang and O'Boyle, 2018] Wang, Z. and O'Boyle, M. (2018). Machine learning in compiler optimization. *Proceedings of the IEEE*, 106(11):1879–1901.