FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Strategies for Compiler Phase Ordering Targeting CPUs

João Miguel Araújo Monteiro da Rocha

Mestrado em Engenharia Informática e Computação

Supervisor: Prof. Dr. João M. P. Cardoso

July 31, 2022

Strategies for Compiler Phase Ordering Targeting CPUs

João Miguel Araújo Monteiro da Rocha

Mestrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

Chair: Prof. Dr. Pedro C. Diniz
External Examiner: Dr. Cupertino Miranda, Synopsys
Supervisor: Prof. Dr. João M. P. Cardoso

July 31, 2022

Abstract

Compilers can optimize programs by performing a sequence of transformation phases. The set of transformations, along with their order, can significantly impact the performance (e.g., execution time and energy consumption) of the optimized program.

Using the precomputed compiler phase orders usually improves performance, but it is possible to have better results by individually tailoring compiler sequences, known as phase ordering, to each specific program and target pair. However, exploring sequences of phases is a complex and time-consuming task, and an exhaustive exploration of all viable sequences is not feasible.

Selecting compiler phases alone represents a complex problem to be solved, and ordering the phases adds further complexity, making it a long-standing problem in compiler research.

We propose to develop a Design Space Exploration (DSE) strategy to recommend phase orders that lead to better performance than possible with the current approaches or similar results with less exploration time, using mainly the LLVM compiler infrastructure and target computing platforms using ARM microprocessors.

We evaluate our DSE with 30 PolyBench (v4.2.1) kernels. Results show an increase in performance concerning the number of CPU cycles of up to 1.66x and a code size reduction of up to 21% against the best LLVM (v12.0.1) standard optimizations, considering `O1`, `O2`, `O3`, and `Os` compiler options when targeting a Cortex-A53 64-bit device.

Keywords: Phase-ordering. Design Space Exploration. Compiler. Clang. LLVM. Optimization. Performance.

Resumo

Os compiladores conseguem optimizar programas, executando sequências de fases de transformações. O conjunto de transformações, juntamente com a sua ordem, pode ter um impacto significativo no desempenho (i.e., tempo de execução e consumo de energia) do programa optimizado.

Usar as ordenações de fases pré-computadas do compilador resulta, normalmente, em melhorias de desempenho, mas é possível obter melhores resultados adaptando individualmente as sequências, conhecido como ordenação de fases, a cada par—programa específico e alvo. Contudo, explorar as sequências de transformações é uma tarefa complexa e demorada, e a exploração exaustiva de todas as sequências viáveis não é praticável.

Selecionar as fases de compilação já representa um problema complexo a ser resolvido, e a ordenação das fases aumenta ainda mais a complexidade, tornando-o num problema de longa data na área da investigação dos compiladores.

Nós propomos desenvolver uma estratégia de Exploração do Espaço de Soluções (EES) para recomendar ordenações de fases de optimização que levem a um melhor desempenho do que o possível com as abordagens actuais ou que levem a resultados semelhantes com menos tempo de exploração, usando principalmente a infraestrutura do compilador LLVM e tendo como alvo as plataformas de computação que usem os microprocessadores ARM.

Nós avaliamos o nosso EES com 30 kernels do PolyBench (v4.2.1). Os resultados mostram um aumento no rendimento, no que diz respeito ao número de ciclos do CPU, até 1.66x, e uma redução do tamanho do código até 21%, face às melhores optimizações standard, fornecidas pelo LLVM (v12.0.1), considerando as opções `O1`, `O2`, `O3`, e `Os` quando tem como alvo um dispositivo Cortex-A53 64-bit.

Palavras-chave: Ordenação de Fases. Design do Espaço de Exploração. Compilador. Clang. LLVM. Optimização. Desempenho.

To my family, especially my mom, siblings, and nephews.

Acknowledgments

This work benefited from the contribution of many people, and now is the time to mention and appreciate them.

When the time to choose a thesis came, I faced the paradox of choice: too many outstanding proposals. So, I decided to put the proposals aside and talk with professor João M.P. Cardoso with whom I (rightly) believed I would enjoy working. However, Cardosos research areas were out of my comfort zone.

Today I can say it was worth taking the risk. This work tested my limits, forced me to improve and learn new skills, and sharpened my appetite for this field. For that, I feel satisfied.

My supervisor and I had more ambitions for this work, but this was what I was able to deliver given the circumstances in which I found myself. I am indebted to him for the quality and duration of his guidance.

I also owe a particular debt to Ricardo Nobre. His dissertation shaped this work, and he made himself available to borrow hardware and answer any questions about his doctoral study.

An individual thanks go to João Bispo, who helped me prepare for the oral examination.

Thank the Special-Purpose Computing Systems, languages, and tools (SPeCS) group and FEUP for making available infrastructure and resources.

On a final note, I am grateful to the committee—Pedro C. Diniz, Cupertino Miranda, and João M. P. Cardoso—who contributed with suggestions to this document and to an examination session that I will hardly forget. I appreciate the committee for encouraging the extension of my academic journey.

João

"I think that it is extraordinarily important
that we in computer science keep fun in computing."

Alan J. Perlis

Contents

List of Figures

List of Tables

Acronyms

ACM Association for Computing Machinery. 41–43

AMD Advanced Micro Devices. 7

ARC Advanced RISC Computing. 7

ARM Advanced RISC Machines. 2, 7, 42

CGO Code Generation and Optimization. 42

CoRR Computing Research Repository. 42

CPU Central Processing Unit. xiii, 5, 16, 21, 25, 26, 28, 30–36, 39, 47

CSV Comma-Separated Values. 13, 16, 21, 23, 55, 56

DB Database. 7

DSE Design Space Exploration. 2, 7

FIFO First In, First Out. 26

FPGA Field Programmable Gate Array. 7, 43

GCC GNU Compiler Collection. 6, 7, 42

GNU GNU's Not Unix. 42

GPU Graphics Processing Unit. 7

IEEE Institute of Electrical and Electronics Engineers. 42, 43

JSON JavaScript Object Notation. 13, 47, 54, 55

LLVM Low Level Virtual Machine. 2, 3, 6, 8–10, 16, 25, 39–42

LPDDR Low-Power Double Data Rate. 25

LTS Long-Term Support. 25

MIT Massachussetts Institute of Technology. 10

OS Operating System. 9, 26

PAPI Performance Application Programming Interface. 26, 49

RISC Reduced Instruction Set Computer. 7

SA Simulated Annealing. 7

SDRAM Syncrhronous Dynamic Random Access Memory. 25

SIGBED Special Interest Group on Embedded Systems. 43

SIGPLAN Special Interest Group on Programming Languages. 43

SPARC Scalable Processor Architecture. 7

SQL Structured Query Language. 23

SSH Secure Shell. 55

Chapter 1

Introduction

1.1 Motivation

Compilers initially had one objective in mind. It was to transform programs written in a more natural language into code that the machine understands. A compiler also allows one to write a program once and make it work in different computer systems. Nowadays, we have *optimizing compilers* that, besides converting a program to machine code, optimize code to be more performant. In order to optimize code, these compilers can perform transformations in code regions. A code region can be a loop or an entire function.

A typical compiler can perform hundreds of transformations, commonly known as *compiler phases*. These phases attempt to do multiple transformations, each of which preserves the program's behavior and usually improves its performance. Many of these phases also need certain conditions to be applicable (such as loops or constants). The code transformed by an optimization phase is the input of another phase. As a result, phases interact with each other by enabling or disabling the chance of another phase to perform work. These interactions can produce different optimizations, positively or negatively impacting the generated code's performance. Hence, searching for the best way to apply the phases is imperative. This challenge is known as the *phase ordering problem* [1–3].

In the last five decades, compiler researchers have tried to solve the *phase ordering problem* [3–8]. It is challenging because an optimization sequence does not have the same performance results for every program or target architecture. Finding the best sequence of phases is essential for every industry and individual concerned with performance.

Environments with strict non-functional requirements such as execution time or energy consumption take particular advantage of performance gains. These environments are the case with embedded systems and in high-performance domains.

1.2 Goals and Contributions

A formal way to describe this problem is as follows. Given a program P, a set of compilation phases S, and a set of directed graphs G that contain knowledge about S, we want to find a sequence s of these phases so that the optimized program P^* is optimal. We define optimal as a program that performs better than when compiled with the best default optimizations.

We examine the performance of a set of benchmarks in three ways. We record the performance for (1) the best default compiler optimization sequence, (2) a state-of-the-art Design Space Exploration (DSE) graph-based approach [9], and (3) our approach. We propose to develop an algorithm to search for sequences of compiler optimizations using the LLVM compiler infrastructure [10] and targeting ARM devices [11].

Finding the optimal sequence of optimizations to achieve the best performance (such as execution time or energy consumption) for a given program and architecture is a problem remaining unsolved. Compilers try to optimize for the generality of programs and supported architectures, but specializing in this sequence of optimizations often results in higher performance, which leaves space for improvement. With this study, we want to answer the following research questions:

Q1 What impact do a program's features have in computing a heuristic to guide the search of the solution space?

Q2 What is the efficiency of limiting the maximum number of phases in a sequence?

Q3 How does limiting the search space size affects the results achieved?

Q4 What is the impact of using combinations of graphs representing knowledge about compilation phases in searching the solution space?

Q5 What is the efficiency of limiting the number of different compilation phases that can appear in a sequence?

We present strategies to address the phase ordering problem. Solving this problem requires finding optimization sequences and evaluating them. To find and evaluate these strategies, we open-sourced three software packages[1]. The module `dervin` is responsible for implementing the strategies, the module `ghopper` measures the impact of the given sequences, and the last module `amidala` allows us to compute statistics and visualize them.

1.3 Document Outline

The organization of the remainder of this thesis is as follows. In Chapter 2, we provide background material and describe related work. Chapter 3 presents our approach, and we show the experimental procedures and discuss the results obtained in Chapter 4. Chapter 5 concludes this document with our contributions and future work. Appendix A describes the adaptations made

[1] https://github.com/jmrocha/ms-thesis

to the benchmark suite evaluated. Appendix B sets out how we optimize a program with LLVM. Appendix C describes how to install the software packages we developed, and examples of their usage.

Chapter 2

Background and Related Work

This chapter introduces the phase ordering and best selection problems, and presents state-of-the-art approaches.

2.1 Phase Ordering Problem

A compiler can perform several transformations to the code, and we can call each transformation a *phase*. Consider the for-loop in Listing 2.1. The compiler can apply the *loop unrolling* [12] phase to remove the loop and transform it into five instructions (see Listing 2.2).

Listing 2.1: C code excerpt with a for-loop.

```
for (int = 0; i < 5; i++) {
  a[i] = b[i] + c[i];
}
```

Listing 2.2: The code of Listing 2.1 after the *loop unrolling* phase.

```
a[0] = b[0] + c[0];
a[1] = b[1] + c[1];
a[2] = b[2] + c[2];
a[3] = b[3] + c[3];
a[4] = b[4] + c[4];
```

Removing the loop reduces the number of instructions to be executed. There is no need for (1) check at each iteration if the loop has already terminated, (2) instructions to store and increment the index value, and (3) instructions associated with the control flow for repeating the loop body.

A phase can degrade or improve the following phase, and the same pair of phases can have different effects if their order is changed. The set of phases chosen also have side effects. We want to find the best order of phases to apply to a given program to achieve its maximum performance potential, be it the number of CPU cycles, code size, or energy consumption.

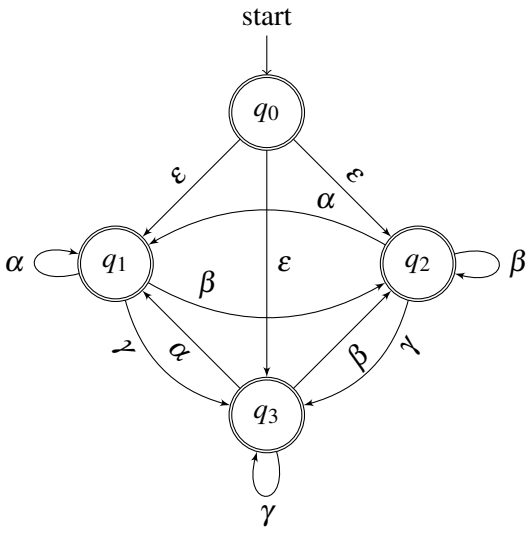


Figure 2.1: Representation of all the possible phase orders in a nondeterministic automaton (NFA) when considering phases α, β, and γ.

2.1.1 The Dimension of the Search Space

Considering that a compiler can perform three optimization phases α, β, and γ, the solution space can be represented by a directed graph, assuming that all phases are compatible. In this case, the search space is unbounded, and each sub-path is a valid sequence, as shown in Figure 2.1. It is clear that we cannot use a brute force approach and that we need to place constraints if we want to find a solution.

Let S be the search space, P the set of phases supported by the compiler, and N the length of the sequence we want to search. The cardinality of S, denoted by $|S|$, is given by the number of possible phase orders. Equation 2.1 shows how to calculate $|S|$ [9].

$$|S| = \sum_{k=0}^{N} P^k \tag{2.1}$$

To simplify this problem, compiler researchers partitioned the problem into two categories: *phase selection* [13–16] and *phase ordering* [17–19]. The phase selection problem is only concerned with reducing the set of optimizations considered, while the phase ordering is responsible for finding the best sequence. Considering that compilers such as the LLVM 12.0.1[1] are capable of more than 78 transformation phases, even dividing the problem in two, the search space is enormous for each problem [20].

2.1.2 Specialized Compiler Sequences

Optimizing compilers can perform individual program transformations, but they ship precomputed sequences of optimizations. In the case of GCC [21] and LLVM [10], these sequences of phases

[1] https://releases.llvm.org/12.0.1/docs

Table 2.1: Overview of phase ordering approaches.

(1)	(2)	(3)	(4)[a]	(5)	(6)	(7)	(8)
[18]	GCC	C	NI	PO	x86, AMD, ARC	ET, S, CT	MiBench, Berkeley DB
[9, 22]	Clang, CoSy	C	I	PO	MicroBlaze (RISC), LEON3 (SPARCv8), ARM, Mor1kx (RISC)	ET, S, CT, EC	Texas Instruments, REFLECT, PolyBench
[19, 23]	Clang	C	H	PO, PS	x86	ET	CBench

Note: Column headings are as follows: (1) Reference; (2) Compiler; (3) Programming Language; (4) Kind of Approach; (5) Problem Class; (6) Target; (7) Metric; and (8) Benchmark Suite.

[a]Execution time (ET), energy consumption (EC), compilation time (CT), and size (S).
[b]Phase ordering (PO), and phase selection (PS).
[c]Iterative (I), non-iterative (NI), and hybrid (H).

can be applied to any program by enabling the flags `-O{1, 2, 3, s}`, where `-O1` is the least aggressive optimization of the three, and `-O3` the most aggressive, while `-Os` focuses in reducing the code size. Even though the flag `-O3` is the most aggressive optimization, it does not mean it achieves the best performance, as we show in Section 4.2.2.

These pre-computed sequences of optimizations, built by the compiler writers that have deep knowledge about these phases, do not guarantee to be representative of the functions one wants to optimize. Also, they usually optimize for execution time or code size, while there are use cases where energy consumption is the priority [22].

2.1.3 Related Work

The phase selection and phase ordering problems have been a longstanding problem. Compiler researchers have mitigated these problems with iterative and non-iterative approaches, machine learning, and hybrid approaches. Table 2.1 lists an overview of relevant approaches, which we briefly describe in the following text. Nobre presents a Design Space Exploration (DSE) approach to evaluate different schemes for exploring phase ordering on different architectures, including GPUs, and FPGAs, and on different compilers [9]. The DSE system relies on a graph representing favorable optimizing phases. Then it proceeds to use a Simulated Annealing (SA + Graph) algorithm and a search method based on probabilistic distribution and a random factor (IterGraph).

Nobre et al. [9, 24] uses iterative compilation and a graph-based predictive model to find new optimization sequences. This model learns the sequences used in training programs and then explores the most promising ones, starting the sequence at the most connected node. To avoid not taking other paths that could give even more significant results, this model also uses a random factor in its heuristic. They concluded that (1) the SA + Graph can achieve results with comparable performance but with less exploration time, and (2) the IterGraph and the SA + Graph can achieve higher geometric mean speedups than with the approaches in [25, 26].

Nobre et al. [22] evaluate the impact of phase ordering to reduce the energy consumed by a set of programs compared with the LLVM standard optimizations. They show that optimizing performance to improve energy consumption is not always successful.

Ashouri et al. [19] introduce a machine learning approach to mitigate the phase ordering problem by predicting the speedup of a complete sequence of optimizations. They go further and improve their machine learning model through Recommender System techniques.

Martins et al. [27,28] propose clustering methods to reduce the dimension of the search space. Based on empirical data, they identify clusters of phase sequences that result best for specific features Given a new program, they choose the cluster that best matches its features. They propose a Genetic Algorithm as one of the techniques to search in the set of phases found in that matched cluster.

Fursin et al. [18] present the Milepost framework to make machine learning-based multi-objective optimization a realistic, automatic, reproducible, and portable technology for general-purpose production compilers. This framework starts with the observation that similar programs may exhibit similar behavior and require similar optimizations, so it is possible to correlate program features and optimizations, thereby predicting good transformations for unseen programs based on previous optimization experience.

2.2 Best-Selection Problem

A different phase ordering problem dedicates to choosing the best set of phases to apply for a given program [13–16]. This problem is not interested in what the order is and it is similar to the problem of selecting compiler flags (a problem known as flag selection [29]). The best selection problem can be a first step of the phase ordering problem and in that case can reduce the number of phases to be considered. Given a program with loops, one is interested in phases related to loops, and if given a program with constants, one may want to include phases that handle transformations related to constants. Nevertheless, these phases can be more effective in the presence of other phases because of the interaction between them.

2.3 Summary

This chapter presented two main problems in the compiler research field, the phase ordering and best-selection problems. The phase ordering problem is responsible for selecting the best sequence for a given program, while the best selection problem only focuses on choosing the best set of phases or compiler flags.

Chapter 3

Our Approach

In order to explore the performance potential in current state-of-the-art approaches, we consider some strategies. One of these strategies is a state-of-the-art approach developed by Nobre [9]. One of the author's recommendations is, for example, changing the first node policy. Changing where we start our search can profoundly impact the sequences found. Another aspect subject to change is the *heuristic* because it can improve as more information is used to guide the search. An example of such information is the features of a program. A program feature can be a static one, such as the number of loops, or a dynamic one, such as the number of floating-point operations.

Sequences that perform well on known programs could also perform well in unseen programs that share some of the features. Because we are not sure what the optimal sequence looks like, we can start from a random place, and then search for phases that contribute to the functions that are most similar to the target program. Phases also have dependencies, meaning wrong combinations could invalidate the whole sequence. Using this information about dependencies could also be used to improve the heuristic.

Having an engine that implements these strategies is not enough. We need a way to test how well these strategies perform. We need a baseline to compare how well a strategy performs. A good starting point is comparing a sequence with the best predefined sequence in LLVM[1]. If we can improve the performance of a program better than the best of the compiler-provided standard optimizations, we are up to a good start, which means optimizing the program with our sequence, executing the program, and collecting metrics about its execution. Then we do the same for each standard optimization and compare their metrics.

There is a lot that can influence our observations. Starting with the sequence search, since it is not a deterministic method, we can find a great sequence and observe a substantial speedup, but we are also subject to no speedup. Maybe a sequence only works well for a particular program we are testing, and it is not valuable for most of the programs we encounter or even the particular domain in which we work. The operating system itself could influence our observations; maybe the OS was busy when we evaluated the standard optimizations, and our comparisons were not fair. Our calculations and statistics are error-prone or may be inappropriate, and we end up taking

[1] https://releases.llvm.org/12.0.1/docs

the wrong conclusions. How we benchmark a program, which is how we measure the performance of a program, can also introduce errors. Maybe our instruments introduce too much overhead, and that overhead surpasses any improvement at all. Maybe we are not seeing too many speedups and think a particular approach does not give good results, and it is all the fault of our measuring method. So, this is to say that we need to be careful in many steps of our pipeline.

We just described what it looks like to compare, for one program, two sequences. Nevertheless, we want to compare a considerable amount of sequences over a set of programs. We need to have the flexibility of collecting data using different strategies with different parameters, and then analyzing that data, looking for patterns, and seeing what improvements we have. If those improvements are consistent, ask what the sources for those improvements are, and then tweak the strategies. So, this involves a pipeline that goes all the way from being able to automatically optimize any program with a given sequence, collecting metrics, storing the results, analyzing those results, and visualizing those same results.

We adopt three search strategies to explore the performance potential in optimization sequences using the PolyBench suite.[2] We built a framework that collects data on these strategies' performance for a given benchmark suite. We developed a software package to analyze and visualize this data. All software developed is open-source with an MIT license.

3.1 Strategies

This section describes the strategies we decided to implement to find optimization sequences. We will compare later the sequences found by this strategy with other state-of-the-art approaches, including the compiler-provided standard sequences. We open-sourced an engine that implements these strategies, which is described later in this chapter.

The first section describes a random strategy. This strategy shows what kind of results we can get with a simple approach and creates a much bigger and different search space than the predefined sequences from the LLVM compiler that we can explore. We can change the maximum sequence length and allow phase repetitions. The second section describes one of the approaches developed by Nobre, where the author uses a graph-based iterative compilation [9]. Nobre explores the compiler's predefined sequences starting at the most connected phase but prioritizes sub-sequences appearing more often and using a random factor to explore more of the search space. We also describe several changes to the root node policy we tried.

3.1.1 Strategy $\mathscr{S}_{\mathrm{S}}$ and $\mathscr{S}_0$

The standard strategy $\mathscr{S}_{\mathrm{S}}$ applies the predefined sequences present in the standard optimization levels `O0`, `O1`, `O2`, `O3`, and `Os`. Strategy $\mathscr{S}_0$ builds an optimization sequence by choosing random phases from a given set.

[2] https://github.com/MatthiasJReisinger/PolyBenchC-4.2.1

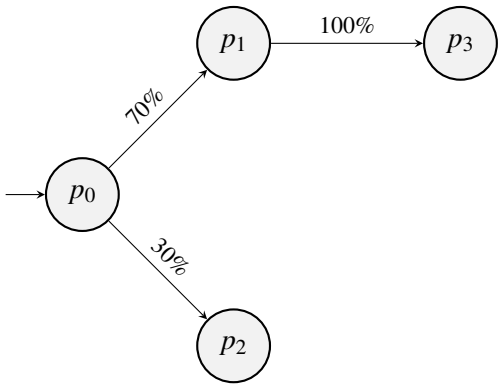


Figure 3.1: Example of the initial graph generated from a set of phase orders.

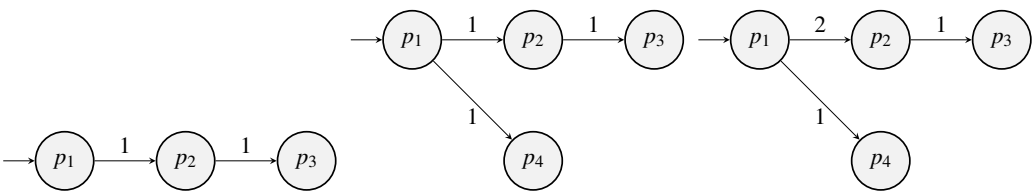
3.1.2 Strategy $\mathscr{S}_1$

This is a state-of-the-art strategy developed by Nobre [9], and we describe possible variations at the end of this section. This model finds a new phase order by choosing random phases from a given set of phase orders, but phases with more occurrences have higher chances of being selected. We derive variations of this strategy by choosing a different starting phase.

Given a set of phase orders, we start by representing them in a weighted directed graph as in Figure 3.1. Nodes represent the phases, and edges preserve their order as observed. The edge's weight determines the chances of being followed. Then, we can build the phase order—choose a node to start, follow a path based on random values and the probability distribution, and append the corresponding phases to the phase sequence. Next, we describe how to construct the graph, the probability distribution, and how to find a phase order.

Let $\Phi = \{\phi_1, \phi_2, \phi_3\}$ be the set of phase orders from which we will find a new phase order. Let each element of this set be $\phi_1 = [p_1, p_2, p_3]$, $\phi_2 = [p_1, p_4]$, $\phi_3 = [p_1, p_2]$. Where p_1, p_2, p_3, and p_4 represent optimization phases. For each unique phase, we create a node and register each pair of phases' order and the number of occurrences. Figure 3.2 illustrates the graph updates for each element of Φ.

The next step is to normalize the weights of each node on the total weight of their outgoing edges, as shown in Figure 3.3. Consider the phase p_1 from Figure 3.2c. The total weight from the outgoing edges of p_1 is 3. Therefore, the normalized weights of its outgoing edges are $\frac{2}{3}$ and $\frac{1}{3}$, respectively.



(a) After observing ϕ_1. (b) After observing ϕ_2. (c) After observing ϕ_3.

Figure 3.2: Example of the initial step to generate the graph for the set Φ.

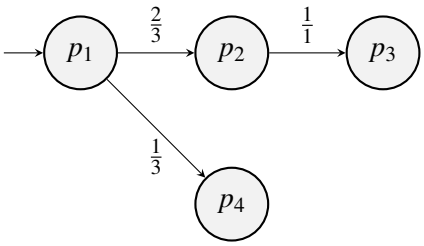


Figure 3.3: Example of the graph representing the set Φ, with weights normalized.

Consider the example shown in Figure 3.4. Starting at p_4, p_5 has 10% chance of being selected, p_6 has 20%, and p_7 has 70%. This distribution is represented visually in Figure 3.4 and it is represented by the following intervals: $p_4 \rightarrow p_5 \in [0, 0.1[$, $p_4 \rightarrow p_6 \in [0.1, 0.3[$ and $p_4 \rightarrow p_7 \in [0.3, 1]$. Where $p_i \rightarrow p_j$ is the probability of moving from the phase p_i to the phase p_j. We obtain these intervals by sort ascending the weights of the outgoing edges. Since p_5 has the lowest chances, it bounds to the lower interval.

Consider the previous example in Figure 3.4. Assume we want to find a phase order with length three. Using the canon variation of this strategy $\mathscr{S}_{1a}$, see Section 3.1.2, we start the phase sequence with p_3. Let $X_1 = 0.1$ and $X_2 = 0.05$ be the random values generated. Based on these values, we append p_4 and p_5 to the phase sequence. Since the phase order has the desired length, we do not proceed further. The resulting phase order is then $\phi_r = [p_3, p_4, p_5]$. If the target length were four, we would also return ϕ_r because p_5 has no outgoing edges. Figure 3.5 illustrates this step.

Table 3.1 describes variations of the strategy $\mathscr{S}_{1a}$ by changing the phase that starts the search. We use the term *root phase* to describe a phase that starts an optimization sequence.

3.2 Software Packages Developed

We developed three packages—`dervin`, `ghopper`, and `amidala`—that form the infrastructure needed to evaluate the strategies we propose (Figure 3.6). Given a search space, often a graph

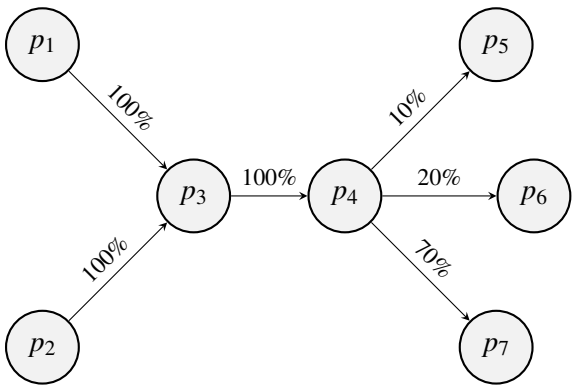


Figure 3.4: Example of a probability distribution for a set of phase orders.

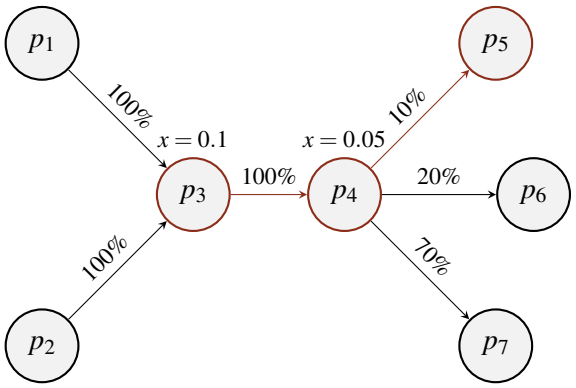


Figure 3.5: A possible phase order when searching the space represented in Figure 3.4.

described through a JSON file, the `dervin` package finds phase orders and outputs them in the JSON format. These phase orders will be used by the `ghopper` module to optimize a set of benchmarks and the results are persisted in a CSV file. The `amidala` module processes the results and is capable of showing plots, a numerical summary, and presents a shell to interact with the data.

3.2.1 Design Space Exploration Engine `dervin`

Given a search space, this module finds phase orders using one of the available strategies (Figure 3.7). For the strategy $\mathscr{S}_0$, the search space is a set of unique phases, while for the strategy $\mathscr{S}_1$, the search space is a graph. These different search spaces can be generated from previous phase orders that are stored in a text file (Figure 3.8). The new phase orders can then be evaluated on a benchmark suite by the `ghopper` package.

To implement the strategies described in Section 3.1, we implemented an engine that receives the input: strategy, maximum sequence length, base graph, or phases file, and outputs a single sequence. This engine implements a Python library, is open-source, and allows for the extension of other strategies, as we will show. There are two main algorithms: (1) to compute the probability distribution of phases to be selected into a phase order and (2) to find a phase order. Given a graph G and a set of phase orders P the COMPUTE-PROBABILITY-DISTRIBUTION procedure (Algorithm

Table 3.1: Variations of the strategy $\mathscr{S}_1$ by changing the *root node* policy.

Strategy	Description
$\mathscr{S}_{1a}$ (Canon)	We choose the phase with the most occurrences but exclude root phases.
$\mathscr{S}_{1b}$	We choose a random phase.
$\mathscr{S}_{1c}$	We choose a random phase from the set of root phases.
$\mathscr{S}_{1d}$	We choose a random phase from the root phases, but the number of occurrences increases the chances.
$\mathscr{S}_{1e}$	We choose the phase with the most occurrences.

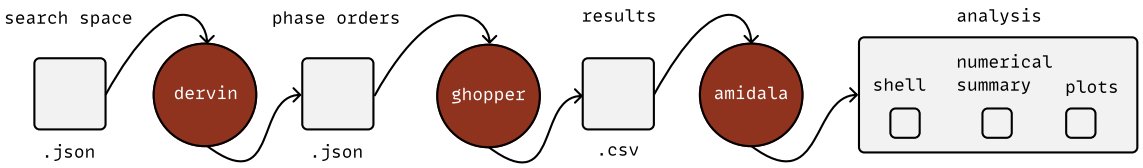


Figure 3.6: Overview of the infrastructure developed.

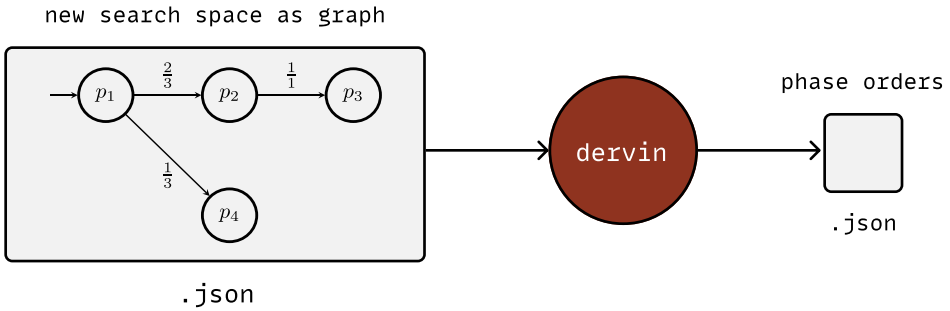


Figure 3.7: Overview of the input/output of the dervin package—I.

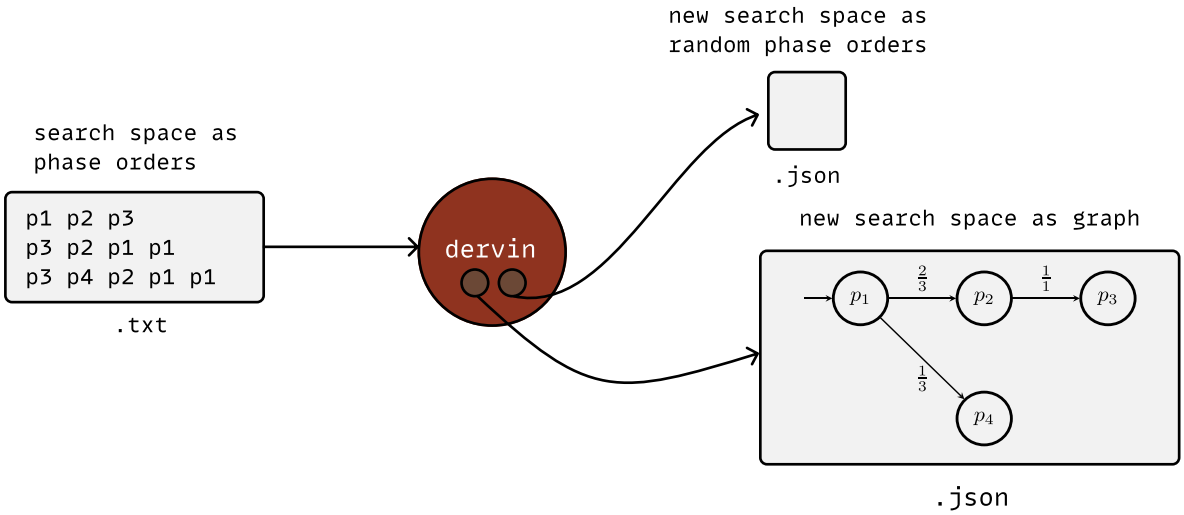


Figure 3.8: Overview of the input/output of the dervin package—II.

1) returns a graph G in which the edges weights represent the probability of that edge to be chosen as illustrated in Figure 3.3). The procedure works as follows. We record the number of occurences of each pair of phases as illustrated in Figure 3.2 and described in lines 28. Then we iterate over each edge to set the definitive weight (lines 910). The definitive weight is the edge's weight divided by the outgoing degree of the source vertex. The procedure SEARCH-PHASE-ORDER (Algorithm 2) builds a phase order. Given a weighted graph G (that represents the search space) such as the one returned by COMPUTE-PROBABILITY-DISTRIBUTION and a strategy S, the procedure SEARCH-PHASE-ORDER returns a phase order π. The phase order starts with a phase returned by the procedure GET-ROOT-PHASE. While there is a phase to be added (HAS-NEXT-PHASE), given the last phase in the optimization sequence, a graph and strategy, NEXT-PHASE returns the next phase to be added to the sequence π. Every strategy to be applied by this engine needs to implement these procedures.

Algorithm 1 Algorithm to compute the probability distribution on the graph's edges.

```
1:  COMPUTE-PROBABILITY-DISTRIBUTION(G, P)
2:      for each phase order p ∈ P do
3:          for i = 0 to p.length − 1 do
4:              e = (p_i, p_{i+1})
5:              if HAS-EDGE(G, e) then
6:                  INCREMENT-WEIGHT(G, e)
7:              else
8:                  INSERT(G, e)
9:      for each edge e ∈ G.E do
10:         e.weight = e.weight / e[0].outdegree
11:     return G
```

Algorithm 2 Base algorithm to find a phase order.

```
1:  SEARCH-PHASE-ORDER(G, S)
2:      p = GET-ROOT-PHASE(G, S)
3:      π[0] = p
4:      while HAS-NEXT-PHASE(G, S, π, p) do
5:          p = NEXT-PHASE(G, S, p)
6:          insert p into π
7:      return π
8:  HAS-NEXT-PHASE(G, S, π, v_i)
9:      return π.length < M and v_i.outdegree > 0        ▷ M is the max. phase order length
```

Appendix C.1.1 has instructions on how to install this package and shows usage examples.

3.2.2 Experiment Framework `ghopper`

To understand how good a strategy is in searching for optimization sequences, we find and evaluate multiple optimization sequences over a set of programs and compare them against a baseline. This

Figure 3.9: Overview of the input/output of the ghopper package.

process involves finding optimization sequences, optimizing each program, and collecting metrics about their execution. Figure 3.9 depicts an overview of the input/output and Figures 3.10, 3.11, and 3.12 describe the key components of the software architecture.

The metrics we collect are the performance counters (see Table 4.1) and the binary size. We use the Performance Application Programming Interface (PAPI)[3] library to collect the CPU cycles. The metrics, besides other data, use the CSV format. The collection of programs we chose belongs to the PolyBench benchmark suite, and each program needs to be a non-optimized bitcode. We describe how we optimize a program in Appendix B and how we adapt the PolyBench benchmark suite so that this framework in Appendix A can evaluate it.

We just described how we implemented the engine to search for optimization sequences. Now, we describe how we use that engine to evaluate the performance of each strategy. We test strategies by requesting sequences, evaluating them, collecting data about their performance, and analyzing them. In this section, we show how we evaluate phase orders, and collect data. Before we move on to more complex tasks, we start with a simple one: we want to get an optimization sequence from a specific strategy, optimize all programs from a given set with that sequence, and then store the results in a CSV file.

Optimizing a program in the context of the LLVM toolchain means using the optimizer to optimize the code at the intermediate representation (target-independent) level, then performing

[3] https://icl.utk.edu/papi

target-dependent optimizations giving as output an object file, and converting that object file to binary (see Appendix B.5).

The benchmarks can have dependencies or need linking with other sources. They can be in a complex hierarchical file structure, with benchmarks inside different levels of folders. So, we do some adaptations a priori to make it easier on the framework to find the available benchmarks, and to compile them. These adaptations are (1) flattening the benchmarks folder, (2) emitting non-optimized standalone bitcodes, and (3) changing the output so that our framework can easily parse the metrics. The framework knows what libraries to link, and the header files' location through external input. Appendix C.2.2 shows the process of emtiting a standalone bitcode and executing it.

This work focused on evaluating benchmarks remotely in a computer board. The connection can easily break and abort the current experiment. Appendix C.2.3 describes how we avoid breaking the remote connection.

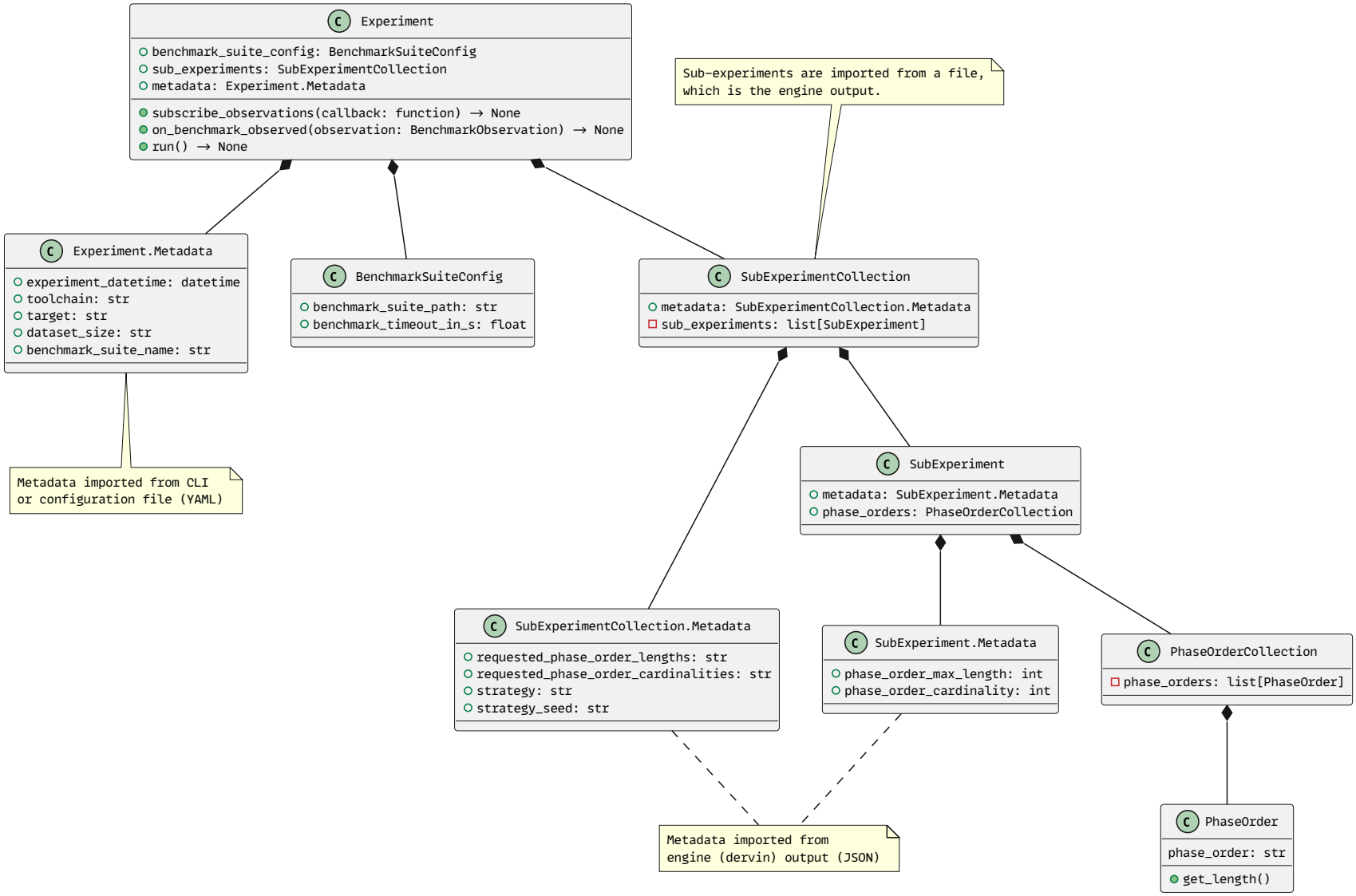


Figure 3.10: Class diagram depicting the experiment entity in the developed software package `ghopper 1.0.0`.

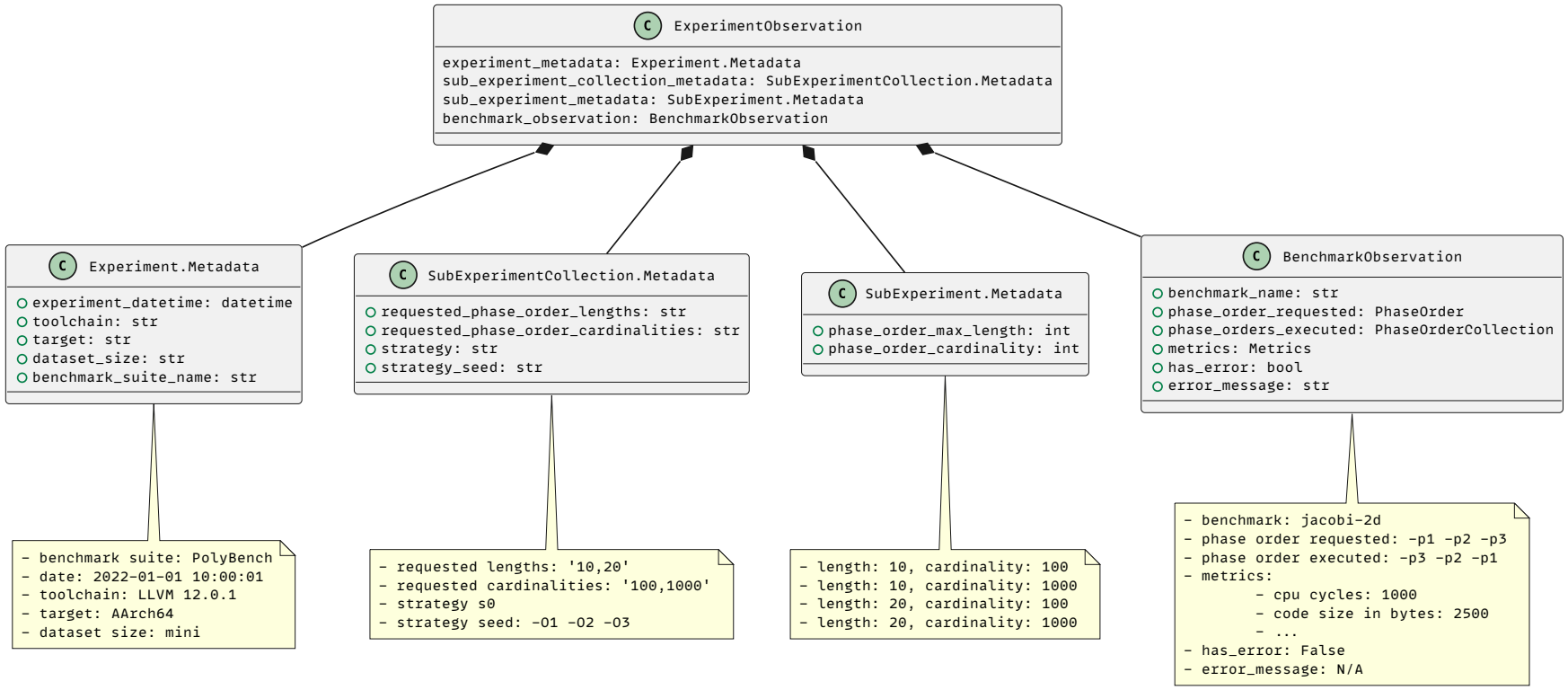


Figure 3.11: Class diagram depicting the experiment observation entity in the developed software package `ghopper 1.0.0`.

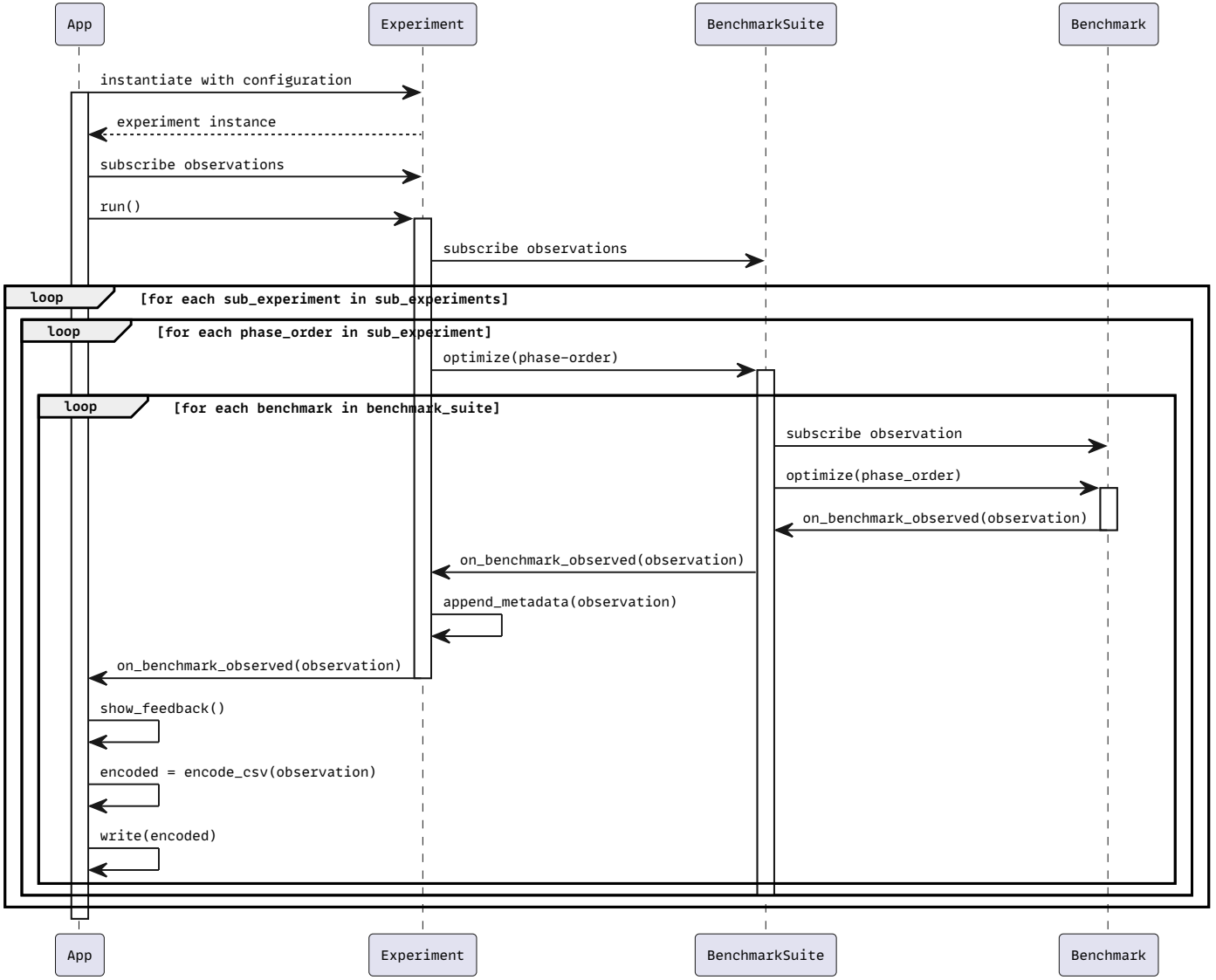


Figure 3.12: Sequence diagram depicting the core internal interactions of the developed software package `ghopper 1.0.0`.

3.2.3 Statistics and Visualization `amidala`

We developed a software package named `amidala` that automates the process of computing and visualizing statistics (Figure 3.13). This software depends on two Python libraries: (1) the `pandas` package to perform statistics and (2) `matplotlib` to plot them.

Here we are at the last stage of the pipeline, where we have all the collected data we need, and now we are going to process it, make comparisons between strategies, and visualize some statistics.

The first step is to compute the median of each repeated benchmark execution benchmark. We repeat ten times each benchmark execution, so we consider the median of those ten executions as the performance value for that benchmark.

The second step is to add a virtual benchmark for each experiment that represents the overall performance of the set of benchmarks. We compute the virtual benchmark using the *geometric mean*.

The third step is to compare one experiment A with another experiment B and add the results in a new column whose values are given by Equation 3.1.

$$\text{Metrics comparison} = \frac{\text{Metrics of } B}{\text{Metrics of } A} \tag{3.1}$$

This metrics comparison column is composed of multiple columns, one for each metric, and these metrics range from the number of CPU cycles to the number of L1 cache misses.

Each strategy can create thousands of records, and we need to process this data to know how well an approach performed against a baseline and tune its heuristics. To compare each strategy, we compute statistics over the collected metrics stored in a CSV file, compare them against a baseline and visualize them.

One metric that often changes per execution is the number of CPU cycles. These changes happen because when we measure the cycles used by a program, we are actually including other tasks the processor may be doing. To minimize that change, we repeat our experiments several times, and consider the median of those executions. We also add a virtual program to the collected data representing the set of tested programs to know how well the strategy performs overall. The virtual program has the same metrics, but each metric represents the average performance.

After processing the collected data, we query the results to answer questions such as:

- The highest speedup achieved for each benchmark.
- The number of speedups more significant than 10%.
- The sequences that accomplish speedups more significant than 10%.
- The parameters (e.g., maximum sequence length allowed, cardinality) that achieve a given speedup.
- The number of invalid optimization sequences we get.

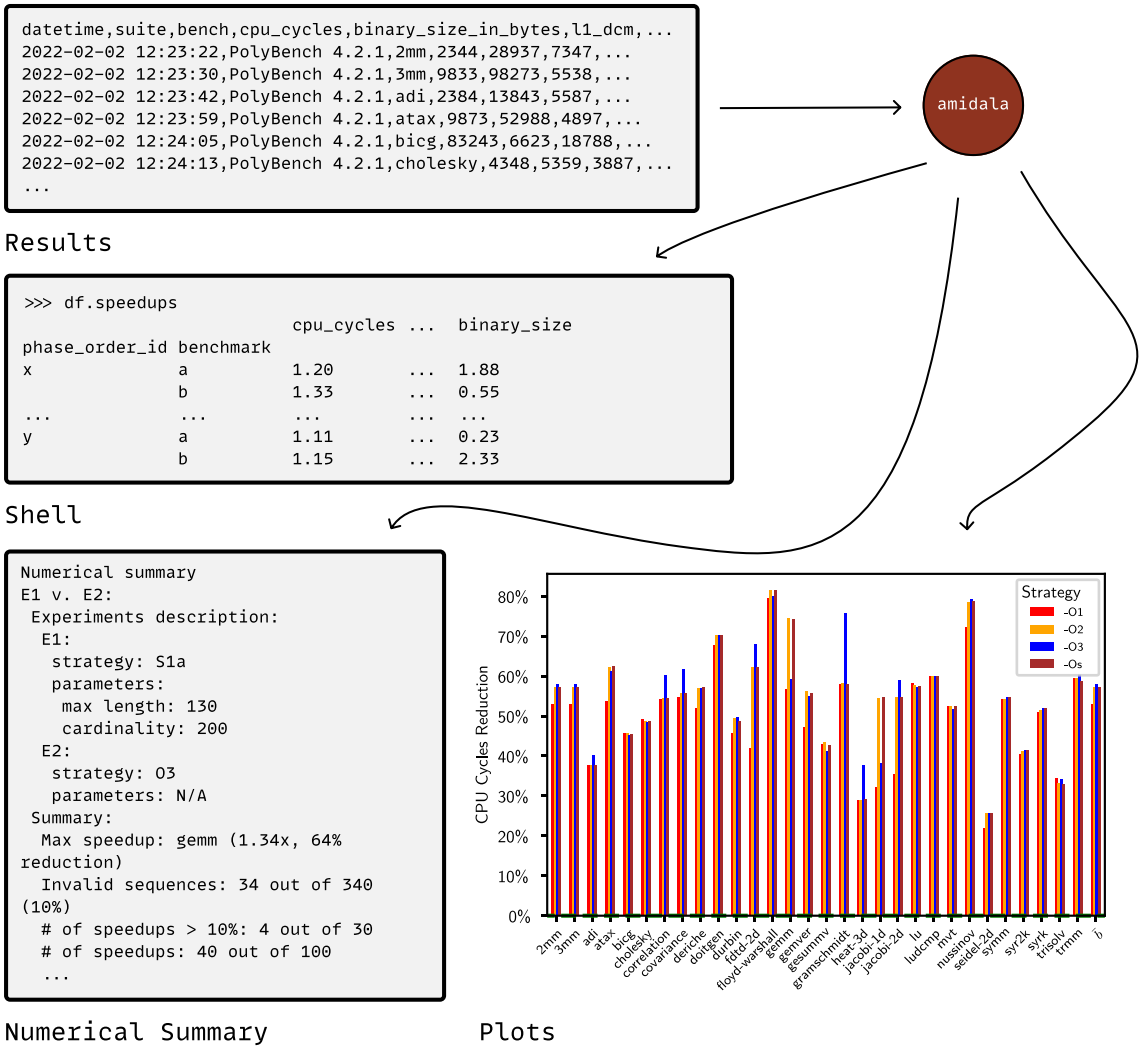


Figure 3.13: Overview of the input/output of the amidala package.

- The percentage of the search space that we explore.

Since there are many queries to data, we need to extract knowledge from the collected results, it is helpful to have some tool like SQL where we have the flexibility to formulate questions. There are tools known to be used by data analysts developed in languages such as R, Matlab, or Python. We chose to use the Python library `pandas`[4], where there is the concept of a *data frame*, a flexible tabular data structure where we manipulate it and ask questions. Appendix C.3.2 presents an example.

3.3 Summary

We presented four strategies to mitigate the phase ordering problem and developed the software to evaluate them.

The first strategy applies the standard optimizations `O0`, `O1`, `O2`, `O3`, and `Os`; the second chooses random phases from a set; a third strategy is a state-of-the-art approach that serves as a reference for our work, where we also derive variations of our own. The last one considers the program's features to improve the search heuristic.

We open-sourced the three modules necessary to conduct this study. The first package is the engine `dervin` that implements these strategies. The second package `ghopper` is the framework that takes the sequences found by the engine, measures them, and outputs the results in the CSV format. The last one, `amidala`, compares the experiments, and allows us to visualize statistics about them.

[4] `https://pandas.pydata.org`

Chapter 4

Experimental Results and Discussion

In Chapter 3, we proposed strategies to mitigate the phase ordering problem and presented the software we developed to evaluate the strategies. This chapter describes the results obtained from the experiments and analysis of those results.

4.1 Experimental Procedure

4.1.1 Hardware and Software Requirements

Our experiments run on a Raspberry Pi 3 Model B Plus Rev 1.3[1] board, which includes a quad-core processor Cortex-A53 64-bit—that has an Armv8-A architecture with the instruction set AArch64, and it operates at a maximum frequency of 1.4 GHz—and 1 GB LPDDR2 SDRAM of memory.

The board runs the Ubuntu Server 22.04 LTS with a Linux kernel 5.15.0-1008-raspi. We use the Python runtime environment 3.10[2], the tools from LLVM 12.0.1[3], the Performance Application Programming Interface (PAPI) 6.0.0.1[4], and the software packages we developed[5], `dervin` 1.0.0, `ghopper` 1.0.0 and `amidala` 1.0.0.

We evaluate the performance using the benchmarks from PolyBench 4.2.1[6], which consists of 30 kernels.

4.1.2 Procedures

The execution time of each benchmark is the median execution time of ten runs. We define the kernel scaling governor[7] to `performance` which sets the CPU statically to the highest frequency[8] (i.e., 1.4 GHz), flush 33 MB of cache before each start, and set the scheduler policy to First In,

[1] https://www.raspberrypi.com/products/raspberry-pi-3-model-b-plus
[2] https://docs.python.org/3.10
[3] https://releases.llvm.org/12.0.1/docs
[4] https://bitbucket.org/icl/papi/src/papi-6-0-0-1-t
[5] https://github.com/jmrocha/ms-thesis
[6] https://github.com/MatthiasJReisinger/PolyBenchC-4.2.1
[7] https://community.arm.com/oss-platforms/w/docs/528/cpufreq-dvfs
[8] https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt

Table 4.1: Table summarizing the performance counters available in the Raspberry Pi 3 Model B Plus Rev 1.3.

Performance Counter	Description
PAPI_L1_DCM	Level 1 data cache misses
PAPI_L2_DCM	Level 2 data cache misses
PAPI_TLB_DM	Data translation lookaside buffer misses
PAPI_TLB_IM	Instruction translation lookaside buffer misses
PAPI_BR_MSP	Conditional branch instructions mispredicted
PAPI_L1_DCA	Level 1 data cache accesses
PAPI_L2_DCA	Level 2 data cache accesses
PAPI_TOT_INS	Total instructions completed
PAPI_LD_INS	*Load* instructions completed
PAPI_SR_INS	*Store* instructions completed
PAPI_BR_INS	*Branch* instructions completed
PAPI_HW_INT	Hardware interrupts

Source: The available performance counters were retrieved with PAPI by executing, in a shell, the following command: `papi_avail | grep Yes`.

First Out (FIFO) mode to minimize the OS interference.
The PolyBench has macros to flush the cache (POLYBENCH_CACHE_SIZE_KB), set the scheduler policy (POLYBENCH_LINUX_FIFO_SCHEDULER) and collect metrics with PAPI (POLYBENCH_PAPI).

Every processor can have different performance counters, and the PolyBench authors provide a configuration file to define what counters we want enabled. Table 4.1 describes the performance counters available in our hardware.

The performance counters, available through hardware registers, give metrics such as the number of CPU cycles. We reset and start each counter before executing the program and stop the counter after the program exits. Although using the CPU cycles counter is more accurate than measuring the time—this is even more important because these benchmarks can take milliseconds to complete if given small inputs—we will have fluctuations in our results. This fluctuation is because the OS will take on other tasks besides the program we are evaluating, and we will be counting those cycles too.

To increase the accuracy of our results, we evaluate the same benchmark several times and take the median value. Besides that, we also change the Linux OS scheduler policy to FIFO to reduce the number of tasks the OS performs when running a benchmark, and clean the cache between executions.

Each benchmark operates on a given dataset which can vary in size, and there are five sizes available: mini (MINI_DATASET), small (SMALL_DATASET), medium (MEDIUM_DATASET), large (LARGE_DATASET), and extra large (EXTRA_LARGE_DATASET).

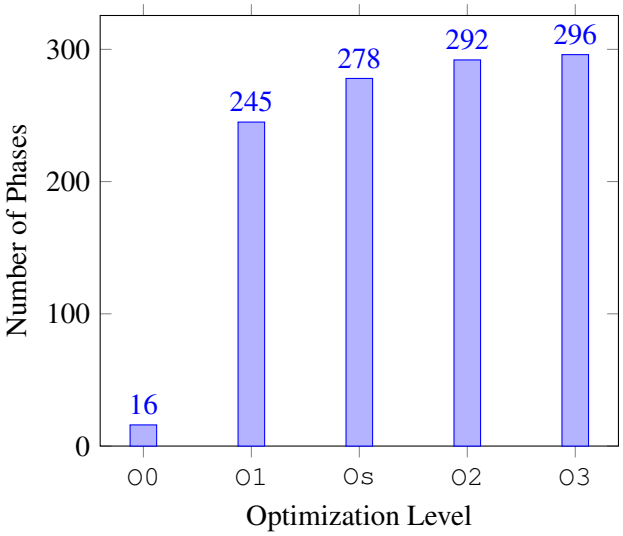


Figure 4.1: Phase order length per standard optimization.

4.1.3 Experiments Conducted

To answer the questions in this study, we conducted several experiments and summarized them in Table 4.2. We consider S to be the set of the standard sequences present in `O0`, `O1`, `O2`, `O3`, and `Os`, and R the set of random sequences with length $L \in [5,296]$ and cardinality 10^5. Figure 4.1 describes how many phases each standard optimization has.

Each experiment has an *id* which we reference when showing the results in the next section. We add a virtual benchmark $\bar{b}$ that represents the geometric mean performance. The performance results use the standard optimizations as baseline on the PolyBench Suite in Section 4.2.1, and these optimizations comprise the set `O0`, `O1`, `O2`, `O3`, and `Os`.

In Section 4.2.3, we test the strategy $\mathscr{S}_{1a}$ to predict new sequences from the standard ones against the best `-O` to see if we could find better permutations.

Table 4.2: Table summarizing the experiments conducted.

ID	Strategy	Maximum Sequence Length	Cardinality	Sequences	Dataset Size
$\mathscr{E}_1$	$\mathscr{S}_{\mathrm{S}}$	—	—	—	Mini
$\mathscr{E}_2$	$\mathscr{S}_0$	Random $\in [5,296]$	10^3	S[a]	Mini
$\mathscr{E}_3$	$\mathscr{S}_{1a}$	292	300	S	Mini
$\mathscr{E}_4$	$\mathscr{S}_{1a}$	292	300	R	Mini
$\mathscr{E}_5$	$\mathscr{S}_{1b}$	292	300	R	Mini
$\mathscr{E}_6$	$\mathscr{S}_{1b}$	292	300	S	Mini

[a]This experiment considers the phases included in S, but not its sequences.

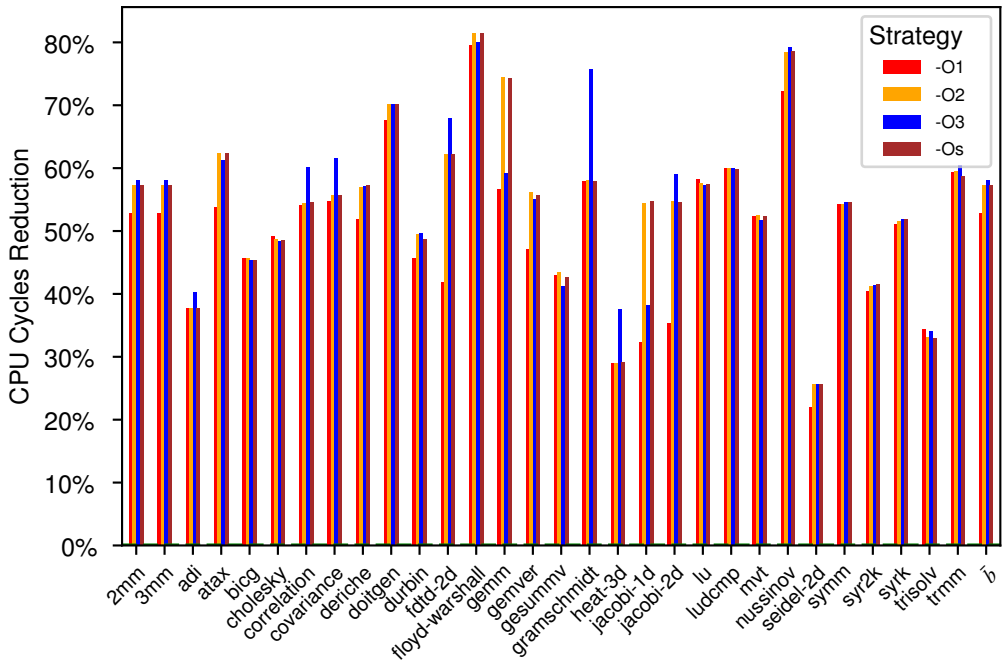


Figure 4.2: CPU cycles reduction of the standard optimizations against `-O0`.

4.2 Results

Optimizing for a specific metric such as the number of CPU cycles can degrade other metrics such as the memory footprint, the energy consumption, or compilation time, and we do not measure such degradation. Besides the degradation, ensuring each benchmark remains correct after applying a sequence of transformations is another essential aspect of the evaluation.

The benchmark suite used includes internal validations to ensure the results are valid. However, such guarantees could also be subject to code transformations; ideally, we would complement these validations with external ones. All experiments evaluated 96 unique phases.

4.2.1 Experiment $\mathscr{E}_0$

The best overall standard optimization from the common optimizations is `-O3` with a geometric mean speedup of 2.44x against `-O0` (Figure 4.2). The maximum speedup was 5.42x and obtained on `floyd-warshall` by `-Os`. The speedup difference between the standard optimizations is about the same for most benchmarks, but there are some relevant differences, such as in `gramschmith`, `gemm`, and `jacobi-1d`.

The standard optimization `-O3` was responsible for 45% of the maximum performance increases, while `-O1` achieved the maximum speedup for three benchmarks (Figure 4.3). Table 4.3 shows that different levels of optimizations combine various code transformations that do not always result in the best performance. Standard optimizations aim to improve as many programs as possible while balancing execution time, code size, energy consumption, and compilation time.

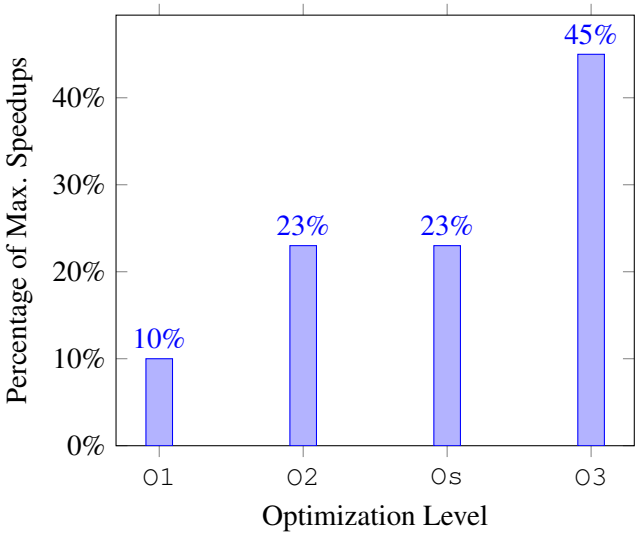


Figure 4.3: Percentage of maximum speedups per standard optimization against `O0`.

Table 4.3: Maximum speedups of the standard optimizations against `-O0` on the PolyBench 4.2.1.

Benchmark	Strategy	Speedup	Benchmark	Strategy	Speedup
`2mm`	`-O3`	5.31	`gramschmidt`	`-O3`	4.14
`3mm`	`-O3`	5.41	`heat-3d`	`-O3`	1.60
`adi`	`-O3`	1.67	`jacobi-1d`	`-Os`	2.21
`atax`	`-Os`	2.66	`jacobi-2d`	`-O3`	2.44
`bicg`	`-O2`	1.84	`lu`	`-O1`	2.39
`cholesky`	`-O1`	1.97	`ludcmp`	`-O2`	2.50
`correlation`	`-O3`	2.51	`mvt`	`-O2`	2.11
`covariance`	`-O3`	2.61	`nussinov`	`-O3`	4.80
`deriche`	`-Os`	2.34	`seidel-2d`	`-O2`	1.35
`doitgen`	`-Os`	3.36	`symm`	`-O3`	2.21
`durbin`	`-O3`	1.99	`syr2k`	`-Os`	1.71
`fdtd-2d`	`-O3`	3.12	`syrk`	`-Os`	2.08
`floyd-warshall`	`-Os`	5.42	`trisolv`	`-O1`	1.53
`gemm`	`-O2`	3.91	`trmm`	`-O3`	2.53
`gemver`	`-O2`	2.28	$\bar{b}$	`-O3`	2.44
`gesummv`	`-O2`	1.77			

4.2.2 Experiment $\mathscr{E}_1$

This experiment runs on a sub-set of benchmarks related to the maximum and minimum speedups achieved in experiments $\mathscr{E}_0$ and $\mathscr{E}_2$. This subset comprises the benchmarks `2mm`, `3mm`, `adi`, `correlation`, `floyd-warshall`, `gemm`, `heat-3d`, `jacobi-1d`, `jacobi-2d`, `nussi-nov`, `seidel-2d`.

The results in Figure 4.4 show that this experiment is the only one reducing the code size for `jacobi-1d` and `seidel-2d`. This experiment achieves the maximum reduction size of 20.8% for `seidel-2d`. The `jacobi-2d` benchmark obtains the maximum speedup of 40.9% concerning the number of CPU cycles.

4.2.3 Experiment $\mathscr{E}_2$

Figure 4.5 shows the results of applying the model of strategy $\mathscr{S}_{1a}$ to predict 300 new sequences with a length of 292, the same of `-O2` (see Figure 4.1), from the compiler predefined ones and then comparing them with the best `-O` (see Table 4.3).

This experiment found a phase order that reduces the number of CPU Cycles by 39.8% for `jacobi-2d` against `-Os`. Concerning the same benchmark and metric, it achieves a maximum of 1.5% on average against `-O3`. However, there are no improvements for benchmarks such as `gramschmidt`, `correlation`, and `covariance` for the number of CPU cycles, and there is no phase order that achieves an overall speedup.

This experiment is the only one that can significantly reduce the code size for the benchmarks `doitgen` and `gramschmidt`, with a reduction of 17%.

4.2.4 Experiment $\mathscr{E}_3$

We want to find the impact of having the Strategy $\mathscr{S}_{1a}$ searching on a more extensive search space than the standard sequences. This experiment searches for 300 phase orders with a length of 292 in a search space that consists of 10^5 random phase orders with an arbitrary length. This experiment runs on the same sub-set of benchmarks of Experiment $\mathscr{E}_1$.

The results depicted in Figure 4.6 show that in the benchmark `jacobi-2d`, most phase orders achieve a speedup in the CPU cycles of more than 20%. However, this experiment did not find phase orders that can have an overall speedup, and the code size has no significant impact.

4.2.5 Experiment $\mathscr{E}_4$

To understand the impact of updating the first node where we start searching for sequences, we run an experiment with the same parameters as Experiment $\mathscr{E}_3$, but we choose the first node randomly.

This experiment runs on the same sub-set of benchmarks that Experiment $\mathscr{E}_1$. It could also find a phase order that achieves the maximum speedup for the benchmark `gemm`; the code size does not have any significant impact.

No phase order can decrease the number of CPU cycles in all benchmarks, as described in Figure 4.7.

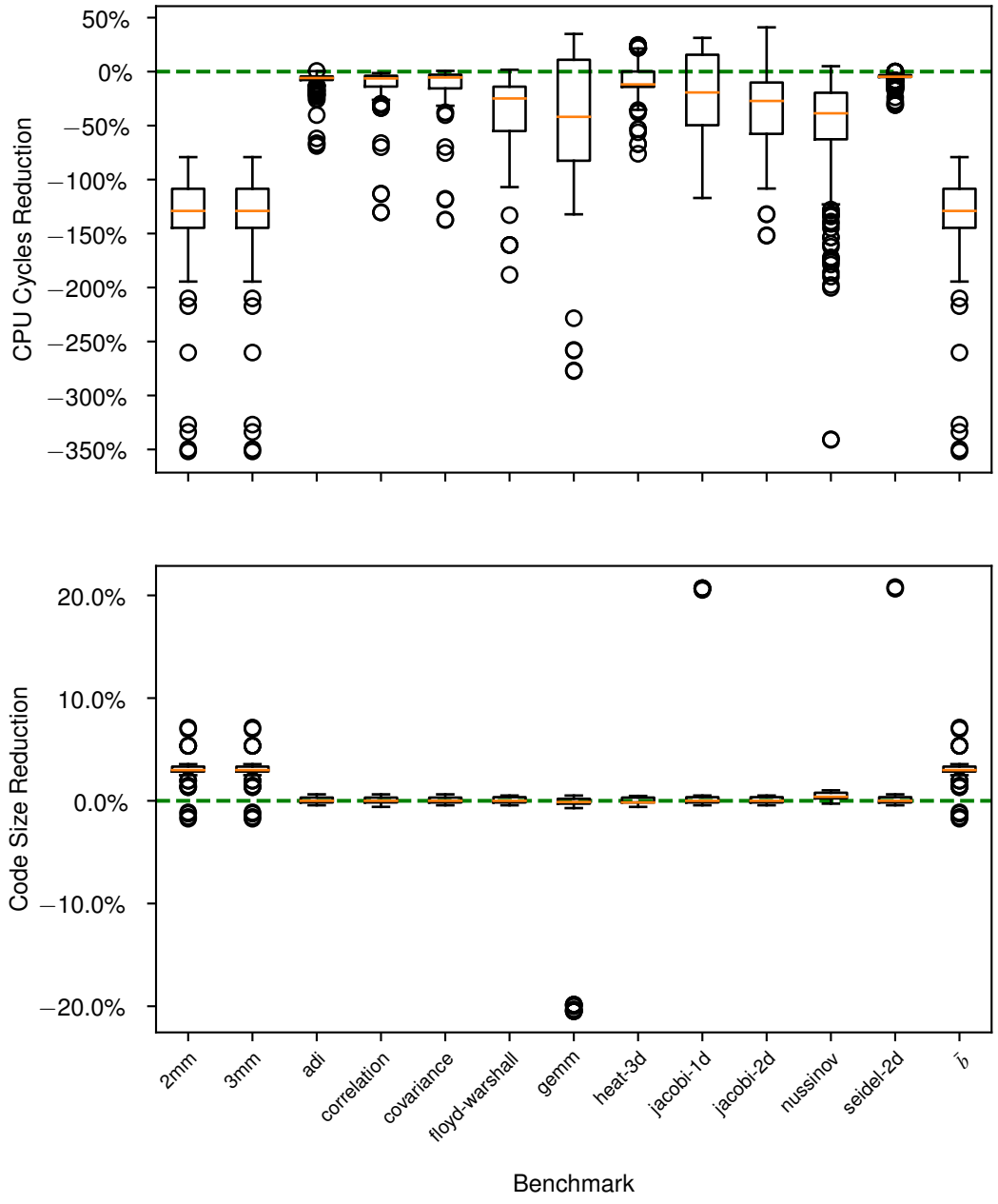


Figure 4.4: CPU cycles speedups and Code Size Decrease of Experiment $\mathscr{E}_1$ against the best `-O`.

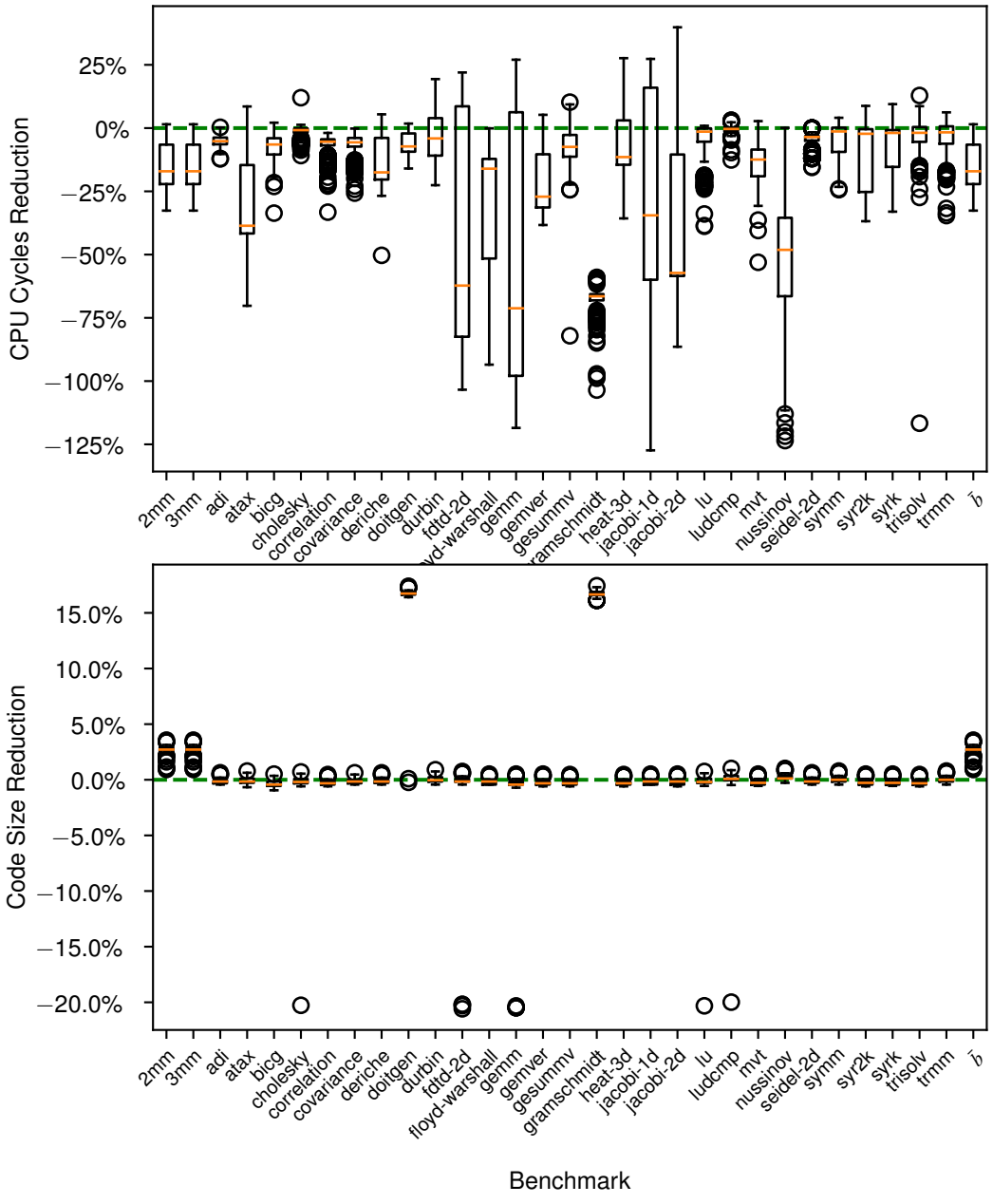


Figure 4.5: CPU cycles speedups and Code Size Decrease of Experiment $\mathscr{E}_2$ against the best `-O`.

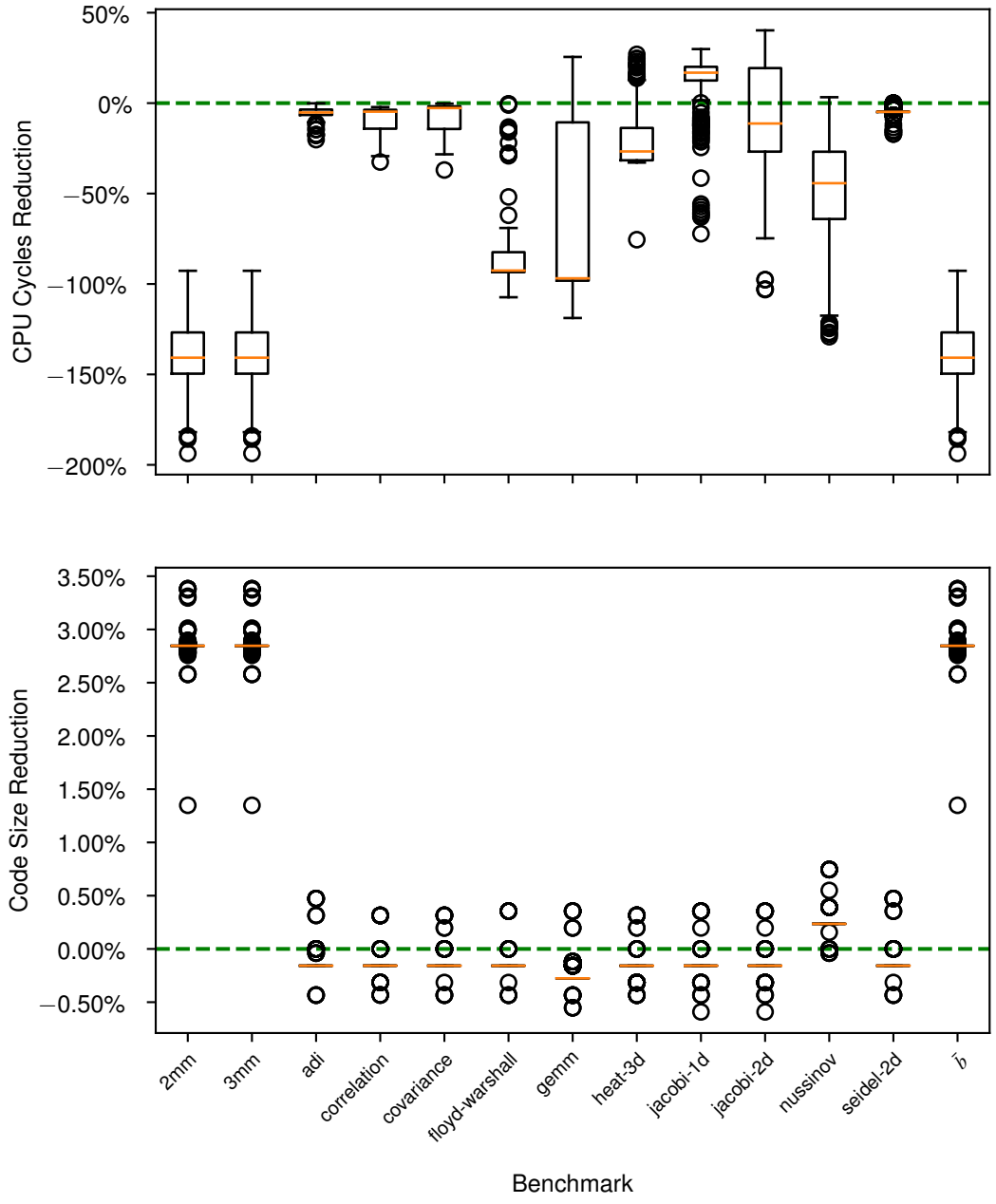


Figure 4.6: CPU cycles speedups and Code Size Decrease of Experiment $\mathscr{E}_3$ against the best `-O`.

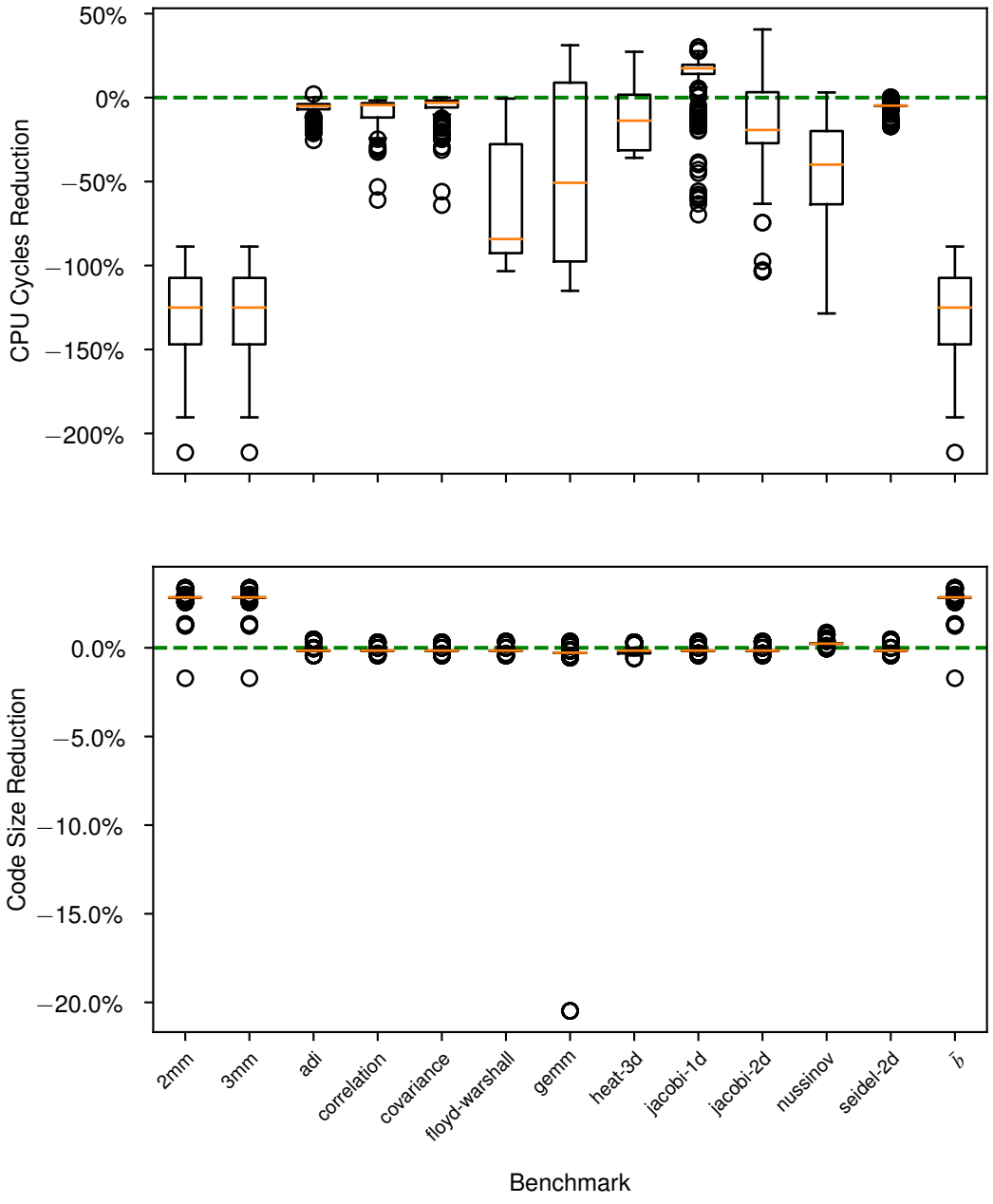


Figure 4.7: CPU cycles speedups and Code Size Decrease of Experiment $\mathscr{E}_4$ against the best `-O`.

4.2.6 Experiment $\mathscr{E}_5$

This experiment runs on the same sub-set of benchmarks that Experiment $\mathscr{E}_1$. To understand the impact of changing the search space, we use the same parameters as Experiment $\mathscr{E}_4$, but we use as the base graph the standard sequences.

This experiment does not find any phase order capable of achieving a speedup in the number of CPU cycles, the code size does not have any significant impact, and no phase order reduces the number of CPU cycles in all benchmarks (see Figure 4.8).

4.3 Overview

We evaluated the PolyBench benchmark suite, using the smallest dataset size, on a Raspberry Pi, and measured each benchmark by taking the number of CPU cycles from the performance counters. Each execution repeats ten times to ensure our results are the most accurate.

Additionally, we set the CPU to the max frequency, flush 33 MB of cache before each start, and set the scheduler policy to First In, First Out. We see the importance of having different standard optimizations because different levels achieve maximum speedups. However, most of them are through the highest level of optimization `-O3`. We also see potential in the standard optimizations.

We found speedups concerning the number of CPU cycles up to 1.66x against the best optimization level and an increase of 1.5% on average. We could also reduce the code size of some benchmarks up to 21%. However, as standard optimizations do, we could not achieve speedups in all kernels.

We see that the predefined compiler optimizations aim to improve most of programs, but that leaves room for improvement. The speedups we measured do not consider other metrics that could be affected by some trade-offs, such as energy consumption, the memory footprint, or the compilation time.

4.4 Summary

We designed experiments to evaluate the impact of different strategies against the best standard optimizations `O1`, `O2`, `O3`, and `Os`, and the effect of searching the common sequences `S` or the random sequences R.

We could achieve speedups concerning the number of CPU cycles by up to 1.66x and reduce the code size by up to 21% against the best optimization levels `1`, `2`, `3`, and `s`. In the context of our experiments, we could see a maximum speedup of the number of CPU cycles using a random search space. We saw that only the strategy $\mathscr{S}_{1a}$ could find a phase order that increases the overall performance, although the increase is less than 1%.

While strategy $\mathscr{S}_{1a}$ starts the search with the phase with the most occurrences, we study the impact of choosing a random node to initiate the sequence by implementing strategy $\mathscr{S}_{1b}$. The strategy $\mathscr{S}_{1b}$ could not find any increase in performance when searching the space `S` that contains

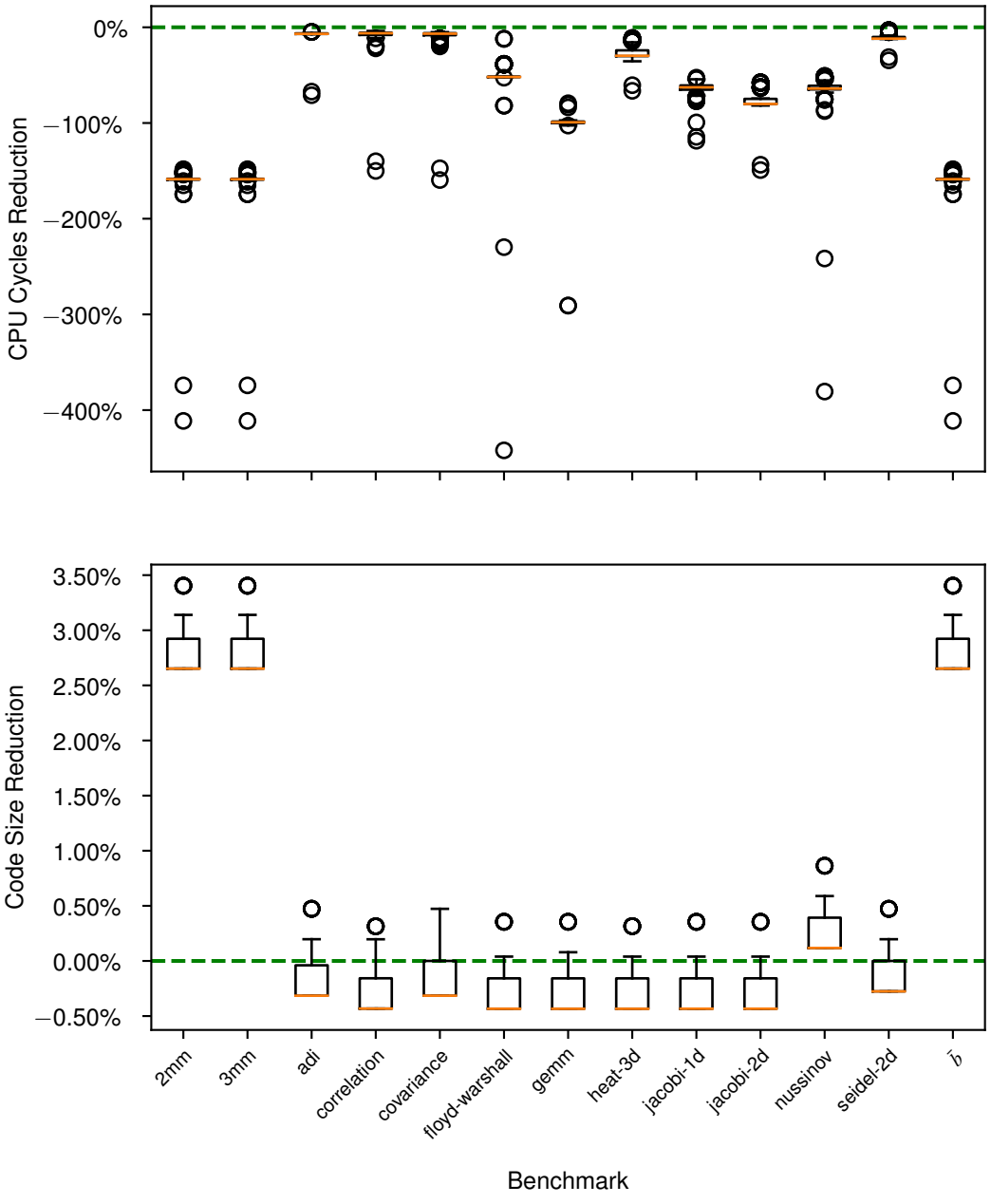


Figure 4.8: CPU cycles speedups and Code Size Decrease of Experiment $\mathscr{E}_5$ against the best `-O`.

the standard phase orders but could find a maximum speedup when searching a random search space.

Chapter 5

Conclusions

5.1 Concluding Remarks

We presented the phase ordering problem of finding the best sequence for a given program. We describe several strategies to approach this problem and compare them against the predefined sequences of the LLVM compiler.

We conducted experiments using 30 benchmarks of the PolyBench repository. The standard sequences could achieve reductions, in the number of CPU cycles, up to 80% with respect to `O0`. The optimization level `O3` represented 45% of the maximum reductions achieved. These results show the importance of optimizations and of having different optimizing levels.

Finding new sequences from standard and random ones, we could achieve a speedup of up to 1.66x for the number of CPU cycles against the best optimization level *O*. We could also reduce the code size by 21% against *O*. These results show that these predefined sequences specially crafted by compiler developers are far from the optimal ones. Those can change from program to program. We note, however, that these results did not fully check if the benchmarks remain valid after being optimized.

In order to evaluate the proposed strategies, we developed and open-sourced three software packages:

1. The engine `dervin` implements the strategies and recommends phase orders.
2. The framework `ghopper` takes phase orders and measures them on a benchmark suite.
3. The third software package `amidala` analyzes the collected data.

5.2 Future Work

Our strategies do not take advantage of the information about phase dependencies. The software we built to conduct our experiments has limitations. More precisely, the program `ghopper` that runs on the target to measure the strategies only works with an LLVM compiler and up to version 12.0.1 since other versions introduce changes to the optimizer interface. Another limitation is that

we are not testing the benchmark's correctness after being optimized. The PolyBench, through the macro `POLYBENCH_DUMP_ARRAYS`, sends output to the standard error we can use to validate each program.

The target also needs to be able to run a Python environment. For some targets, this is not possible. Our software also introduces overhead on the target because it is the target that optimizes besides executing the benchmarks. An alternative is to cross-compile each benchmark. A C program can listen on a socket for a pre-optimized benchmark, execute it, and send the metrics back. This program should be pre-compiled and run directly by the target.

The proposed strategies and our engine `dervin` do not consider parameterized phases, such is the case of the *loop unrolling*.

Additional strategies can be evaluated. An example of such strategy consists in combining a randomness factor that allows us to explore more of the search space, with favoring phases that contributed to the best phase order known for the training programs that share the most features with the optimized one.

To further improve the model of our strategies, we can enhance the search heuristic with more knowledge about the phases. For example, we could store in a graph data-structure (1) pairs of phases to avoid and (2) phase dependencies. Information related to (1) and (2) derives from the LLVM documentation, source code, and by trial and error. Finally, we recommend evaluating more benchmark suites besides the PolyBench.

References

[1] M. R. Jantz and P. A. Kulkarni, "Exploiting phase inter-dependencies for faster iterative compiler optimization phase order searches," in *2013 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, 2013, pp. 1–10.

[2] S.-A.-A. Touati and D. Barthou, "On the decidability of phase ordering problem in optimizing compilation," in *Proceedings of the 3rd Conference on Computing Frontiers*, ser. CF '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 147156. [Online]. Available: https://doi.org/10.1145/1128022.1128042

[3] S. Kulkarni and J. Cavazos, "Mitigating the compiler optimization phase-ordering problem using machine learning," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 147162. [Online]. Available: https://doi.org/10.1145/2384616.2384628

[4] W. A. Wulf, R. K. Johnsson, C. B. Weinstock, S. O. Hobbs, and C. M. Geschke, *The Design of an Optimizing Compiler*. USA: Elsevier Science Inc., 1975.

[5] S. R. Vegdahl, "Phase coupling and constant generation in an optimizing microcode compiler," *SIGMICRO Newsl.*, vol. 13, no. 4, p. 125133, oct 1982. [Online]. Available: https://doi.org/10.1145/1014194.800942

[6] C. Click and K. D. Cooper, "Combining analyses, combining optimizations," *ACM Trans. Program. Lang. Syst.*, vol. 17, no. 2, p. 181196, mar 1995. [Online]. Available: https://doi.org/10.1145/201059.201061

[7] P. A. Kulkarni, D. B. Whalley, G. S. Tyson, and J. W. Davidson, "Practical exhaustive optimization phase order exploration and evaluation," *ACM Trans. Archit. Code Optim.*, vol. 6, no. 1, Apr. 2009. [Online]. Available: https://doi.org/10.1145/1509864.1509865

[8] H. Wang, Z. Tang, C. Zhang, J. Zhao, C. Cummins, H. Leather, and Z. Wang, "Automating reinforcement learning architecture design for code optimization," in *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*, ser. CC 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 129143. [Online]. Available: https://doi.org/10.1145/3497776.3517769

[9] R. J. F. Nobre, "Efficient Target and Application Specific Selection and Ordering of Compiler Passes," Ph.D. dissertation, Faculdade de Engenharia da Universidade do Porto, October 2017.

[10] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the International Symposium on Code Generation and*

Optimization: Feedback-Directed and Runtime Optimization, ser. CGO 04. USA: IEEE Computer Society, 2004, p. 75.

[11] ARM Ltd., "Architecting a Smarter World — ARM," https://www.arm.com.

[12] J. M. Cardoso, J. G. F. Coutinho, and P. C. Diniz, "Chapter 5 - source code transformations and optimizations," in *Embedded Computing for High Performance*, J. M. Cardoso, J. G. F. Coutinho, and P. C. Diniz, Eds. Boston: Morgan Kaufmann, 2017, pp. 137–183. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780128041895000053

[13] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. O'Boyle, J. Thomson, M. Toussaint, and C. Williams, "Using machine learning to focus iterative optimization," in *International Symposium on Code Generation and Optimization (CGO'06)*, 2006, pp. 11 pp.–305.

[14] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. F. O'Boyle, and O. Temam, "Rapidly selecting good compiler optimizations using performance counters," in *International Symposium on Code Generation and Optimization (CGO'07)*, 2007, pp. 185–197.

[15] Y. Chen, S. Fang, Y. Huang, L. Eeckhout, G. Fursin, O. Temam, and C. Wu, "Deconstructing iterative optimization," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 3, oct 2012. [Online]. Available: https://doi.org/10.1145/2355585.2355594

[16] A. H. Ashouri, G. Mariani, G. Palermo, E. Park, J. Cavazos, and C. Silvano, "Cobayn: Compiler autotuning framework using bayesian networks," *ACM Trans. Archit. Code Optim.*, vol. 13, no. 2, jun 2016. [Online]. Available: https://doi.org/10.1145/2928270

[17] P. A. Kulkarni, S. R. Hines, D. B. Whalley, J. D. Hiser, J. W. Davidson, and D. L. Jones, "Fast and efficient searches for effective optimization-phase sequences," *ACM Trans. Archit. Code Optim.*, vol. 2, no. 2, p. 165198, June 2005. [Online]. Available: https://doi.org/10.1145/1071604.1071607

[18] G. Fursin, Y. Kashnikov, A. W. Memon, Z. Chamski, O. Temam, M. Namolaru, E. Yom-Tov, B. Mendelson, A. Zaks, E. Courtois, F. Bodin, P. Barnard, E. Ashton, E. Bonilla, J. Thomson, C. K. I. Williams, and M. O'Boyle, "Milepost GCC: Machine learning enabled self-tuning compiler," *International Journal of Parallel Programming*, vol. 39, no. 3, pp. 296–327, June 2011. [Online]. Available: https://doi.org/10.1007/s10766-010-0161-2

[19] A. H. Ashouri, A. Bignoli, G. Palermo, C. Silvano, S. Kulkarni, and J. Cavazos, "Micomp: Mitigating the compiler phase-ordering problem using optimization sub-sequences and machine learning," *ACM Trans. Archit. Code Optim.*, vol. 14, no. 3, Sep. 2017. [Online]. Available: https://doi.org/10.1145/3124452

[20] LLVM, "LLVM's Analysis and Transform Passes — LLVM 12.0.1 documentation," https://llvm.org/docs/Passes.html.

[21] GCC, "GCC, the GNU Compiler Collection," https://gcc.gnu.org.

[22] R. Nobre, L. Reis, and J. M. P. Cardoso, "Compiler phase ordering as an orthogonal approach for reducing energy consumption," *CoRR*, vol. abs/1807.00638, 2018. [Online]. Available: http://arxiv.org/abs/1807.00638

[23] A. H. Ashouri, G. Palermo, J. Cavazos, and C. Silvano, *Automatic Tuning of Compilers Using Machine Learning*, ser. SpringerBriefs in Applied Sciences and

Technology. Cham: Springer International Publishing, 2018. [Online]. Available: http://link.springer.com/10.1007/978-3-319-71489-9

[24] R. Nobre, L. G. A. Martins, and J. a. M. P. Cardoso, "A graph-based iterative compiler pass selection and phase ordering approach," in *Proceedings of the 17th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, Tools, and Theory for Embedded Systems*, ser. LCTES 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 2130. [Online]. Available: https://doi.org/10.1145/2907950.2907959

[25] Q. Huang, R. Lian, A. Canis, J. Choi, R. Xi, S. Brown, and J. Anderson, "The effect of compiler optimizations on high-level synthesis for FPGAs," in *2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*, 2013, pp. 89–96.

[26] S. Purini and L. Jain, "Finding good optimization sequences covering program space," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, January 2013. [Online]. Available: https://doi.org/10.1145/2400682.2400715

[27] L. G. Martins, R. Nobre, A. C. Delbem, E. Marques, and J. a. M. Cardoso, "Exploration of compiler optimization sequences using clustering-based selection," *SIGPLAN Not.*, vol. 49, no. 5, p. 6372, June 2014. [Online]. Available: https://doi.org/10.1145/2666357.2597821

[28] L. G. A. Martins, R. Nobre, J. a. M. P. Cardoso, A. C. B. Delbem, and E. Marques, "Clustering-based selection for the exploration of compiler optimization sequences," *ACM Trans. Archit. Code Optim.*, vol. 13, no. 1, Mar. 2016. [Online]. Available: https://doi.org/10.1145/2883614

[29] U. Garciarena and R. Santana, "Evolutionary optimization of compiler flag selection by learning and exploiting flags interactions," in *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion*, ser. GECCO '16 Companion. New York, NY, USA: Association for Computing Machinery, 2016, p. 11591166. [Online]. Available: https://doi.org/10.1145/2908961.2931696

Appendices

Appendix A

PolyBench Benchmark Suite Adaptations

We have made the following customizations to the PolyBench suite[1]: (1) changes in the file structure, (2) changes to each benchmark output a JSON with metrics, and (3) each benchmark is a non-optimized bitcode.

A.1 Adapting the File Structure

We describe the enabled options provided by PolyBench, how we converted all the benchmarks to bitcodes in the same folder flatted and how we updated `polybench.c` to give the metrics in a JSON format with specific keys.

The reason we place all the benchmarks in the same folder and make each benchmark output a JSON file is for convenience. The engine we developed expects all benchmarks to be in the same folder so there is no need to parse the file directory, that each benchmark is a bitcode so the engine does not have to know how to link each benchmark and that each benchmark outputs the metrics in JSON with specific keys, so the engine can easily collect metrics.

A.2 Metrics Output in JSON

We updated the function `polybench_papi_print()` located in the file `utilities/polybench.c` to output the CPU cycles in the JSON format. The output has the following structure:

```
{
  "cpu_cycles": 100,
  "l1_dcm": 200,
  "l2_dcm": 300,
  "tlb_dm": 400,
  "tlb_im": 500,
```

[1] https://github.com/MatthiasJReisinger/PolyBenchC-4.2.1

```
"hw_int": 600,
"br_msp": 700,
"tot_ins": 800,
"ld_ins": 900,
"sr_ins": 1000,
"br_ins": 900,
"l1_dca": 800,
"l2_dca": 700
}
```

A.3 Emit Non-Optimized Bitcodes

We emit non-optimized bitcodes because it is easier to optimize each benchmark, since each benchmark is now a standalone file without dependencies. The framework does not need to know how to link each benchmark. See the Appendix B.1.

Appendix B

Optimizing a Program with LLVM 12.0.1

B.1 Emit Non-Optimized Bitcode

To emit non-optimized bitcode for one benchmark we run the following script:

```
#!/usr/bin/env bash

papi_dir=/opt/papi
includes=-I polybench -I {papi_dir}/include
defines=-D MINI_DATASET -DPOLYBENCH_LINUX_FIFO_SCHEDULER\
  -DPOLYBENCH_PAPI
emit_bitcode_options=-emit-llvm -c
non_opt_bitcode_options=-O0 -Xclang -disable-O0-optnone\
  -Xclang -disable-llvm-passes \
options=${includes} ${non_opt_bitcode_options}\
  ${emit_bitcode_options} ${defines}

clang ${options} benchmark.c
```

We assume that PAPI[1] is installed at `/opt/papi`.
Then we link each bitcode with the utility `polybench.bc` using the `llvm-link`[2]:

```
#!/usr/bin/env bash

llvm-link benchmark.bc polybench.bc -o benchmark.bc
```

[1] https://icl.utk.edu/papi/software/index.html
[2] https://llvm.org/docs/CommandGuide/llvm-link.html

B.2 Optimize Bitcode

To optimize a bitcode we invoke opt[3]:

```
#!/usr/bin/env bash

opt -a -b -c benchmark.bc -o benchmark.opt.bc
```

In this case, we assume the phase ordering: $a \rightarrow b \rightarrow c$.

B.3 Emit Optimized Binary

To emit an optimized binary we start by using the `llc`[4] to emit an optimized object file:

```
#!/usr/bin/env bash

llc -filetype=obj benchmark.opt.bc -o benchmark.o
```

Then, we convert the optimized object file to an executable through `clang`[5]:

```
#!/usr/bin/env bash

clang -lm -L${PAPI_DIR}/lib -lpapi benchmark.o -o benchmark
```

B.4 How to Find the Phases Executed by the Optimizer

When we optimize a program with the sequence $a \rightarrow b \rightarrow c$, the optimizer can add phases before or after the sequence or even execute more sequences. The executed sequences can be obtained with:

```
#!/usr/bin/env bash

opt -a -b -c -debug-pass=Arguments benchmark.c
```

The optimizer will use the standard error to output the sequences, and the output follows this format:

```
Pass Arguments: -a -b -c
Pass Arguments: -d -e -f
...
```

[3] https://llvm.org/docs/CommandGuide/opt.html
[4] https://llvm.org/docs/CommandGuide/llc.html
[5] https://clang.llvm.org

B.5 Flow From Non-Optimized Bitcode to Optimized Binary

```
#!/usr/bin/env bash

opt -p1 -p2 -p3 program.bc -o program.bc
llc -filetype=obj program.opt.bc
clang program.opt.bc -o program
```

Appendix C

Software Packages Developed

C.1 Package `dervin`

C.1.1 How to Install

Installing this package can be done using the Python package installer `pip`[1]:

```
pip install dervin
```

C.1.2 Example of the Output for the Package `dervin`

The `dervin` package can run without arguments and its default behavior is to find random sequences from the `clang` optimization level `O3`.

```
dervin --pretty
```

```
{
  "parameters": {
    "length": "5",
    "cardinality": "1",
    "strategy": "s0",
    "seed": "o3"
  },
  "output": [
    {
      "metadata": {
        "length": 5,
        "cardinality": 1
      },
      "sequences": [
        "-lazy-block-freq -function-attrs -tailcallelim -licm
        -loop-rotate"]}]}
```

[1] https://pip.pypa.io/en/stable/installation

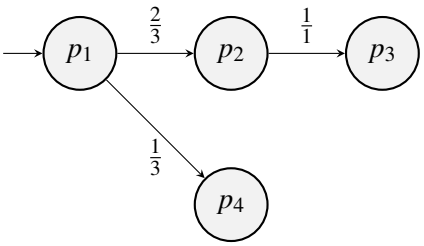
C.1.3 Search Space for Strategy $\mathscr{S}_0$

One of the strategies implemented by the Design Space Exploration Engine is a strategy that builds a random phase order from a set of phases. The engine accepts a set of phases as a file with the same structure as the optimizer output:

```
Pass Arguments: -p1 -p2 -p3
Pass Arguments: -p4 -p5 -p6
...
```

C.1.4 Search Space for Strategy $\mathscr{S}_{1a}$

The strategy $\mathscr{S}_{1a}$ uses a search space such as the one represented here:



Which is stored in a JSON file:

```
{
  "directed": true,
  "multigraph": false,
  "graph": {},
  "nodes": [
    {"id": "p1"},
    {"id": "p2"},
    {"id": "p3"},
    {"id": "p4"}],
  "links": [
    {"weight": 0.333, "source": "p1", "target": "p2"},
    {"weight": 0.666, "source": "p1", "target": "p4"}
    {"weight": 1.000, "source": "p2", "target": "p3"}
  ]
}
```

C.2 Package `ghopper`

C.2.1 How to Install

Installing this package can be done using the Python package installer `pip`[2]:

[2] https://pip.pypa.io/en/stable/installation

```
pip install ghopper
```

C.2.2 Executing a Benchmark

We emit standalone bitcodes with `llvm-link`, linking the necessary sources:

```
#!/usr/bin/env bash

llvm-link benchmark.bc lib.bc -o benchmark.bc
```

Having a set of standalone benchmarks, we end up with benchmark suite folder like this:

```
benchmark-suite/
  bench1.bc
  bench2.bc
  ...
```

Then for each program, we place the optimized bitcode, the optimized object file, and the binary in the `/tmp` folder, so for each execution, our `/tmp` folder will be like:

```
/tmp/
  benchmark.opt.bc
  benchmark.opt.o
  benchmark
```

Finally, we execute `/tmp/benchmark`, delivering output in JSON with several metrics. Moreover, we collect this metric alongside other data that we will describe later and store them in a CSV file.

C.2.3 Evaluating a Benchmark Remotely

We evaluate a benchmark remotely on a computing board through SSH. If the remote connection is interrupted, the current experiment is aborted. To allow the experiment to run continuously, we use the `nohup` program:

```
nohup ghopper --config experiment.yml &
```

C.3 Package `amidala`

C.3.1 How to Install

Installing this package can be done using the Python package installer `pip`:

```
pip install amidala
```

C.3.2 Example of Usage

Given a CSV file with numerical results of multiple experiments, we can compare experiments $\mathscr{E}_1$ and $\mathscr{E}_2$ by showing a plot:

```
amidala plot e1 e2 plot1
```

We can also open a Python shell to query the data:

```
amidala shell e1 e2
```

And within the shell we have access, for example, to speedups:

```
>>> df.speedups
                          cpu_cycles ...  binary_size
phase_order_id benchmark                 ...
             x         a          1.20  ...  1.88
                       b          1.33  ...  0.55
            ...      ...           ...  ...   ...
             y         a          1.11  ...  0.23
                       b          1.15  ...  2.33
```

Moreover, we can ask for a numerical summary:

```
amidala summary e1 e2
```

```
Numerical summary
E1 v. E2:
 Experiments description:
  E1:
   strategy: S1a
   parameters:
    max length: 130
    cardinality: 200
  E2:
   strategy: O3
   parameters: N/A
 Summary:
  Max speedup: gemm (1.34x, 64% reduction)
  Invalid sequences: 34 out of 340 (10%)
  # of speedups > 10%: 4 out of 30
  # of speedups: 40 out of 100
  ...
```