Compiler Flag-Selection and Phase-Order Optimization via Machine Learning in the Glasgow-Haskell Compiler

Thesis by
Connor Rhys Peper

In Partial Fulfillment of the Requirements for the
Degree of
International Computer Science

ROSE-HULMAN INSTITUTE OF TECHNOLOGY
Terre Haute, Indiana

2024

ACKNOWLEDGEMENTS

I am extremely grateful to my thesis advisor, Dr. Mike Hewner, here at the Rose-Hulman Institute of Technology. Without him, this thesis would probably never have been written, or at least it would be of significantly diminished quality. His insight and suggestions have never served me wrong and there is no other professor at the Institute I would rather have on my side.

I am also deeply indebted to my childhood and best friend, Andrew Reese, for being the only person outside of academics who showed a genuine interest in my work. More importantly, however, is that he was always eager to proof-read and offer suggestions based on his understanding as a lay-person to the topic. Thank you, Andrew.

I would like to thank Dr. Jason Yoder, also at Rose-Hulman, for his dedication to research at Rose-Hulman. His deadlines are another thing that kept me focused and progressing throughout this school year.

I would like to acknowledge Prof. Dr. Georg Schied. He is my co-advisor at the Ulm College of Applied Sciences. Professor Schied, despite being across the Atlantic, was always eager to help. He trusted me and gave me leverage for which I am thankful.

Finally, I could not have remained sane throughout this entire process were it not for the presence and encouragement of my friends and family, especially my sister Courtney, who just so happens to be at Rose-Hulman along with me.

ABSTRACT

Little research has been done into automating the flag selection problem and even less research has been done into automating the phase-order problem in the Glasgow-Haskell Compiler. In this thesis, I implement three machine learning algorithms to solve the flag-selection problem: Random Iterative Optimization (RIO), Flag Optimization using Genetic Algorithm (FOGA), and Bayesian Optimization for Compiler Auto-tuning (BOCA). Additionally, I implement RIO and a modified BOCA for the phase-order problem. I achieved across the board run-time improvement over the -O2 optimization flag preset in the GHC and across the board run-time improvement over the default phase-order. Modifications to the GHC Driver were required to allow for modification of the optimization phase-order. While a program specific optimal flag selection and phase-ordering was found for each test application, FOGA and BOCA did not converge on the same optimal ordering, and it was also found that the default phase-ordering in the simplifier is already very effective across most applications at finding an optimal phase-order, though there is room for improvement.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

Chapter 1

INTRODUCTION

Most programs, even for real-time or critical systems, are written in a high-level programming language because it's easier to write, read, and modify. However, high-level code can sometimes be inefficient when it is translated to machine language. There is a trade-off that every developer makes when they decide to write something in C, C++, Java, etc... instead of in an assembly languages. Instead, developers delegate the responsibility for making efficient machine codes to compilers. A compiler translates high level programming languages into machine code. Compilers can apply optimizations to code to make it more efficient. An example of an optimization in the GHC [24] is the `-floopification`, which will turn tail-end self-recursive calls into local jumps instead of function calls. Since the invention of the first optimizing compiler, FORTRAN I, compilers have long improved to a state where there are many potential optimizations, so many that the optimizations now interfere with each other, so many that developers could not be expected to know the optimal set of flags. For example, the GHC compiler contains over 50 possible optimizations, allowing for over one quadrillion possible combinations. For a compiler with N possible optimization flags, there exist $\sum_{k=0}^{N}(\binom{N}{k})$ possible combinations. Furthermore, with so many flags, a developer might not be familiar with which flags are even applicable to their code and would have any impact.

For this reason, developers of compilers began to create presets of flags; what they think might be a good general set of optimizations for a general program. In the GHC, there are three flag presets:

- **-O0** - Disables all optimization.
- **-O1** - Equivalent to **-O**. Described in the GHC documentation as, "Generate good-quality code without taking too long about it."
- **-O2** - Anything greater than -O2 (-O3, -O4, ...) is equivalent to -O2. Described in the GHC documentation as, "Apply every non-dangerous optimisation, even if it means significantly longer compile times."

A developer can just pass in one of these flags as a substitute for passing in dozens of specific compiler optimization flags. For most programmers and most programs, these three flags (-O0, -O1, -O2) might be sufficient. Even though -O2 has the most optimizations included, it might not be the best for a specific program. Despite the name "optimization," not all optimizations will actually improve the runtime of a specific program. Some optimizations might actually *decrease* program performance! Instead of -O2, every program has a set of flags, *specific to it*, which will achieve the maximum speed-up. The big challenge is determining what this set of flags is. This is called the "Flag Selection Problem." This is the start of compile-time optimizations, which naturally led to the first attempts at compiler auto-tuning. My approach took three known auto-tuning techniques that have been applied to the GCC and LLVM C compilers and apply them to the Glasgow-Haskell Compiler.

The GHC is a good candidate for optimizing the flag-selection problem because so little research has been done on the GHC on flag-selection. Therefore, I believe the GHC could be ripe for exploration in this area. However, flag-selection is not the only way to impact optimizations and their effect on the source code, it also depends on the order in which those optimizations are applied [7].

When a optimization is selected, the compiler will use an algorithm to find areas where the optimization can be applied. A compiler might have several algorithms to do the same common optimization. For example, consider the following two algorithms for removed redundant expressions: Denominator Value Numbering (DVNT), which replaces redundant computations with early references and also performs constant folding, and Global Common Sub-expression Elimination (GCSE), which uses data flow analysis and then does another compiler pass to replace redundant computations with previous computations. These two methods eliminate redundant computations, but they are different. For example, GCSE cannot identify $x \cdot y$ equals $x \cdot z$ when $y = z$. which DVNT can [7]. Now, suppose a developer was fully knowledgeable about each optimization flag and was knowledgeable about the algorithms that go into each flag, there are limits to what a programmer can tell the compiler to do. While a developer has full freedom to choose *which* optimizations are applied, he has no control over *how* they are applied. More specifically, the order in which the optimizations are applied is determined by the compiler.

Having an even greater control over the compiler's optimization pipeline, *the phase order*, allows a specific program's runtime to be improved further than what is solely

possible through the optimal selection of the compiler flags. However, like compiler flags, this is very large search space. To do an exhaustive search of all possible phase-orders would be $O(n!)$, and thus far from being a reasonable approach. To mediate this problem, I invoke compiler auto-tuning for the phase-order problem as well. By using compiler auto-tuning I can attempt to find an optimal phase order by exploring only a fraction of the total search space. My approach uses two machine learning techniques, RIO and a modified BOCA (BOCPA), to optimize the compiler phase order.

The Haskell Compiler is a great candidate for phase-order optimization. Ideally, compiler passes would be implemented in a strict pipeline that could be re-ordered. GHC is written primarily in Haskell, and Haskell, as a purely functional language, should, in theory, allow me to re-order optimization passes without fear of side-effects. Furthermore, while some domain knowledge of the Haskell compiler is obviously required to modify GHC to support this change, the flexible nature of the pipeline allows me to attempt changes without possessing in-depth knowledge of how each optimization is modifying the original source code. In theory, I should have great flexibility changing the order optimizations are applied.

The two big problems addressed in this thesis are *Do optimal, program-specific sets of flags exist* and *Does changing the phase-order impact program runtime positively?*. The answer to both these questions is "yes." In this thesis, I implemented three machine learning algorithms: RIO, BOCA, and FOGA to solve the flag-selection problem. Then I modified the GHC to accept different phase-orders when compiling, and I implemented RIO and a modified BOCA (BOCPA) to solve the phase-order problem. I achieved speed-up among all test programs when optimizing flag selection and even greater speed-up when optimizing the phase-order. All algorithms implemented were effective in decreasing the runtime.

Chapter 2

BACKGROUND & RELATED WORK

The task of getting compilers to improve themselves through code-generation and optimization is called *auto-tuning* [4]. Compiler auto-tuning has taken many forms throughout its existence. The different forms of auto-tuning and the history of auto-tuning leading up to the approaches I hope to implement is discussed below.

2.1 Early Stages of Compiler Auto-Tuning

One of the earliest examples of compiler auto-tuning is Feedback-Directed Optimization (FDO), otherwise known as Profile-Directed Optimization (PDO). This approach involves gathering statistics and information either while the program is running (online approaches), or after the program has finished executing (offline approaches), and then using that information to better compile the program. Many implementations of FDO exist, but the one I will be focusing on are the offline approaches.

Offline approaches involve running an application, gathering metrics (profiling), and then using those metrics to make decision about compilation and optimization. For example, using the number of memory allocations a program performed to make an informed decision about memory-related compiler flags. Importantly, offline FDO occurs *after* the program has finished executing and requires a programmer to re-compile after making changes. Online FDO can suffer from large overhead during execution which may, in-fact, decrease performance [22].

Conceptually though, the main problem with FDO is its requirement of domain knowledge. Someone *has* to write the FDO tool, and that person *must* be knowledgeable about the compiler. Furthermore, while offline FDO approaches aim to make educated guesses, that is fundamentally all they are. There is no guarantee that the choices the FDO will have any meaningful impact on the performance of a program. This is a problem machine learning aims to solve.

2.2 Auto-Tuning via Machine Learning

Machine learning approaches do not require domain knowledge to implement. Many machine learning approaches do not rely on the underlying architecture of program-

ming language, and thus can be easily transferred between computers and programs. They also have particular strengths in exploration and exploitation that can rival the educated guesses of offline FDO approaches. Bayesian approaches make guesses themselves as to future performance, their guesses do not rely on and are independent of specific knowledge of the program they are optimizing or the compiler they are using.

Phase Order Problem

As mentioned earlier, compiler optimization order does matter. Naturally, the first research conducted on this topic had fixed-orders in mind. For example, Whitfield and Soffa detail an axiomatic method for determining the best phase-order [28]; however, this approach only sought to maximize the potential of each optimization, not select the best optimizations for a specific program. Order also matter for specific programs. This was confirmed by Cooper et al. when they found an adaptive compiler that could change its phase-order performed better than a compiler with a fixed-phase order [7].

However, while the order does matter, the Phase Order Problem is a harder problem than the optimization selection problem [4]. One of the elements that make certain orders better than others is potential "lock-out" of some optimizations if other optimizations are performed earlier. That is, doing Optimization A first changes the source code that prevents Optimization B from being applied; however, had Optimization B been applied, the runtime may have been approved further. The Phase Order Problem is computationally expensive resulting from its large search space, even greater than that of optimization flag combinations. Given a compiler with n distinct optimization passes, there are $\Omega(n!)$ phase-orders to compute. Considering some optimization passes could be applied multiple times, the search space can be even greater.

Plenty of research has been done into The Phase-Order Problem, including with other machine learning approaches.

Kulkarni et al. throw the entire phase-order problem on its head. They contend that its not a large search space. Some compiler passes when applied, depending on the program architecture, will not change the code at all, and if they remove these types of permutations from consideration, the search space becomes more manageable [13]. Kulkarni and his team designed an exhaustive algorithm to test phase-orders on the Very Portable Optimizer (VPO) compiler. Their algorithm

seeks to identity dormant phases, where a compiler pass does nothing, and identical phases, where the result of one phase order will be the same as another. In a similar fashion to the algorithm presented in this paper, they construct an acyclic directed graph to represent phases. One problem that Kulkarni et al. encountered was inter-dependencies among different optimizations, which made changing their ordering forbidden [13]. It is this problem I hope to avoid by using the GHC. GHC optimization pipeline is structured as a list of compiler passes. As mentioned, Haskell's purely functional nature should make it less dramatic to move compiler passes around.

Ashouri et al. introduced the Mitigates the Compiler Phase-Ordering Problem (MiCOMP) framework [17]. MiCOMP is a mixed offline and online machine-learning approach which examines the original source-code to make predictions on the best phase-order in the LLVM C Compiler. Ashouri et al. decided to split -O3 into subsections and then form clusters based on high-performing subsections of -O3. MiCOMP, unlike the approaches used in this paper, uses additional information besides just the application run-time. MiCOMP, in its offline stage, will use a representative sample of programs for a target architecture and gather data on their compilation and subsequent runtime. This is the training phase. This is only performed once. In the online phase, new programs are fed into the model which extracts features from the program and makes a prediction on the best optimization sequence[17]. Ashouri et al. used the same set of optimizations used in -O3. Given the same set, MiCOMP was able to find alternative orders that not only met the performance of -O3, but exceeded it. Ashouri et al. The average speed-up gained was 31 % over the default ordering from exploring 0.001% of the search space. Unsurprisingly, they also found that larger compilation sequences cause better performance. Ashouri et al.'s experimental results reinforce that phase-order does matter [17].

Kulkarni and Cavazos set out to do not just program-specific phase-ordering, but *function specific* phase-ordering in the Jikes Research Virtual Machine (RVM), a compiler for Java. They used function features to make predictions on the best phase-ordering using Artificial Neural Networks (ANN) induced by Neuro-Evolution for Augment Topologies (NEAT) and compare their results to a traditional Genetic Algorithm [14]. Jikes RVM uses Just-In-Time (JIT) Compilation, therefore the optimizations are performed at runtime, which also allows Kulkarni and Cavazos to easily use features that are not just the runtime. Like MiCOMP, ANN requires an

expensive training phase which is hopefully counter-balanced by a quicker prediction phase. Kulkarni and Cavazos treat the phase-order problem as *Markov Property*; each individual method in an application is treated as its own property, with no consideration for the environment of the rest of the code [14]. Kulkarni and Cavazos used four different Java bench-marking suites. For training, they used exclusively Java Grande. For testing, they used SPECJvm98, SPECJvm2008, and DaCapo. Kulkarni and Cavazos achieved better runtime performance across all benchmarks. Furthermore, by using a ANN, they were able to achieve increased performance with *fewer* optimizations applied than -O3, on average 6 fewer optimizations. In the adaptive compiler, Kulkarni and Cavazos achieved an average speed-up in runtime of 8% and 8.2% in the Optimizing Compiler. However, as mentioned previously, Kulkarni and Cavazos approach relied significantly on online optimization. This is different from my approach which is entirely offline.

An example of unsupervised machine learning that also seeks to perform function-specific phase-order optimizations is Design Space Exploration (DSE). Martins et al. use a clustering approach with three separate algorithms: Normalized Compression Distance, Neighbor Joining, and a novel ambiguity-clustering algorithm to achieve function-specific speed-up in the LLVM C Compiler and ReflectC Compiler [16]. Martins et al. found that their DSE approached had increased improvement over other common phase-order optimizing approaches including but not limited to Genetic Algorithm and random sample; the latter of which is similar to Random Iterative Optimization. On average, across the entire benchmark suite, Martins et al's. DSE approach secured an average performance improvement of 7% [16].

In this paper, my approaches differ from the approaches mentioned previously by treating each program solely as an objective function, with no online element in any of the machine learning algorithms. I examined three machine learning approaches for the flag selection problem and two machine learning approaches in an attempt to optimize the phase order selection in the GHC. The first approach in both problems is Random Iterative Optimization: a purely exploratory algorithm. Random Iterative Optimization is the equivalent to randomly selecting permutations of compiler optimizations and checking for improvement. The second algorithm used in both problems is more exploitative, meaning that they seek to use data collected in each iteration to further optimize the program [4]. In this paper, I use a supervised approach which relies on predictions when selecting new phase-order candidates called Bayesian Optimization for Compiler Auto-tuning, and a slight

modification called Bayesian Optimization for Compiler Phase-Order Auto-tuning. This modification allows the optimization of the phase-order instead of flag selection. The third approach used in the flag-selection problem is Flag Optimization using Genetic Algorithm. These three algorithms are discussed below.

Random Iterative Optimization (RIO)

The implementation of RIO used in this paper is provided by Chen et al. [6]. Chen et al's. main research question was whether an optimal flag set for a specific program kept its "optimal-ability" when executed on different datasets from the MiBench benchmark suite. As a part of their RIO, Chen et al. also included compiler selection in addition to flag-selection as part of the iterative process. In their experiment, they allow either the use of the GCC compiler or the ICC C compiler when compiling the test programs, this is obviously not feature that I have implemented in the RIO used in this thesis. They found, on average, the use of ICC to compile C programs was faster, but that the GCC allowed for the maximum possible speed-up. They found that a program-specific optimal flag set was still optimal even on larger, more intensive datasets [6]. Chen et al. found for the experimented programs, a 375% speedup could be achieved relative to -O3 using their approach in the GCC compiler. For 39% of datasets, a speed-up of at least 10% was achieved, and for over 50% of the dataset, a speed-up of at least 5% was achieved. Further speed-up was impacted by compiler choice. Furthermore, they re-affirmed what I have already described, that certain interactions between flags can cause performance issues [6].

Flag Optimization with Genetic Algorithm

Genetic Algorithms (GA) are an unsupervised machine learning approach [4]. As mentioned previously, GAs are inspired by biology. I used a specific GA implementation: FOGA. Flag Optimization with Genetic Algorithm (FOGA) is an optimization algorithm applied to the GCC compiler. FOGA is unlike other GAs as it does selection first instead of last in a cycle [21]. The algorithm for FOGA can be seen in Figure 3.2. Tagtekin et al's. approach to GA involves (a). implementing a GA and (b). find the best parameters for a GA using the OpenTuner tool [3]. The parameters explored by OpenTuner for crossover, mutation, and selection are as follows: *{One Point, Two Point, Uni- form, Shuffle, Segment}*, *{Gauss by Center, Uniform}*, and *{Fully Random, Roulette, Sigma Scaling, Ranking, Linear Ranking, Tournament}*. Segment-based crossover was determined to be the best crossover algorithm. Gauss-by-Center was found to be the best mutation algorithm, and Linear

Ranking was found to be the best selection algorithm.

Post-tuning, FOGA converged on an optimal set quicker than OpenTuner's implementation of Genetic Algorithm. They also found the optimal set had significantly better performance than -O2 or -O1 [23]. Tagtekin et al. found that the most impactful and important flags for speed-up weren't even included within -O1, -O2, or -O3 of the GCC compiler. Following a similar approach with a few alterations, a GA could also work to find an optimal set of flags in the GHC. A more in-depth explanation of how FOGA has been implemented in this thesis is found in the Methodology section.

Bayesian Optimization for Compiler Auto-tuning (BOCA)

Bayesian Optimization for Compiler Auto-tuning (BOCA) is a supervised machine learning approach to flag optimization based on Bayesian optimizations [5]. Bayesian optimization is the maximization of an objective function f using past results as an indicator of improvement. BOCA separates itself from contemporary methods by also using Random Forest (RF) as a selection strategy [15]. Random Forest is another supervised learning approach which uses many decision trees, which are vulnerable to over-fitting, that are then merged together for more accurate and complex results. BOCA tries to balance exploration and exploitation by first designating a list of "high-impact" optimizations, the "impact-fullness" of an optimization is determined by its Gini-importance. Gini-importance is the average gain purity when a decision tree inside the random forest uses a specific feature for splitting; the calculation for Gini-importance as it is used in BOCA appears in equation 3.2. BOCA also includes lesser optimizations as determined by a decay function. High-impact optimizations are then randomly paired with less optimizations to make new candidates for optimization. Candidates are evaluated for their Expected Improvement (EI). The candidate with the highest EI is then added to the training set [5]. Chen et al. found that BOCA out-performed other Bayesian optimizations for compiler auto-tuning, as well as RIO and GAs [5]. Also, unlike RIO and FOGA, BOCA has also been applied to the LLVM compiler with similar results [5].

My implementation of BOCA is explained further in Methodology. To use BOCA to auto-tune the compiler phase-order, I had to modify this algorithm. Modification is detailed in Algorithm 1.

2.3 Auto-tuning in the GHC

In the realm of compiler auto-tuning related to Haskell, there is little relevant work for compiler auto-tuning, though there is some.

Hollenbeck et al. explored the effectiveness of currently existing compiler optimizations, specifically in-lining [10]. Unsurprisingly, functional programming typically contains a lot of functions. In-lining is one of the most important optimizations for reducing the cost of functions and thus is a prime candidate for improvement. The GHC contains heuristics for when the compiler should in-line a function, Hollenbeck et al. refer to these heuristics as "magic numbers." Hollenbeck et al. describe the GHC's in-lining heuristics as insufficient and thus developers often use pragmas. Hollenbeck et al. were able to find a single parameter configuration to boost average performance across benchmark programs. They also introduced a new Haskell benchmark suite for testing these "magic numbers." Hollenbeck et al. found that through changing both the GHC in-liner and using pragmas, performance increases of up to 27% are possible [10]. While Hollenbeck et al. do use some machine learning when attempting to find an optimal configuration using a clustering approach, it is not the focus of their research.

Another approach to optimizing Haskell code is by not optimizing the compiler itself, but by optimizing compiler annotations in a specific Haskell program. However, as Wang et al. describe it, creating these annotations by hand is a "black art, known only to expert Haskell programmers." Instead, Wang et al. introduce Autobahn, an automated tool for determining the optimal amount of strictness when creating thunks [27]. Autobahn creates program-specific annotations and uses a GA, and uses the nofib benchmark suite [18], just like I do in the flag-selection process. On average Autobahn was able to improve program performance by 8.5%, sometimes reaching improvement as high as 89% [27]. Autobahn is similar to my work as it, for the most part, treats each application as an objective function and uses only metrics obtained by nofib after program evaluation, though they do use more than just runtime in their consideration.

Finally, there has been some GPU auto-tuning using Haskell, Vollmer et al. present a framework for auto-tuning GPU kernels, specifically CUDA [26]. Vollmer et al.'s research is only compiler auto-tuning adjacent, and the focus on Haskell is coincidental. Therefore, Vollmer's work is not relevant to what I seek to accomplish in optimizing the GHC.

Little research has been done into compiler auto-tuning in the GHC for flag-selection

or for phase-order optimization. Therefore it is ripe for exploration, which I seek to accomplish. The most fruitful approach is to apply successful machine-learning optimization algorithms used in the popular GCC and LLVM compilers, where most research into optimization is done, and apply them to the GHC. The flag-selection problem is the more researched topic of the two, and thus has the most approaches to offer. After implementing several machine learning algorithms for the flag-selection problem, the algorithms can be adapted with some modification for the phase-order problem as well.

GHC is a prime candidate because of several similarities to C-compilers. First, the GHC can compile Haskell code *to* C using LLVM. The GHC also includes many of the same compiler optimization flags included within GCC and also uses the same system of flag presets, but GHC only has -O0, -O1, and -O2. Furthermore, GHC has fewer optimization flags than GCC; a smaller search space will make converging on an optimal flag set more likely. Some optimization work has been done in the GHC already with annotations, showing that there is definitely room for improvement in the compiler. Lastly, I have chosen a selection of both supervised and unsupervised learning algorithms, and a purely exploratory one (RIO). These algorithms were selected because they achieved great speed-up in previously done flag-selection work in the GCC.

Chapter 3

THE FLAG-SELECTION PROBLEM

3.1 Methodology of Flag Selection

I implemented three machine learning algorithms to explore and exploit flag selection in the GHC.

Random Iterative Optimization

Random Iterative Optimization (RIO) is an exploratory algorithm for compiler optimization. To call RIO a machine learning algorithm is being generous. RIO is, at its core, randomly choosing sets of flags and then checking if that set improved the runtime or not. Each set of flags tested by RIO is independent of whatever the runtime of the previous set was. The strengths of RIO come it its simplicity to implement and its efficiency. RIO makes it easy for me to explore a section of the search space in-order to determine if changing compiler flags actually has a statistically significant impact on the runtime of a test program.

The algorithm for RIO is simple and is implemented as described in its original paper by Chen et al. [6]: Let F be the set of compiler optimization flags and let I be the number of iterations, RIO works by first choosing a random number, r between 1 and $size(F)$, afterwards random uniform sets, $\{f_0, f_1, \ldots, f_{I-1}\}$ of r optimization flags are selected for experimentation. Then, for each f, a chosen test program is ran on using f_i as the only optimizations performed by the compiler. This continues until all sets in $\{f_0, f_1, \ldots, f_{I-1}\}$ have been performed. The run time of each program can then be used in statistical analysis to determine if a program is a good candidate for optimization, as was done in my experiments. Some of the tests programs that RIO showed good promise towards can be seen in Experimental Set-up.

Use of RIO in Program Selection

RIO falls into the exploratory realm of optimization [4]; in our paper, I used RIO on a selection of applications offered in the nofib benchmark suite [18], then used analysis of variance (ANOVA) to choose promising candidates for the optimization algorithms implemented later in this paper. Generally, if an application has a large distribution in run-time after RIO with large impact from individual flag changes, it

is a good candidate for optimization.

Flag Optimization with Genetic Algorithm

Genetic Algorithms (GA) are an unsupervised machine learning approach [4]. GAs are inspired by biology. In optimization, GAs represent flag sets as chromosomes and perform biology-inspired operations such as selection, crossover (the most expensive part of a GA), and mutation [20]. Flag Optimization with Genetic Algorithm (FOGA) is an optimization algorithm applied to the GCC compiler. FOGA does selection on its first iteration. Typically in a GA, selection does not occur until after the first iteration [21]. The algorithm for FOGA can be seen in Figure 3.2. Following a similar approach with a few alterations, a GA could also work to find an optimal set of flags in the GHC.

After a new set of chromosomes are ran against the test suite, the best performing ones are set aside and are guaranteed to be in the next generation. As a result, neither selection nor crossover nor mutation is performed on these "elite" chromosomes. All "non-elite" chromosomes are subjected to all the genetic operations in the following order: selection, crossover, mutation.

In this paper, selection is through linear selection; chromosomes are sorted by fitness; the highest performing flag sets are ranked first. As rank decreases, the chance of being selected also decreases corresponding to equation 3.1:

$$P_i = \frac{1}{N}(n^- + (n^+ - n + -)\frac{i-1}{N-1}; i \in \{1, \ldots, N\} \tag{3.1}$$

Where N is the number of selected chromosomes, n^+ is the high-ranked selection weight, n^- is the low-ranked selection weight, and P_i is the selection probability of each chromosome [21].

In this paper, gene crossover is performed using segment-based crossover. The gene sequence of compiler flags is broken up into segments. When chromosomes are crossed-over, a bit mask is used to determine if the offspring gets the segment belonging to the first parent (the more fit parent) or the second parent (the less fit parent) through a process called segment-based crossover [2]. A 0 denotes that the first parent's segment should be used, and a 1 denotes that the second parent's segment should be used; however, the 0s and 1s inside the bit mask are not equally weighted. The bit mask has a crossover probability that is weighted towards the more fit parent. Finally, the newly created offspring replaces the less-fit parent. This

ensures that the population of chromosomes remains constant. A demonstration of segment-based crossover is seen in figure 3.1.

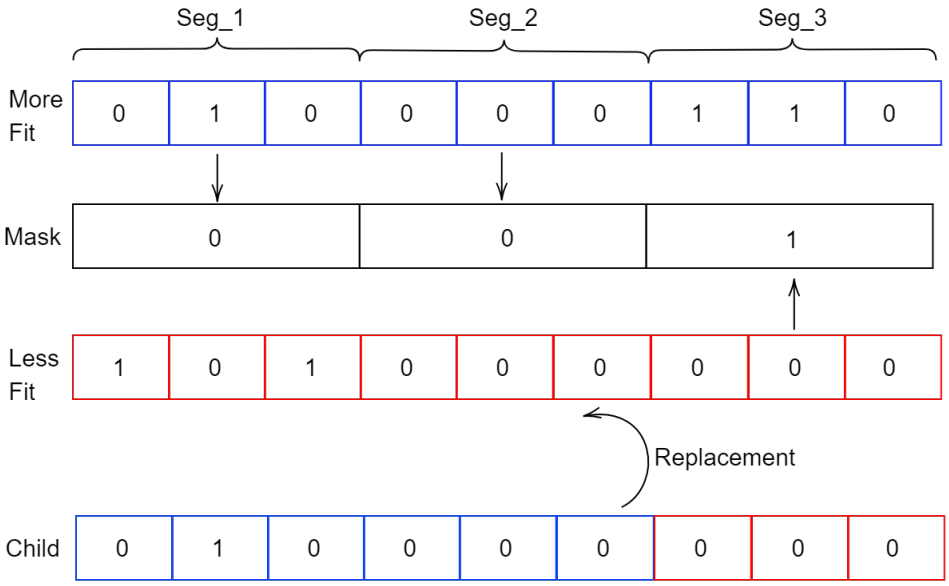


Figure 3.1: Segment-based Crossover

Mutation is performed using a bit-mask. The use of a bit-mask in mutation is a departure from the mutation method used in FOGA, which uses Gauss-by-Center [23]. This alteration was made because a bit-mask is a classic approach to mutation in GA and makes the most sense when dealing with a binary field such as compiler flags (ON/OFF)

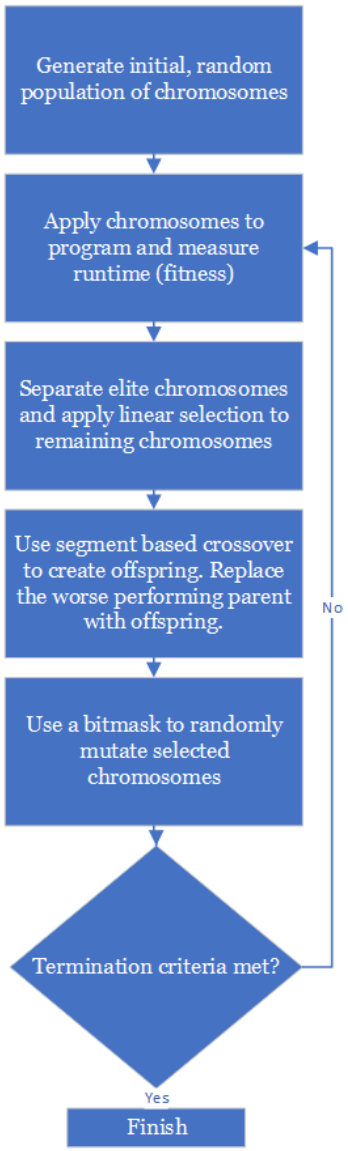


Figure 3.2: Our implementation of FOGA for the GHC

Bayesian Optimization for Compiler Auto-tuning

Bayesian Optimization for Compiler Auto-tuning (BOCA) is a supervised machine learning approach to flag optimization based on Bayesian optimizations. This algorithm is implemented as described in its original paper [5]. Bayesian optimization is the maximization of an objective function f using past results as an indicator of improvement. A diagram of BOCA is seen in figure 3.4. Each major step in the BOCA model has been given its own section.

Generating the Initial Training Set

BOCA generates its initial training set in the same way that FOGA generates its initial population.

Random Forest Regressor

BOCA separates itself from contemporary methods by also using Random Forest (RF) as a selection strategy [15]. Random Forest is another supervised learning approach which uses many decision trees, which are vulnerable to over-fitting, that are then merged together for more accurate and complex results. In each iteration of BOCA, the Random Forest is trained with the entirety of the training set. For this I used the Random Forest Regressor from the popular machine-learning Python libary, Scikit-Learn [19]. There are two types of Random Forest models: classifiers and regressors. Classifiers predict categorical classes of data (E.g ON/OFF). A regressor, which I use here, predicts continuous values, in this case, runtime.

Impactful & Unimpactful Optimization Flags

BOCA tries to balance exploration and exploitation by first designating a list of "high-impact" optimizations, the "impact-fullness" of an optimization is determined by its Gini-importance. In Random Forest, the impact of a flag, f can be found by summing its Gini-importance in every decision tree in the random forest:

$$Impact_f = \frac{\sum_{n=1}^{N} G(t_n)}{N} \tag{3.2}$$

Where N is the number of decision trees in the random forest, t is n^{th} decision tree that feature f was used to split, and G is Gini-importance. BOCA will select the highest rated optimizations for impact and mark them as "impactful optimizations." All other optimizations are then added to a lesser, unimpactful optimization list.

New Candidate Creation

To create a new candidate, BOCA first selects all candidates currently in the training set that share at least one (1) impactful optimization in the impactful optimization list. Then, for each of these candidates, BOCA takes the candidates impactful optimizations and combines them with C optimizations from the unimpactful optimizations list to get a new candidate. How many lesser optimizations are added to

the new candidate is determined by a decay function:

$$C(i) = c_1 \cdot \exp(\frac{(-\max(0, (c_1 + i) - \text{offset}))^2}{2 \cdot \sigma^2}) \tag{3.3}$$

Where $C(i)$ is the number of lesser optimizations to include in on the i^{th} iteration, c_i is the size of the initial training set, and offset is how many iterations to do before decreasing C. σ is defined as,

$$\sigma = \frac{-\text{scale}^2}{2 \cdot \log(\text{decay})} \tag{3.4}$$

The impact of offset, scale, and decay is seen in the plot of this decay function seen in figure 3.3. The parameters that generate this specific decay are the ones shown in the BOCA settings table, table 4.3.

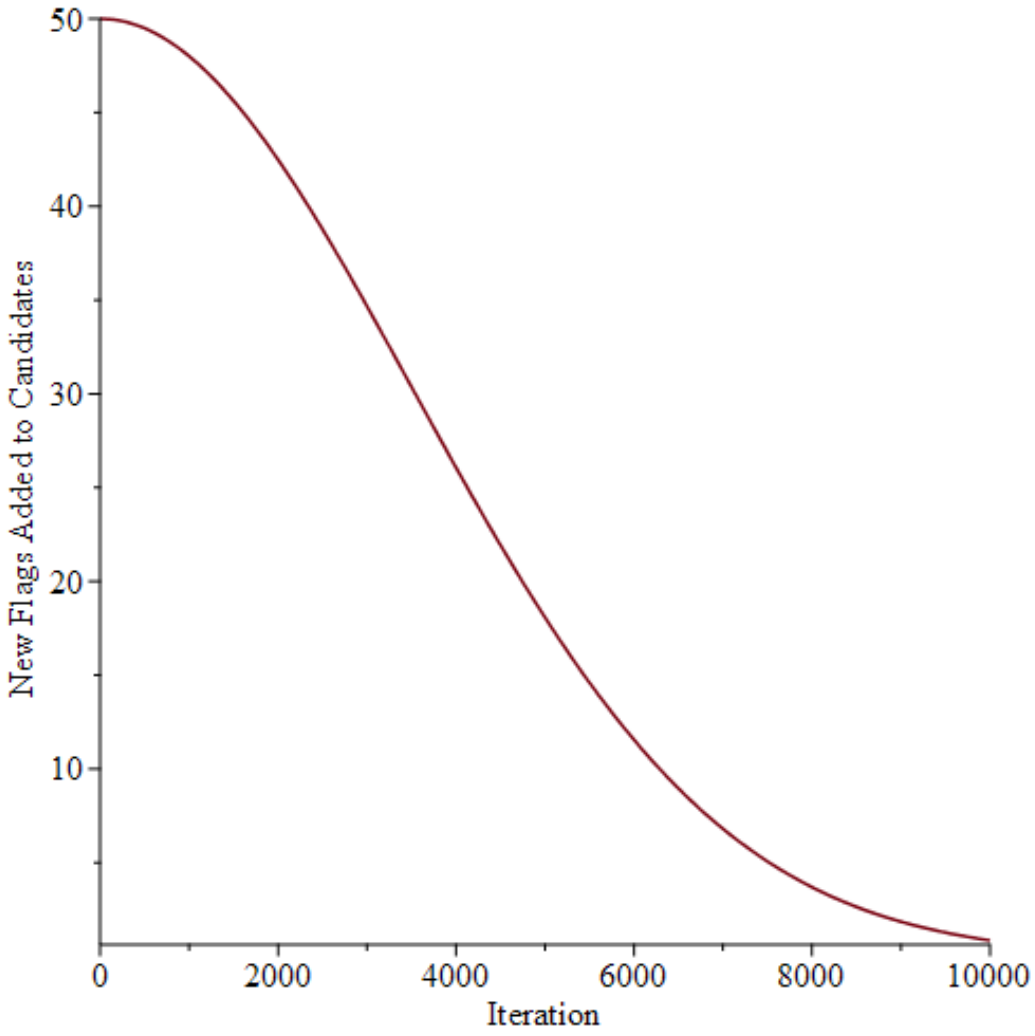


Figure 3.3: Decay function for BOCA as seen in Experimental Setup

While the number of candidates per iteration will increase as the training set increases in size, the variability of each new candidate compared to its progenitor will decrease as C decreases.

Expected Improvement

Candidates are then evaluated for their Expected Improvement (EI). The candidate with the highest EI is then added to the training set. Expected Improvement works by using the Random Forest Regressor to predict the runtime of new candidates, and then comparing those values to the current best flag set. A diagram that describes BOCA is seen in figure 3.4.

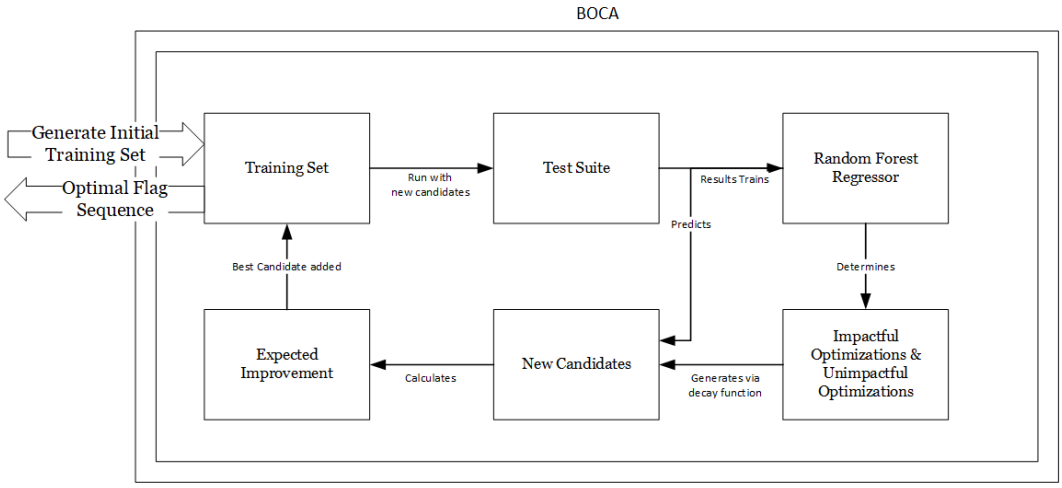


Figure 3.4: BOCA

3.2 Experimental Set-Up

I used the nofib bench-marking suite. I used nofib not only for its selection of "real-world" Haskell applications, but also to take advantage of its analysis tools. All the test programs were ran using the fastest dataset available in nofib: the “fast” setting. The programs I selected from nofib were the following:

- maillist- Mailing list generator
- cacheprof - An assembly code annotator for GCC
- hidden - Removes hidden lines
- sorting - Implementation of sorting algorithms

Setting	Value
Runs/program	5
Max # of calls to the benchmark/program	10000

Table 3.1: Global settings for machine learning models

I chose these programs because they seemed, compared to the other programs in the nofib suite, more sensitive to individual flag changes. I reached this conclusion after performing an analysis of variance (ANOVA) on their run-times based on each flag. However, it should be noted that not all programs in nofib were tested. Some tests in nofib required dependencies that, at the time, were not updated; these tests failed to compiled and as a consequence made poor test programs. Other tests, for example the n-body problem calculation test program found in *"shootout/n-body"*, simply took far too long to run to be considered practical for this experiment. All the test programs used compiled and ran correctly. Table 4.2 shows the global settings for each machine learning model. Despite being widely used, there has been discussion that the nofib test suite is outdated [10]. This might be the case for some experiments, but in my research, I was still able to achieve speed-up amongst its programs, and thus it still has a meaningful use here.

For RIO, while each individual run only had 500 sets, the model was used 10 times to achieve a total of 5000 flag sets. The settings for the other models are shown in tables 3.2 and 4.3.

Setting	Value
Max iterations	120
Population size	150
Mutation probability	0.287
Elitism ratio	0.147
Crossover probability	0.120
Number of segments during crossover	3
Max iterations without improvement	48

Table 3.2: Settings for FOGA.

For these models, the only performance metrics are runtime and benchmark calls (How many times the test program was executed before the machine learning model terminated). I made the assumption that the differences in execution times between the RIO, FOGA, and BOCA algorithms are largely trivial, and therefore our largest cost is calling the benchmark. Other metrics such as compile time, memory allocation, file size, etc. . . are not considered in my results or discussion.

Setting	Value
Initial set size	50
Number of impactful optimizations	5
Max iterations	10000
Max Without Improvement	2500
Decay	0.70
Offset	50
Scale	50

Table 3.3: Settings for BOCA.

Program	Model	Optimal Preset (s)	Runtime vs. O2
Maillist	BOCA	0.548	-15.17%
	GA	0.520	-2.804%
	RIO	0.520	-1.328%
Hidden	BOCA	0.471	-2.282%
	GA	0.502	-3.095%
	RIO	0.506	-1.556%
Cacheprof	BOCA	0.336	-2.041%
	GA	0.431	-1.373%
	RIO	0.431	-1.822%
Sorting	BOCA	0.204	-1.449%
	GA	0.207	-1.896%
	RIO	0.207	-0.481%

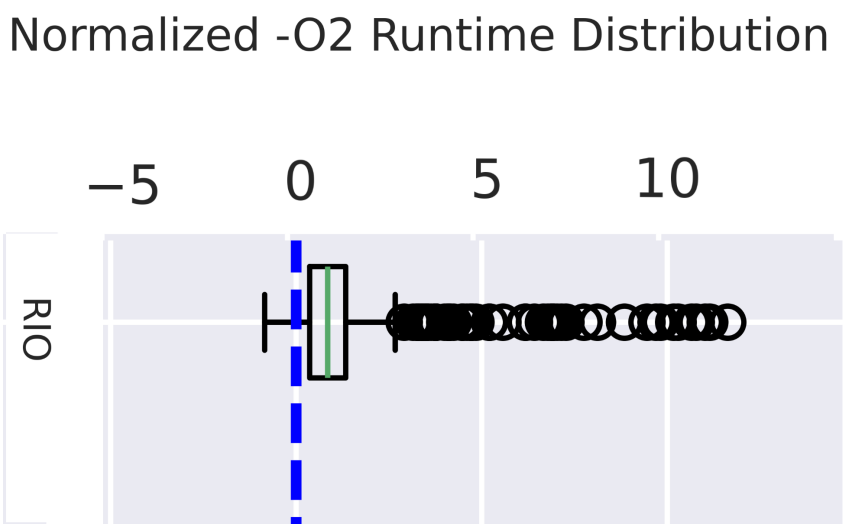
Table 3.4: Improvement over -O0 and -O2 across each program

3.3 Results

Runtime improvement over -O2 and -O0 was achieved by all models for all test programs. The lowest speed-up gained was a 0.481% speed-up over -O2, and the largest speed-up was an amazing 15.17% speed-up over -O2. Table 3.4 shows the run-times of each model across all programs and the speed-up achieved. On average, BOCA had the greatest increase in performance vs. -O2 compared to the other models; however, the impressive 15.17% speed-up accounts for that. If that data point is removed, FOGA narrowly beats BOCA in having the greatest increase in performance vs. -O2. Figure 3.6 as shown on page 5 shows the distribution in run-time per model per program as well as the run-time for -O0 as represented by the dashed blue line. Each model was normalized around its -O2 run-time. A good result would be one that, first, performed better than -O0 and -O2, and, second, had few negatively performing outliers and had most of the flag sets included in its IQR perform better than -O2. Figure 3.5 shows the total "allowance" used by each model for each program. Since allowance is the my prime measure of algorithm cost, this

should be as low as possible. The worst possible result would be to use up the entire allowance. The results, broken down by model, are explained further in this section.

RIO Results



RIO achieved an average decrease of 1.31% vs. -O2, the lowest speed-up achieved across all models. The above figure is a selection from Figure 3.6. It represents the normalized run-time of each set of flags around the -O2 run-time. RIO has a great number of outliers, especially low performing outliers. RIO has the greatest number of low performing outliers present in its distribution among every test program as seen in Figure 3.6. Also seen in 3.6, while RIO was able to find a more optimal flag set in all of the test programs, the average run time of all candidates was below the run time achieved by -O2 in all test programs. In some test programs, the average achieved by RIO was below the -O0 test result. As for how much of the benchmark RIO used: of the 10,000 benchmark runs allocated to RIO in its allowance, RIO consumed all 10,000 of it as seen in Figure 3.5.

FOGA Results

FOGA achieved an average decrease of 2.29% vs. -O2. FOGA the least number of outliers of the three machine learning models. FOGA has no outliers in Hidden, and only has a single outlier in Sorting and Cacheprof as shown in Figure 3.6. In half of the test programs, the average run time of all chromosomes in the population either met or exceeded performance of -O2. In the remaining two tests of Maillist and Cacheprof, the average speed-up achieved by FOGA was almost equal to the runtime of -O2 in Maillist, and was about one (1) standard deviation away from -O2

in Cacheprof. Of the 10,000 benchmark runs allocated to FOGA in its allowance, FOGA consumed between 3200 and 3300 benchmark runs as seen in Figure 3.5. The average amount of benchmark runs consumed was 3240.

BOCA Results

BOCA achieved an average decrease of 1.99% vs. -O2. BOCA also had many outliers; more than FOGA. However, BOCA had less lower performing outliers than RIO as shown in Figure 3.6. In half of the test programs, the average run time of all candidates in BOCA either met or exceeded performance of -O2. In Cacheprof and Sorting, the average speed-up achieved by BOCA was slightly less than that of -O2. Of the 10,000 benchmark runs allocated to BOCA in its allowance, BOGA consumed exactly 2568 for each of its benchmark runs, the lowest amount of runs used across all models as seen in Figure 3.5.

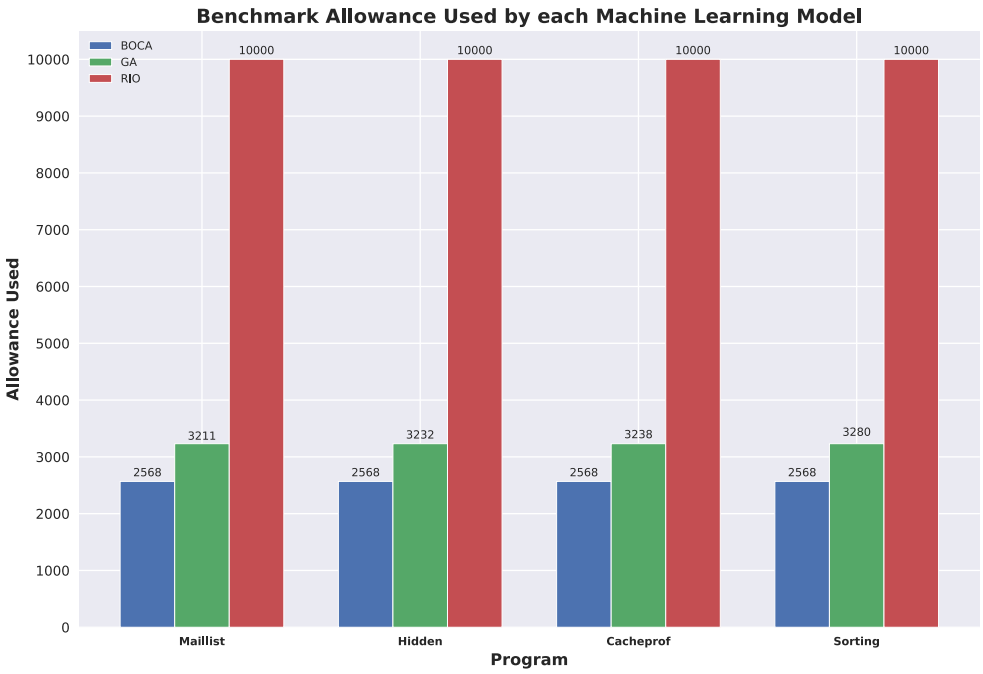


Figure 3.5: Allowance left remaining once model finished running

3.4 Discussion

As mentioned, there are significant differences in the numbers of outliers present in Figure 3.6 between FOGA, BOCA, and RIO. These results are entirely expected. RIO is, in-effect, simply guessing if a random combination of flags will create

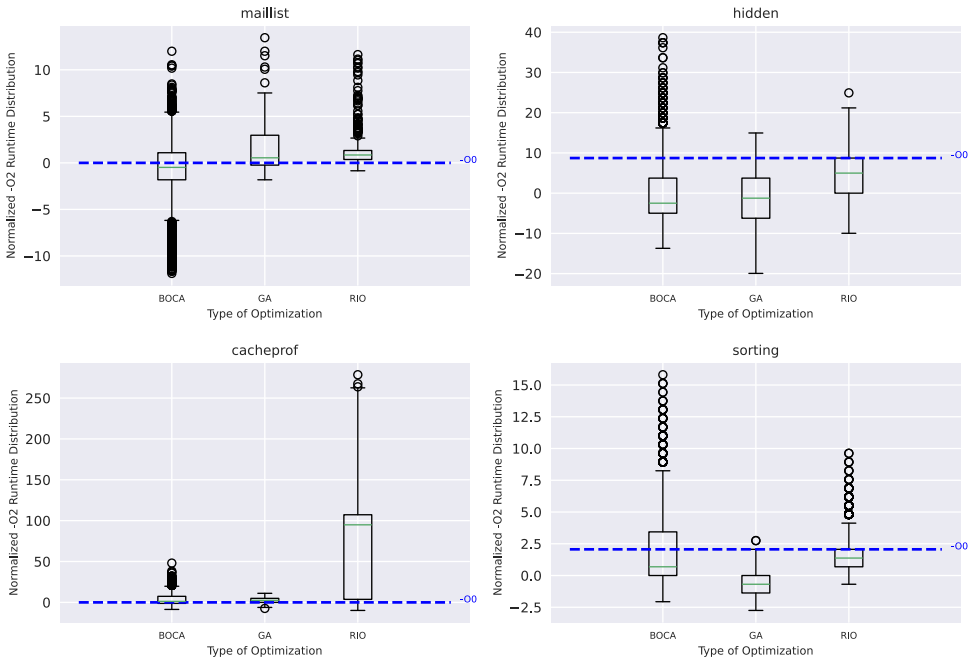


Figure 3.6: Distribution of flag run-time normalized around -O2 run-time

an improvement. Therefore, I expected a large variation in the quality of the sets selected. It is a similar story with BOCA. BOCA generates an initial training set of random flags and these sets remain in the training set through BOCA's entire execution. If an outlier exist within the initial training set, it will persist through the entire set. However, since BOCA is supposed to converge on a correct solution, there should be more lower performing outliers than higher performing outliers since the average improvement should trend upward. The exception is FOGA. FOGA attempts to remove the worst performing chromosomes in every iteration, therefore decreasing the number of poor performing outliers. However, sometimes a bad chromosome just gets lucky and is not selected for removal.

However, since RIO generates a large number of outliers and does not converge on a solution, one would expect an equal number of negative and positive outliers as well; however, that is not the case. All outliers were worse performing than -O2, some upwards of 8 standard deviations. In-fact, many of these outliers perform worse than -O0. This suggest that there are many detrimental optimizations present in Haskell, more so than beneficial ones. The most sensitive program to outliers was Maillist. Maillist has more outliers than any other program and seems to be

especially sensitive to change.

I think there is a potential for even greater gains among Haskell, but perhaps a few detrimental optimizations could tank the run-time of an otherwise optimal set.

In Figure 3.7, I plot the number of shared flags between BOCA and FOGA. Since BOCA and FOGA are expected to converge on a solution, I figured they would converge on a similar or the same solution. However, that does not appear to be accurate. BOCA, in all cases, used more flags than FOGA. BOCA and FOGA shared, in most test cases, about half of their flags, so FOGA was not just generating subsets of BOCA. The only exception is Maillist, where FOGA actually *was* a subset of of the flags that BOCA used. This was also the test program where BOCA achieved its incredible 15% speed-up over -O2.

Comparing my results to the implementations of FOGA [23], RIO [6], and BOCA [5] in the GCC, my results are less impressive. While direct comparisons to -O2 are not present in RIO or BOCA, FOGA achieved a significant improvement over -O2, in some instances upwards of 50% improvement. Similarly, RIO achieved an average improvement of 20% against -O3 in GCC [6]. Meanwhile, BOCA achieved faster convergence and better improvements than RIO or GA in most of their experimental results [5].

The reason for my overall worse performance could be any number of things. As mentioned, the GHC may have more or worse "detrimental" optimizations than GCC. The -O2 preset in GHC may be much more universal to improvement than -O2 in GCC. The programs tested may not have been as conductive to beneficial optimization as originally believed, or GHC has fewer beneficial optimizations than GCC.

Considering run-time, FOGA was the best performing algorithm when the 15% improvement outlier from BOCA is removed. However, FOGA also took almost 1000 more benchmarks runs than BOCA. RIO, the simplest algorithm to implement, took almost triple the number of benchmark runs as FOGA. In a trade-off of less time for slightly less performance, BOCA is the best approach to optimizing programs.

Lastly, using an MCA analysis, I was able to identify several flags that are responsible for a large amount in the variance between the runtimes of different sets of flags. The MCA component plot is shown in figure 3.8. Component 1 accounts for over 70% of the variance and including Component 2 only raises the total variance covered to 75% roughly. The top six (6) optimizations responsible for the variance and their

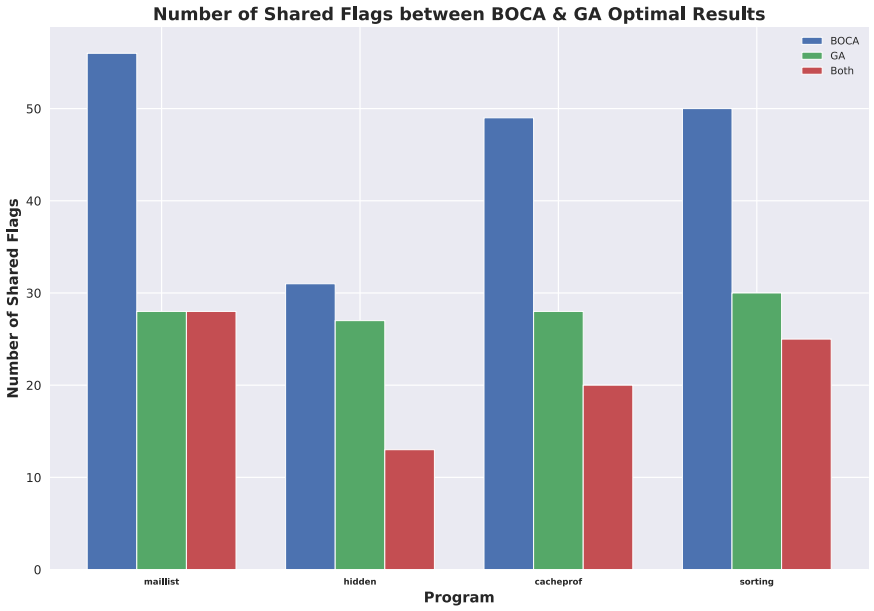


Figure 3.7: Number of shared flags between BOCA and FOGA optimal flag sets

impacts on the source code [24] are, in-order:

1. **-fcmm-sink** - enables the sinking pass
2. -fregs-iterative - “Use the iterative coalescing graph colouring register allocator for register allocation in the native code generator”
3. -fno-pre-inlining - disables pre-inlining
4. **-fliberate-case-threshold=2000** - Sets the size threshold for the case liberation optimization
5. -feager-blackholing - “black hole” thunks immediately
6. **-fspecialise** - specialises type-class-overloaded functions

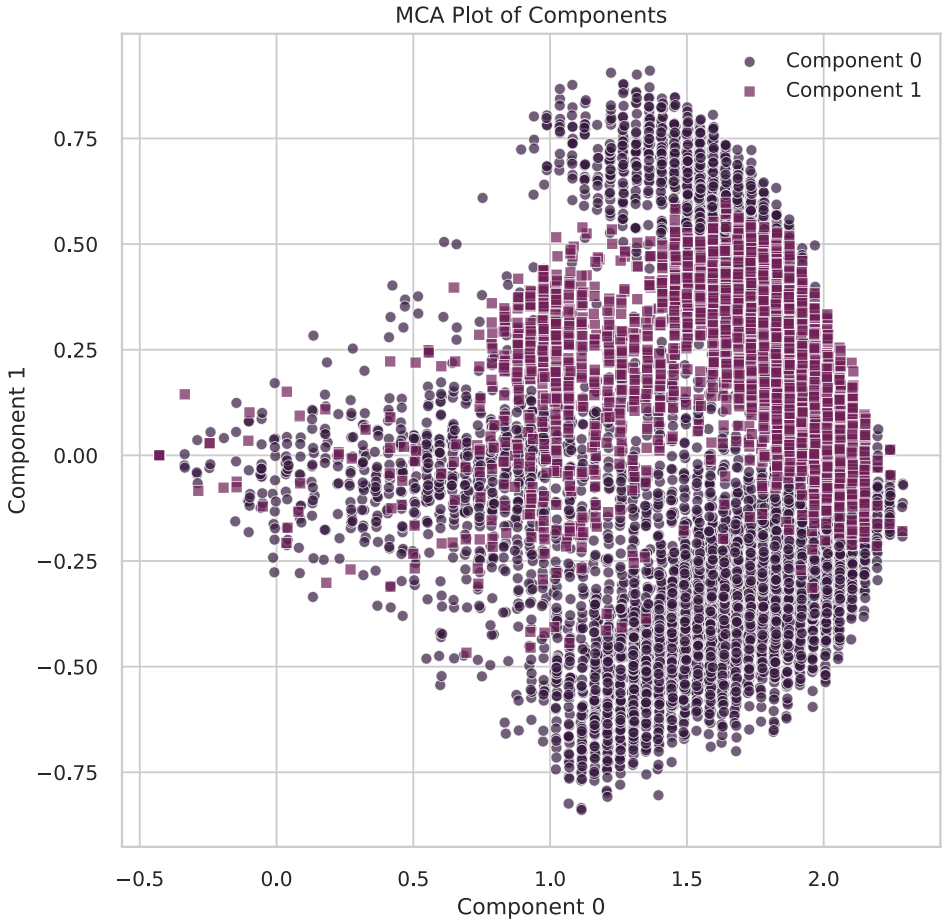


Figure 3.8: Two Component MCA Plot. Data combined from all three models

The **bolded** optimizations are the optimizations that are included in at least -O1, which means they are also included in -O2. These six optimizations flags are most likely the most impactful ones, generally, for any program. This analysis does not determine which of these optimizations are generally good or bad, just the ones that cause the most change. For example, pre-inlining is such a crucial optimization that it is performed even when -O0 is passed in, so it makes intuitive sense why disabling it accounts for a large amount of change. Also noticeable is that half of the most impactful optimizations are included in -O2, while the other half are not.

3.5 Conclusions - Flag Selection

Compiler optimization is an area ripe for exploration in the GHC, and, based on my results, is a fruitful endeavor. For most Haskell programs, there exist a set of flags

that are more efficient than the highest optimization level provided by GHC, -O2, and machine learning models such as BOCA or FOGA are apt at identifying several possible optimal candidates.

Of the three models tested, RIO, FOGA, and BOCA, FOGA had the largest improvement in run-time vs. -O2; however, it came at the expense of a 20% increase in the number of iterations required when compared to BOCA. This would be inconsequential if the improvement of FOGA over BOCA was also 20%, but that is not the case. The improvement in the average speed-up of BOCA (Not including Maillist) vs. FOGA is only 15%. Therefore, BOCA offers the most balanced approach between raw speed-up and the time required to obtain the speed-up.

However, while all three models, RIO, FOGA, and BOCA, achieved speed-up in the GHC, all achieved results less impressive than their original papers, some by an order of magnitude. This was unexpected and shows that the GHC may have less room for compiler auto-tuning than the C compilers. GHC may also have many potentially disastrous "optimizations" for a given program that dramatically increases the run-time. This is an area that warrants further research. So while Haskell programs may benefit from using machine learning models to find optimal flag sets, they do not benefit to the extent that C programs benefit.

Chapter 4

THE PHASE-ORDER PROBLEM

4.1 Methodology of Phase-Order

Most of the related work done in machine learning phase-order selection has relied heavily on online optimizations and has focused on function-level specific phase-order improvements. My approach differs in that it is entirely offline, requiring no runtime statistics of our "objective function" besides the finished application runtime. This approach makes phase-order optimization more accessible. I modified established algorithms for compiler flag selection and used them to solve the phase-order problem.

Bayesian Optimization for Compiler Phase-Order Auto-Tuning

The main machine learning approach I used to work on the phase-order problem is BOCA. BOCA, however, was not developed to solve this problem, but instead the similar problem of compiler flag selection. As a consequence, the algorithm for BOCA that I implemented to optimize the flag selection problem cannot be used to solve the phase-order problem, at least not without major modifications, which I have performed. The source of the incompatibility between BOCA for flag selection and BOCA for phase-order optimization is that flag selection is fundamentally a combination problem, while phase-order is a permutation problem; BOCA must be modified in its candidate creation stage to account for this more complex relationship between different optimizations. For clarity, let us call this modification Bayesian Optimization for Compiler Phase-Order Auto-Tuning (BOCPA). The modified algorithm is shown in Algorithm 1. This algorithm is very similar to the original build, but I will discuss key changes made.

First and foremost, while BOCA required a list of all optimizations, BOCPA requires a list of all possible rules. A rule in BOCPA is in the form (A, B) and denotes *"Optimization A must come before Optimization B"*. BOCA, when creating the starting set of candidates, randomly selects flags to be on. This does not work for the phase-order problem. Each optimization in the optimization pipeline can be represented as a vertex in a graph. For example, a theoretical phase-order permutation could be be $\{B, C, A, D, E\}$. Each rule, for example, (B, C) is represented as an edge. This is shown in picture 4.1(a). All pictures of graphs were generated using the Tikz

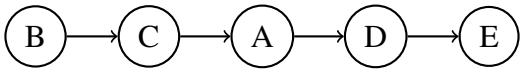
Algorithm 1 BOCA for Phase-Order (BOCPA)

Require: O: a list of possible phase-order rules $[o_i | i \in 1 \ldots m]$
κ: the number of impactful rules
$size_{initial}$: the size of the initial training set
Ensure: $T_{candidates}$: (a table of all candidates along with their execution time)

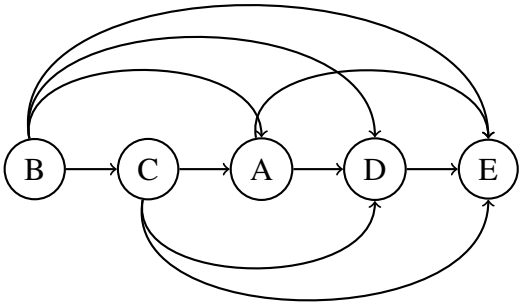
```
1: train ← [];
2: for all i ← 1, size_initial do
3:    for all j ← 1, m do
4:       s[j] ← randomRules()
5:    end for
6:    train.add(s, f(s))
7: end for
8: for all i ← 1, n do
9:    model ← RandomForest(train)
10:   for all j ← 1, m do
11:      importance[j] ← getImportance(o_j, model)
12:   end for
13:   importantOpts ← getImportantRules(importance, O, κ)
14:   unimportantOpts ← set(O − importantOpts)
15:   importantCandidates ← getAllSettings(importantOpts)
16:   allCandidates ← []
17:   C ← normalDecay(i)
18:   for all j ← 1, size(importantCandidates) do
19:      G ← createGraph()
20:      G.addVertices(shuffle(O))
21:      G.addEdges(importantOpts ∩ importantCandidates[j])
22:      r_size ← size(importantCandidates[j]) − C
23:      randomRules ← randomSet(importantCandidates[j], r_size)
24:      G.addEdges(randomRules)
25:      newCandidate ← topologicalSort(G)
26:      allCandidates.add(newCandidate)
27:   end for
28:   for all j ← 1, size(allCandidates) do
29:      (mean, std) ← model.predict(allCandidates[j])
30:      ei[j] ← EI(mean, std)
31:   end for
32:   bestCandidate ← getBestCandidate(allCandidates, ei)
33:   train.add(bestCandidate, f(bestCandidate))
34: end for

35: T_candidates ← convertToTable(train)
36: return T_candidates =0
```

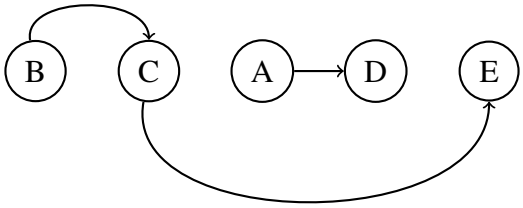
library [25].



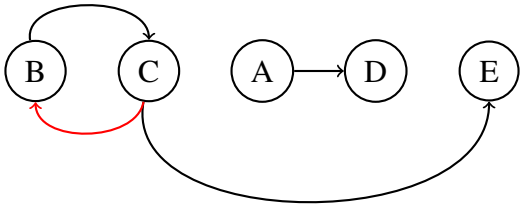
((a)) Original Permutation Rules



((b)) Theoretical Permutation Rules with All Required Rules



((c)) Incomplete candidate with only impactful rules



((d)) Candidate with contradictory rules

Figure 4.1: Creating a New Candidate in BOCPA

However, this graph is insufficient at representing all the rules required to make the permutation $\{B, C, A, D, E\}$. Rules also need to be added to show B's relation to A, D, and E. The same being true for C, A, and D. This updated graph is shown in picture 4.1(b).

To accomplish this, I made the first change on line 4 in algorithm 1. All optimizations must be included in the pipeline. To accomplish the same result, I randomly select a permutation of the optimization pipeline and then use that to generate choose the

initial set of rules. For example, if I select the random permutation, "B|M|A", I will generate the rules: [(B, M), (B, A), (M, A)]. The total number of rules required to form a permutation for a pipeline with n optimizations is $\sum_{k=1}^{n}(k-1)$ and the total number of possible rules is $\binom{n}{2} \cdot 2!$.

In BOCPA, the transition to graphs does not change how the algorithm determines the impactful optimizations. Each rule is still included inside the Random Forest that is used to train BOCPA, so each rule is assigned an impactfulness score as explained in equation 3.2. In BOCA, if an element in the training set contained an "impactful flag", then all of its "impactful flags" are used in the new candidate. This is the same for BOCPA. For example, when generating a new candidate from the permutation above, the impactful rules present could be $[(C, E), (A, D), (B, C)]$. This would be represented by constructing a new graph as seen in picture 4.1(c).

However, this is where I make the next major change to BOCA. In the original algorithm C as calculated in equation 3.3 is used as the upper bound of how many additional "unimportant flags" should be added to the new candidate. This does not work for BOCPA for two reasons. The first reason is that all candidates have the same number of rules and thus using C to determine how many additional rules to add is redundant. The second reason is that by just randomly choosing "unimpactful rules", there is the possibility of introducing contradiction into a candidate. To continue the example from above, imagine if a randomly selected "unimpactful rule" is (C, B). Then the constructed graph would look like picture 4.1(d). This graph contains a cycle, therefore we can not construct a meaningful permutation from it since *"B must come before C" and "C must come before B"*. As a consequence, merely grabbing *random* unimpactful rules will not work in most cases. It works for the flag-selection problem since there are no mutually exclusive optimizations flags, but it cannot work here where there are mutually exclusive rules.

To resolve this issue, I introduced a few changes to the candidate creation process. The first change is in how C is used. Instead of selecting a random set of "unimpactful rules" of size $[0 - C]$ from the total list of possible rules, BOCPA selects a random set of "unimpactful rules" of size $[0, size(\text{<progenitor's rules>}) - C]$ from its progenitor's set of rules. This achieves the same idea of BOCA since as iterations increase, each new candidate will be more and more similar to its progenitor. This mimics BOCA since, theoretically, BOCPA converges on a solution. The second change made to the candidate creation process is by topologically sorting the graph to get the remaining "unimpactful rules." This is not perfect, and is more

computationally expensive than the original BOCA. I believe this change still allow discrimination of good and bad performing rules respectively because there will still be plenty of variety between the candidates when the rest of the permutation order is generated further in the algorithm. This is because, even if only a few rules for a new candidate are selected, all optimizations must be included in a rule *somewhere* since the pipeline must always be the same length. One topological sort of graph in picture 4.1(c) is E, D, A, C, B; that is an acceptable permutation.

In summation, the most significant change in BOCPA over BOCA is how new candidates are constructed: all of the optimizations that are in the GHC optimization pipeline are represented as if they were vertices in a directed graph. The rules represent directed edges between those vertices. A cycle in the graph represents a contradiction within the rules of a phase-order candidate. Therefore, all graphs must be directed acyclic graphs (DAGs). When a candidate is being created from a progenitor; they have the same vertices. Then the impactful rules (edges) present in the progenitor are added to the new candidate. This is guaranteed to be a DAG since the impactful rules is just a subgraph of the progenitor. Then, a selection of additional edges from the progenitor is also added to the candidate. This also has to be a DAG. Finally, the candidate has to generate the rest of its edges to create a valid permutation. The easier way to generate a random phase-order from a incomplete list of rules is through a topological sorting. This can be done since, as mentioned, all the candidates are DAGS.

I use the NetworkX Python library for our graph operations [9]. However, I do not use NetworkX's topological sorting functions. NetworkX's algorithms are based on Kahn's Algorithm for topological sorting [12]. Topological sorts are not unique, and by changing the starting vertex of Kahn's algorithm, it is possible to influence which topological sort Kahn's algorithm returns. However, under the Kahn's algorithm used in NetworkX, not all possible topological sorts are present. Some are omitted, and therefore I elected to not use it. I, instead, use a probabilistic Kahn that introduces probability when selecting source nodes. All possible topological sorts are possible under my modified probabilistic Kahn's Algorithm [1], but not all topological sorts are equally weighted. The probability is biased towards some sorts more than others, but the important fact is that each possible topological sort has a non-zero probaility of getting selected, which distinguishes it from NetworkX's algorithms. This is discussed further in the "Probabilistic Kahn's" section. Algorithm 2 shows probabilistic Kahn's in detail.

Algorithm 2 Probabilistic Kahn's Algorithm

Require: $\mathcal{G}$: A directed acyclic graph (DAG)
Ensure: $topo\_sort$: A list representation of the topological sort of $\mathcal{G}$

```
topo_sort ← [];
while order(G) > 0 do
    source_nodes ← get_all_source_nodes(G)
    weights ← []
    for all s ← 0, size(source_nodes) do
        weights ← weights ∪ [1 + out_degree(source_nodes[s])]
    end for
    elem ← random(source_nodes, weights)
    topo_sort ← topo_sort ∪ [elem]
    G ← remove_node(G, elem)
end while
return topo_sort =0
```

Expected Improvement, as originally presented by Chen et al. [5], has been left untouched. I will show that this algorithm continues to be correct.

Probabilistic Kahn's

As briefly mentioned in Methodology, the Kahn's algorithm I used to topologically sort the graphs of new candidates in BOCPA is modified. I use a non-deterministic probabilistic Kahn's algorithm to ensure that each possible topological sort for a graph has a greater than zero chance of being selected. The topological sorting algorithm present in the NetworkX Python library [9] is deterministic, but will give different topological sorts depending on the order the vertices were added to the DiGraph. There is an issue with NetworkX's topological sorting function: even when trying every possible permutation of vertices, there are some possible topological sorts that are not returned by NetworkX's Kahn's, i.e their probability is 0%. While NetworkX gives an equal weight to all topological sorts with a probability greater than zero, it also neglects many possible sorts, meaning it is also biased. For example, let $\mathcal{G} = (V, E)$ such that $V = \{A, B, C\}$ and $E = \{(A, C)\}$. There are three possible topological sorts to $\mathcal{G}$: $\{A, BC\}$, $\{B, A, C\}$, and $\{A, C, B\}$. However, no matter the permutation of $\{A, B, C\}$ added to $\mathcal{G}$ when making a NetworkX DiGraph, the only two topological sorts returned by NetworkX's topological sorting function are $\{A, B, C\}$ and $\{B, A, C\}$. The third possible topological sort is ignored completely. Of the two that remain, half the of the possible permutations of vertices select one and half of the permutations of vertices select the other.

NetworkX also has the option to return all possible topological sorts of a graph. This also includes the ones that NetworkX's Kahn's skips! It would be most fair to generate all topological sorts and then select one at random, but this is extremely computationally expensive especially with large graphs. BOCPA already struggles with remaining efficient due to graph operations, and this would just make BOCPA completely impractical. Thus, a probabilistic approach is an acceptable approach; however, a better approach may exist.

The algorithm for probabilistic Kahn's is seen in Algorithm 2. An example of the bias of Kahn's is seen in Figure 4.3. Using the Law of Large Numbers, the true probabilities of selecting each topological sort of the graph depicted in Picture 4.2.

((a)) Step 1 ((b)) Step 2

Figure 4.2: Probabilistic Kahn's Example

The graph in picture 4.2(a) has four (4) possible topological sorts: *{A,B,C,D}, {A,C,B,D}, {A,C,D,B}, and {B,A,C,D}*. While the result is clearly biased towards one result, all results are represented. This bias is because Kahn's will selected a source vertex with a weighted probability, this weight is based on the minimum number of possible "steps" that will be remaining after adding the source node to the sort. In the case of 4.2(a), there are two source vertices in the graph: A and B. Initially, the probability of selecting B is $\frac{1}{3}$ since B has an out-degree of 0 and the probability of selecting A is $\frac{2}{3}$ because A has an out-degree of 1. So why is the topological sort that begins with B so prevalent if A has a higher probability? It is because B has only one possible topological sort if it is selected first, while A has several possible sorts if it is selected first. Suppose A is selected, then we perform the next step Kahn's Algorithm as normal; we erase the edges for vertex A and remove A from consideration, which is done in picture 4.2(b).

In the second step of the algorithm , the probabilities are, again, $\frac{1}{3}$ for B and $\frac{2}{3}$ for C, since C has an out-degree of 1. This continues until all edges have been erased from the graph. The order the vertices are removed represents the order of the topological sort. For more information, see algorithm 2.

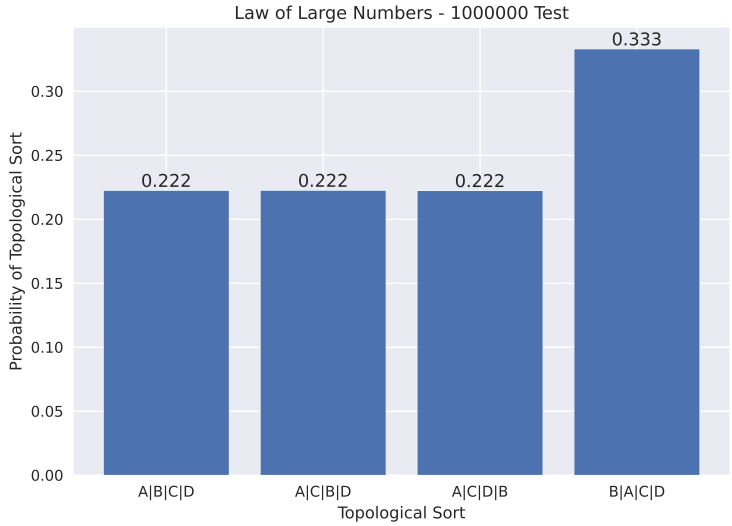


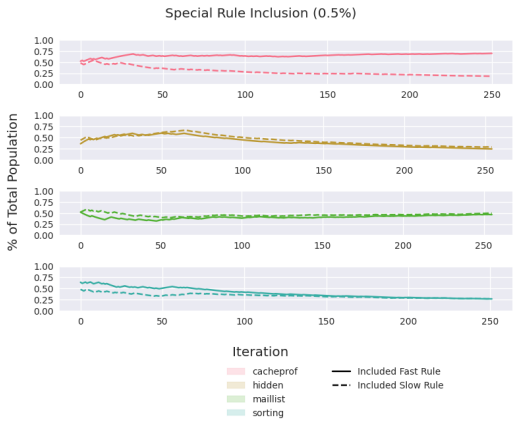
Figure 4.3: Probabilities of different topological sorts using probabilistic Kahn's on a simple graph

Evidence of Discrimination

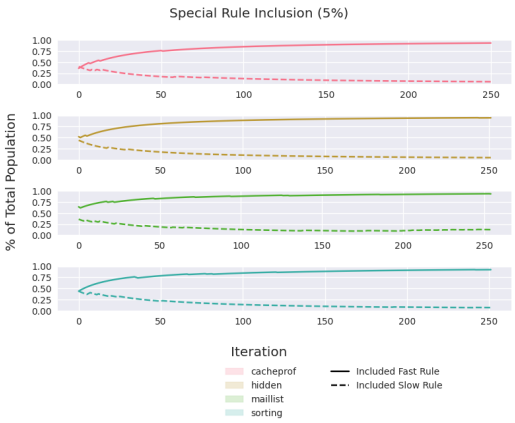
BOCPA should be able to discriminate between "good" rules and "bad rules." If a rule improves the runtime, any new candidates should include it, and if a rule decreases the runtime, candidates should avoid it. This discrimination occurs in the candidate creation stage and the EI stage. The EI stage has been left unchanged, but I must verify that my changes still allow candidates to be created properly, and that BOCPA is still able to discriminate between good and bad performing rules, just like it was able to discriminate between good and bad performing flags.

To provide evidence of discrimination, I manipulated the runtime data for a candidate if that candidate contained a specific, arbitrary test-rule. To not impact the accuracy of the performance of real rules, I added three (3) new "fake" passes to the optimization pipeline; these passes did nothing to modify the original source code. During candidate creation, through BOCPA as described above, a new candidate could be created with two new rules: *("good_optimization", "neutral_optimization")* or *("bad_optimization", "neutral_optimization")*. *("bad_optimization", "neutral_optimization")* was designated the "slow rule" and increased the runtime by a set percent amount of the true runtime result. *("good_optimization", "neutral_optimization")* was designated the "fast rule" and decreased the runtime by a set percent amount of the true runtime result. If BOCPA is working correctly, after the "slow-rule" was selected once, BOCPA should discriminate against it such that it would hardly ever

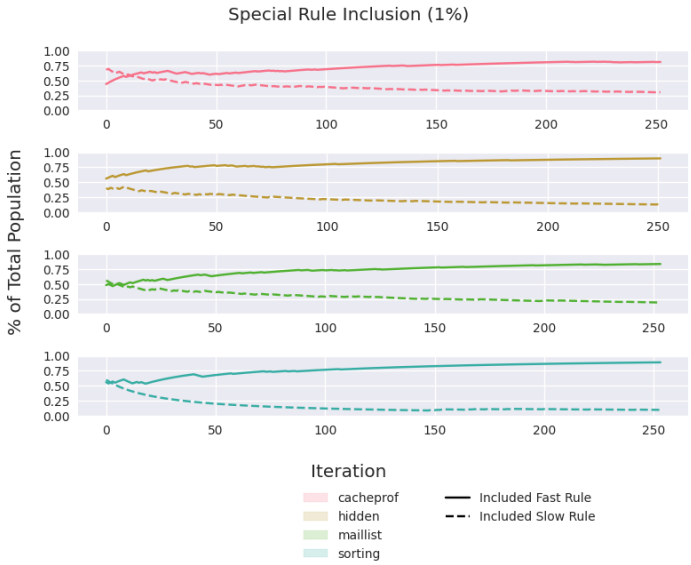
be selected again as the model learned that it significantly decreased performance. Likewise, the "fast-rule" should be frequently selected due to its significant increase in performance. The results for the fast rule and the slow rule are seen in Figure 4.4(c) using a 5% speed-up and slow-down for the slow and fast rules.



((a)) Distribution of Good and Bad rules over BOCPA iterations with a 0.5% Improvement Margin



((b)) Distribution of Good and Bad rules over BOCPA iterations with a 5% Improvement Margin



((c)) Distribution of Good and Bad rules over BOCPA iterations with a 1% Improvement Margin

Figure 4.4: Comparison of Distribution of Good and Bad Rules over BOCPA Iterations with Different Improvement Margins

As expected, BOCPA was able to discriminate between the fast, "good" rule and the slow, "bad" rule. When creating the initial set, there is an equal chance that an initial

candidate has the fast rule, the slow rule, or both rules. After encountering the fast rule, BOCPA identifies that the fast rule should be selected every time. Likewise, the model realizes the slow rule should never be selected. Sometimes, by random chance, both the fast rule and the slow rule get added. In this instance, the 5% increase in performance is countered by the 5% decrease in performance caused by the slow rule. All results are shown in figure 4.4. Results for smaller improvements than 5% can be see in figure 4.4(c). Below 1%, BOCPA can no longer reliably discriminate for the specific fast or slow rule as seen in figure 4.4(a).

Next I tested that BOCPA could still detect these arbitrary rules as impactful. In this test, the slow rule was removed completely and five (5) additional arbitrary fast rules were added. Each of these rules would decrease the runtime by 5%. The results of these tests are seen in Figures 4.5 and 4.6. Figure 4.5 shows that most of the arbitrary rules were added new candidates in each iteration during the candidate creation step of BOCPA, some variation is expected as BOCPA tries to explore for even more optimal orderings. Figure 4.6 similarly shows that the arbitrary rules were deemed impactful most of the time.

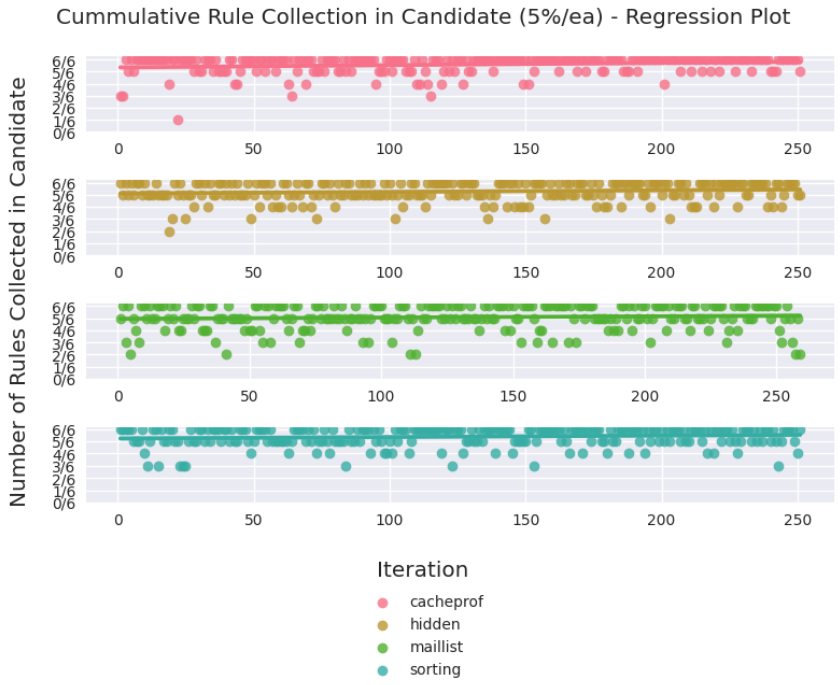


Figure 4.5: Number of arbitrary runtime improving rules included in each new candidate with regression line

Figure 4.6: Number of arbitrary runtime improving rules marked as impactful by BOCPA after each iteration

Eliminating Problem Orderings

I decided to limit changes in permutations to only the optimizations included in -O1 and -O2 (herein refereed to as "-O2 Optimizations" for sake of permutations).The reason for this decision comes from extensive testing; changing the phase-order of early optimizations was much more likely to cause GHC tests to fail than changing the phase-order of later optimizations. In-fact, not a single acceptable permutation has been found with re-ordered O0 optimizations. The specific reason for this has not been determined. It was hoped that since Haskell is a purely function programming language, changing the order of optimizations in the pipeline would not cause significant issues. It is; however, possible that there exists successful permutations within the O0 optimizations; however, I have not devised a method of specifically finding these illegal permutations. Instead, I have focused on the nine O2 optimizations, which still provides 9! possible permutations, still a significantly large search space to warrant this approach.

Even among the O2 optimizations, there still exists some "forbidden positions." To ensure that RIO and BOCPA both only output *working* optimal phase-orders, I added an additional machine learning model: a Support Vector Machine (SVM) [8] with a linear kernel function. I use SVMs for classification of the training set of BOCPA

Metric	Result
Accuracy	0.998
Precision	0.984
Recall	1.0

Table 4.1: Evaluation metrics of the SVM for classifying working and non-working permutations

and the randomly generated phase-orders of RIO into two (2) categories: works and does not work. For training, the SVM uses over 2000 successful phase-orderings. The metrics of the SVM are seen in table 4.1. When I ran each permutation through the compiler test suite, I ran each of them on the compilers fastest test speed: `hadrian/build test --test-speed=fast`.

Changes to GHC

The changes made to the Glasgow-Haskell Compiler are effective at changing the phase order. The changes start in the compiler driver in Main.hs. The driver is responsible for running the Core2Core simplifier. We have the driver read and parse an external file, *parse.txt*, with a very simple string inside: a list of numbers separated by the delimiter '|'. This input is then passed to main Optimization Pipeline in Pipeline.hs which is then passed to the function `core2do` which builds the pipeline which then uses this parsed string to re-order the optimizations. This process can be found in Figure 4.7

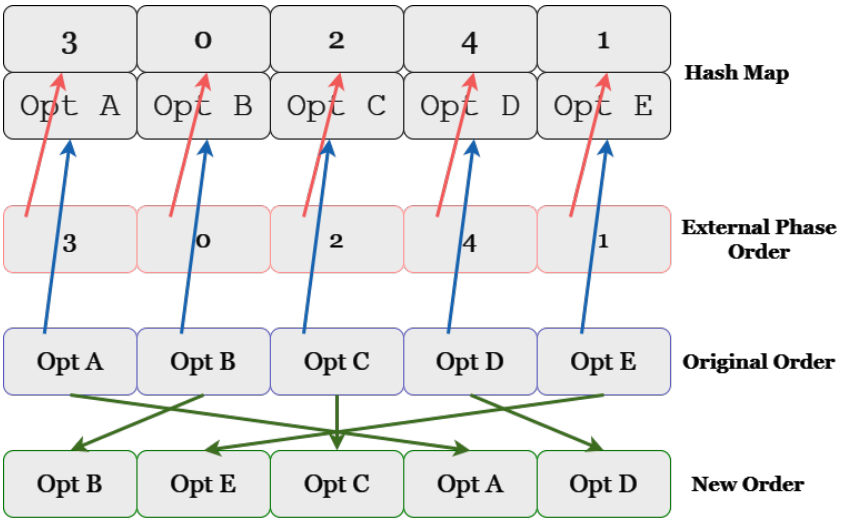


Figure 4.7: Visual depiction of hash building and pipeline re-ordering process

Specifically, I import the phase-order as a String in GHC/Driver/main.hs in hsc-

Simplify, which calls the Core2Core simplifier. The `core2core` function in GHC/Core/Opt/Pipeline.hs calls the `getCoreToDo` function which takes in the compiler flags and uses that list to determine which core passes to do. These Core Pass data structures are then stored in the core_todo list which is then used to modify the source code.

The GHC Optimization Pipeline has 23 steps in it. I have split the pipeline into the two groups described earlier, the "-O2 optimizations" which is also composed of -O1 optimizations, and the "-O0 optimizations". I only re-ordered the steps included in the O2 optimizations. I do not touch the -O0 optimizations for reasons mentioned earlier.

4.2 Experimental Set-Up

Like in flag selection, I used the nofib bench-marking suite. I ran nofib on our modified Stage 2 compiler. Each program was ran with the "-O2" optimization flag. The programs we selected from nofib were the following:

- maillist- Mailing list generator
- cacheprof - An assembly code annotator for GCC
- hidden - Removes hidden lines
- sorting - Implementation of sorting algorithms

These are the same programs selected for flag selection. Table 4.2 shows the global settings for each machine learning model.

Setting	Value
Runs/program	10
Max # of calls to the benchmark/program	1000
Fixed Pipeline Steps	[:13]
Flex Pipeline Steps	[13:]

Table 4.2: Global settings for machine learning models

I gave each step in the pipeline its own name. Thus, the total pipeline becomes: *[static_args, presimplify, specialise, full_laziness_1, simpl3, float_in_1, call_arity, strictness, exitification, full_laziness_2, cse, final, rule_check1, liberate_case, spec_constr, rule_check2, late_specialise, triple_combo, late_dmd_anal, strict_anal, rule_check3, add_caller, add_late]*

For RIO, while each individual run only had 250 permutations, the model was used 4 times to achieve a total of 1000 permutations. The settings for BOCPA can be seen in table 4.3.

Setting	Value
Initial set size	25
Number of impactful optimizations	5
Max iterations	5000
Max Without Improvement	2000
Decay	0.64
Offset	25
Scale	40

Table 4.3: Settings for BOCPA.

For these models, the only performance metric is optimal phase-order runtime. In the flag selection problem, I assumed that the differences in execution times between the RIO and BOCA algorithms were largely trivial. This assumption does not hold for BOCPA and RIO. The graph operations performed are expensive. Other metrics such as compile time, memory allocation, file size, etc. . . are not considered in the results or discussion.

Once an optimal candidate was selected, it was ran against the full GHC testing suite [24] to ensure the compiler still functioned with that specific phase-order. This ensures that each optimal solution is, in-fact, a solution. However, this does create the possibility that several candidates used to generate the optimal one could have caused errors had they been ran against the entire testing suite. This also means that an optimal solution that one of the models produces might fail the compiler tests; however, I have tried to avoid this by using others forms of machine learning to filter out permutations that are unlikely to work properly using the SVM discussed in Methodology.

4.3 Results

Before collecting data, I used RIO and the default phase-order to verify that changes to the phase-order created statistically significant changes in the runtime. I chose a $\alpha = 0.05$. Our results are shown in table 4.4

Based on the statistical evidence, I can reject the null hypothesis.

Speed-up in run-time over the default phase-order was achieved across the board by BOCPA; however RIO, in one instance was unable to find a candidate faster than the

Program	t-state	p-value
cacheprof	21.01	7.78e-61
maillist	-4.25	2.86e-05
hidden	3.65	0.0003
sorting	18.34	7.38e-15

Table 4.4: Statistical significance of changing the phase-order

Program	Configuration	Optimal	Default	RT Change	Avg. Change
cacheprof	BOCA	0.111	0.118	-5.932%	-2.677%
	RIO	0.112	0.118	-5.085%	-3.406%
sorting	BOCA	0.137	0.147	-6.803%	-3.68%
	RIO	0.138	0.147	-6.122%	-3.984%
hidden	BOCA	0.123	0.128	-3.906%	-0.945%
	RIO	0.124	0.128	-3.125%	-1.724%
maillist	BOCA	0.569	0.617	-7.78%	0.312%
	RIO	0.575	0.617	-6.807%	-2.897%

Table 4.5: Optimal phase-order found by RIO and BOCA versus the default phase-order for each program.

default. The lowest speed-up gained by BOCPA was a 3.907% speed-up over the default, and the largest speed-up was 7.78% over the default. Table 4.5 shows the run-times of RIO and BOCPA across all programs, the highest speed-up achieved, and the average speed-up for all candidates per model. On average, BOCPA had the greatest increase in performance vs. the default phase-order with an average optimal speed-up of 5.903% against RIO's 5.285% average speed-increase. BOCPA was able to outperform RIO in all tests. Figure 4.8 shows the distribution in run-time per model per program. Each model was normalized around its Z-Score. Figure 4.9 shows the result normalized around the average run-time per program for the default phase-order.

Similarly to the flag-selection problem, both RIO and BOCPA have outliers. Both RIO and BOCPA have a significant number of negatively performing outliers with only three positively performing outliers. All of the positively performing outliers were a result of BOCPA. The most optimal result, in terms of standard deviation from the -O2, was about ten standard deviations better compared to -O2, while the lowest performing results were upwards of 40 standard deviations slower.

Figure 4.10 plots the outliers seen in 4.9. For Sorting, Cacheprof, and Hidden, the number of outliers in the population is evenly dispersed through all 2000 iterations. The number of outliers over 2000 iterations increases linearly; however, there is an

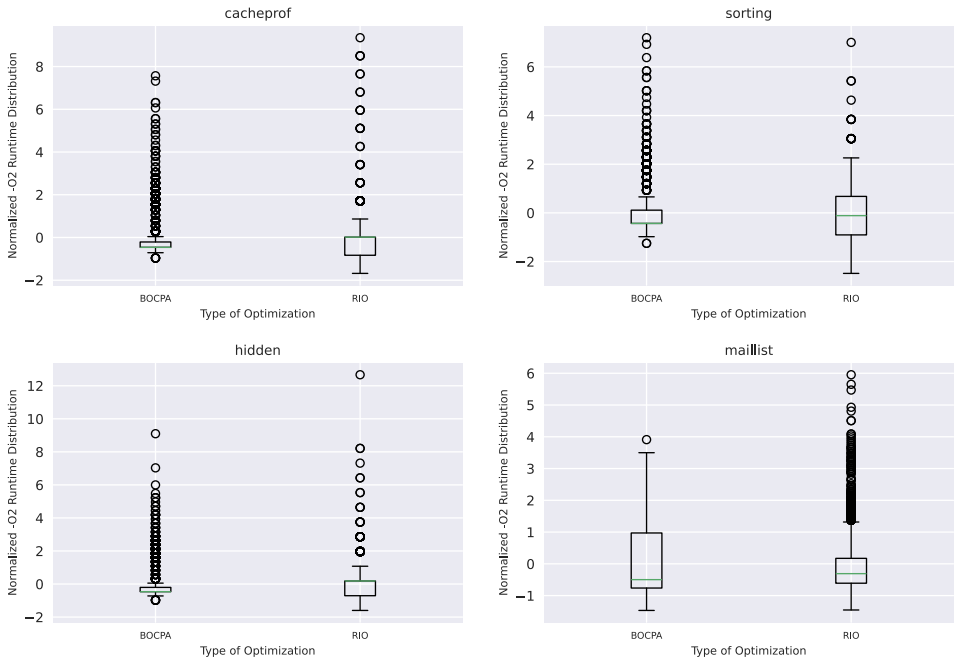


Figure 4.8: BOCPA vs. RIO Runtime distribution normalized around the Z-Score in sorting

exception. Maillist had only one negative performing outlier, and as a result it is not visible on Figure 4.10.

4.4 Discussion

Both RIO and BOCPA had a significant number of outliers, this is unexpected and differs from the flag selection problem. The reason why RIO has many outliers (and why BOCA should also have outliers) has been explained in Discussion section of the flag-selection problem. Here I will explore some possible reasons for the increase in outliers for BOCPA as compared to BOCA. In BOCA, there are two areas were candidates are created: initial training set generation and after selecting impactful optimizations. In BOC(P)A, the initial training set is completely random, so I expect most outliers to be created there. During candidate creation, the similarity to the progenitor phase-order in the training set corresponds to the value of C as determined by the decay function show in figure 3.3. When BOCA selects a number of unimpactful flags to add, it chooses between $[0 - C]$ unimpactful flags. Thus, there is also a greater chance of outliers being created in this step when C is a large value, which it is when the number of completed iterations is small. A

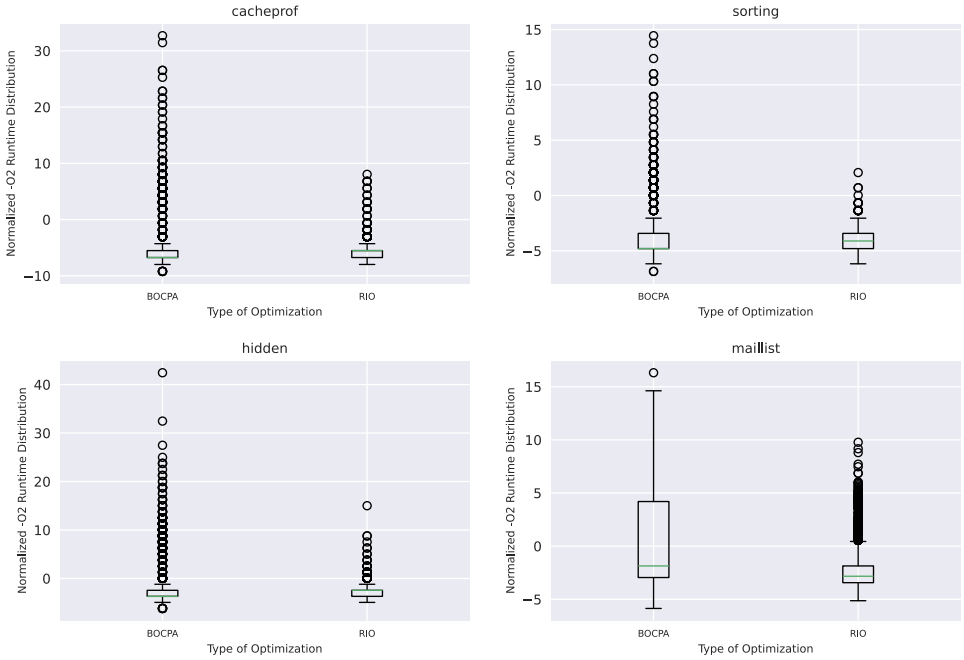


Figure 4.9: BOCPA vs. RIO Runtime distribution normalized around the default phase-order in hidden

similar mechanism exists in BOCPA. In BOCPA, a number of unimpactful rules present in the progenitor phase-order are added to the new candidate. BOCPA chooses between $[0 - (size(\text{Unimpactful Flags of Progenitor}) - C))$. This ensures that the new candidates are, like in BOCA, more random and less similar to their progenitors when completed iterations are low. To test this, I disabled the early-termination present in BOCPA and had the algorithm run to completion to see if the rate in which negatively performing outliers were acquired decreased as C decreased. These results are seen in Figure 4.11. Even when C is low, the number of outliers added seems to be consistent. Therefore I do not think early stopping is responsible for the increase in outliers.

One possible explanation for why BOCPA produced so many outliers is that this altered step in candidate creation in BOCPA creates more variability than its equivalent in flag selection.

Another possibility is the existence of "super bad rules" which dooms any permutation regardless of whatever rules it has. There could be a few of these; rare enough that the random forest does not have time to adapt to them, but common enough

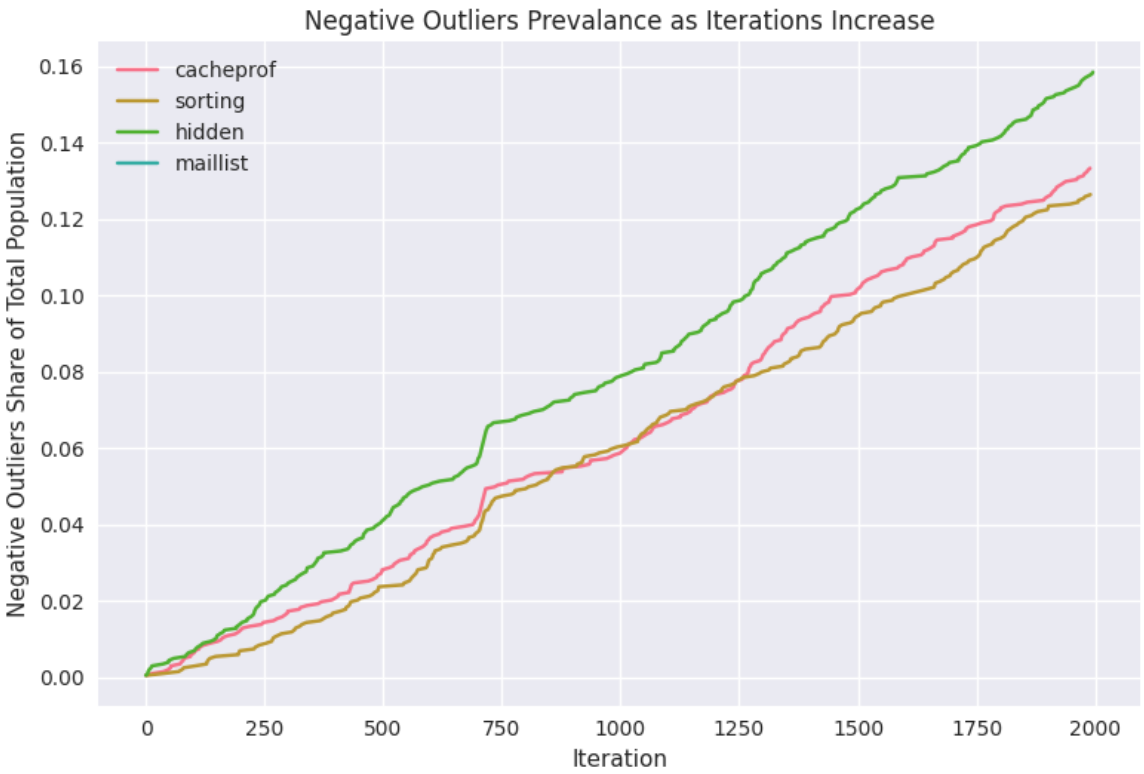


Figure 4.10: Percent of Negative Performing Outliers Present in BOCPA Training Set as Iterations Increase

that a handful of new candidates get saddled with them which then results in the poor showing in the test suite. However, Figure 4.10 shows that the number of bad outliers increases linearly through the entire runtime of the program. So either negative performing rules are so abundant that they are still being discovered at the same rate throughout the entire BOCPA life-cycle, or the phase-order problem is just more susceptible to bad outliers in general.

The data from Figures 4.9 suggest that while there are some phase-orders that are more optimal than the default for that program, there are definitely significantly worse performing phase-orders, and there is more of them than more optimal permutations. This suggests that the default phase-order is already close to being the optimal for each program. For example, while BOCPA was able to find a phase-order faster than the default for maillist, the "average improvement" was actually slower than the default as seen in table 4.5.

The one advantage that RIO has over BOCPA is its time complexity. While RIO and BOCPA for flag optimization are similar in their execution time, BOCPA for phase-order optimization has a few additional steps that significantly decrease the

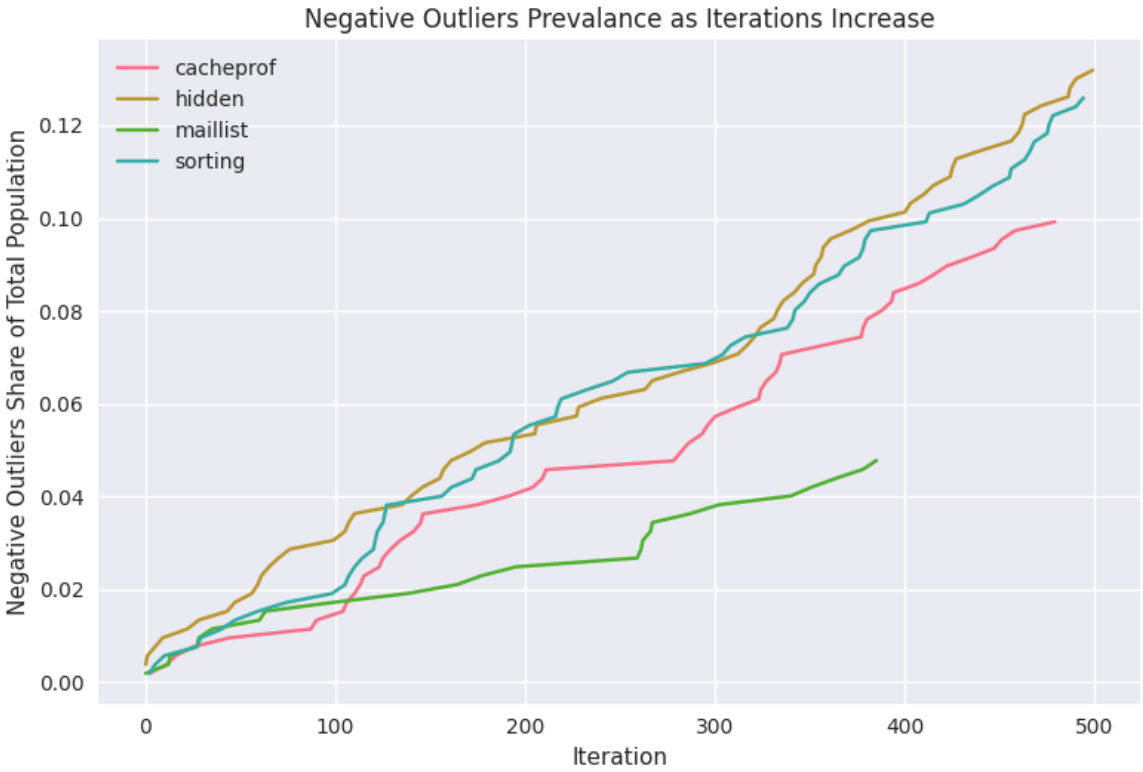


Figure 4.11: Percent of Negative Performing Outliers Present in BOCPA Training Set as Iterations Increase - No Early Stopping

performance, the most pressing of which is creating new candidates. The graph operations required by BOCPA are slower than the simple random selection of RIO.

Lastly, while MCA analysis was able to reveal several flags that were responsible for most of the variance in the flag selection problem, the same analysis revealed no such thing for the phase-order problem. The MCA component plot is shown in Figure 4.12. From this MCA result, I can conclude that there is not a group of "strong" rules that are responsible for most of the Runtime.

4.5 Conclusions - Phase-Order

Compiler phase-order optimizations is an area ripe for exploration within the Glasgow-Haskell Compiler and using machine learning, I was able to achieve program-specific speed-ups of between 3% and 8%. These results are slightly behind the improvements in the phase-order problem received by other models detailed in the Related Works section. My upper result of 8% was the average in most of the Related Works.

While RIO achieved speed-up when applied to the flag combination problem, it

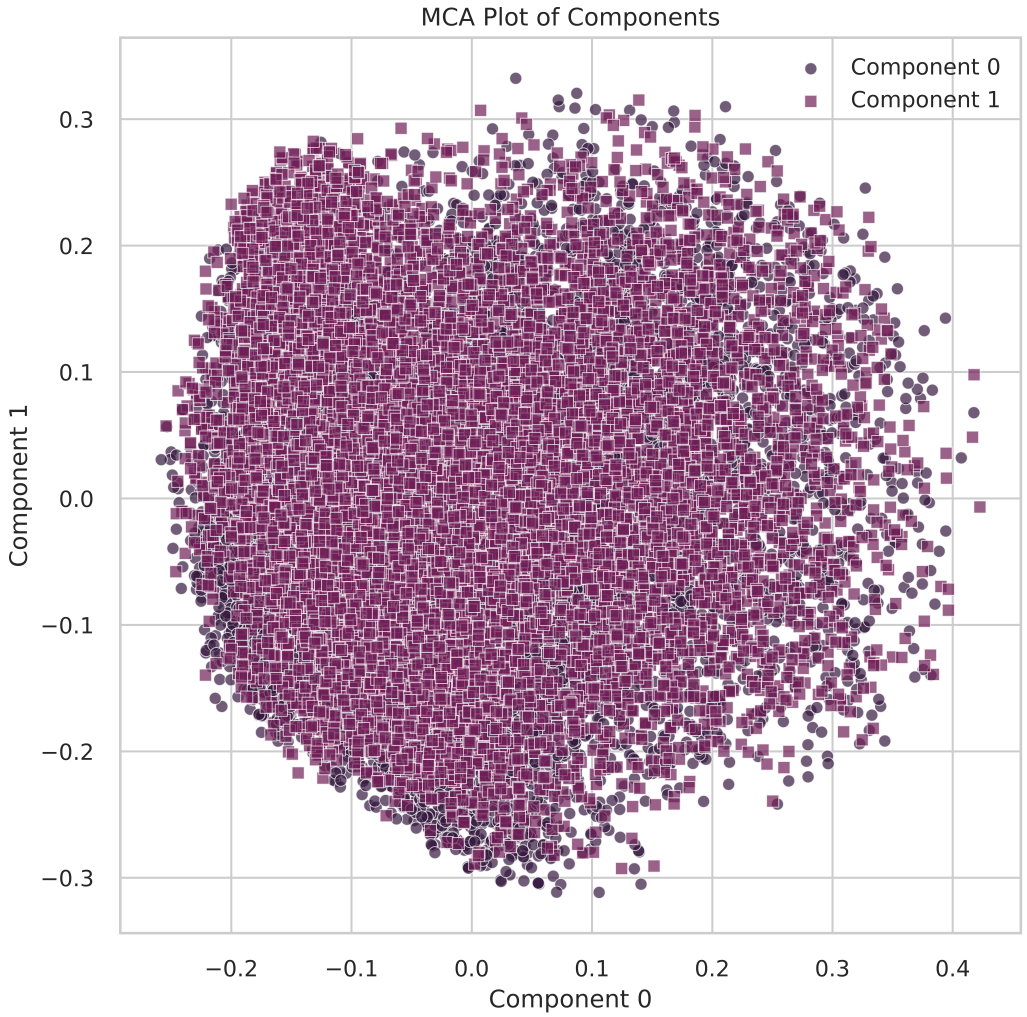


Figure 4.12: Two Component MCA Plot for BOCPA and RIO Runtimes

is clear that it is less effective than BOCPA at finding improvement in the phase-order problem. BOCPA is an effective algorithm in finding optimal phase-orderings; however, BOCPA is significantly slower than BOCA and RIO even when performing fewer iterations.

The default phase-order in the GHC is effective across multiple programs, this is evident in that while the speed-ups were modest, there was lots of room for speed decreases as most evident in Hidden, where some of BOCPA's results were 40 standard deviations worse than -O2 as seen in Figure 4.9 and as glimpsed in Figure 4.10. However, it is still possible that there exists a better "default" phase-order than

the current default. It should be noted that no clear candidates for new optimal rule exists due to the lack of clear and convincing groups in the MCA analysis seen in Figure 4.12. I suggest this as a future area of research in the GHC.

Chapter 5

CONCLUSION

In this thesis, I sought to use machine learning to optimize the program-specific flag-selection and optimization phase-order of several Haskell programs in the GHC compiler. For flag selection, the algorithms were RIO, GA, and BOCA. I achieved across-the-board speed-up with all three algorithms, but FOGA had the best improvement, though BOCA was the quickest to execute. All three models did not achieve the same speed-up as their original papers, but they all did achieve speed-ups. Table 5.1 shows the most significant speed-ups achieved.

For phase-order, the algorithms were RIO and BOCPA. Heavy modifications had to be made to BOCA to get it compatible for the phase-order problem, but the modifications were successful and BOCPA and RIO again achieved across-the-board speed-up. BOCPA achieved the best speed-up, but the graph operations added to BOCPA slow it down, making RIO a more efficient algorithm. Again, I did not achieve as great of a speed-up as other researchers have gained in their papers as detailed in the Related Works section.

Importantly, the average speed-up from optimizing the phase-order is greater than the average speed-up from optimizing the flag-selection, if you discount the exceptional result from Maillist in table 3.4. This was a surprising result. Normally, a developer is prevented from changing the phase-order at all. The phase-order is determined by the compiler, and thus the compiler developers by proxy. Thus, one would expect it would be extremely optimized with little room to improvement. My results show that there is improvement to be made at a relatively low cost. Optimizing in the GHC compiler is relatively easy and effective.

Suggestions for Future Work

One aspect of this research that frustrated me were the occasional forbidden positions among the -O2 optimizations and the inflexibility of the -O0 optimizations in the phase-order problem. Further work should look into whether there is a rational for why some orderings produce compiler errors. Finding such a rationale could further improve BOCPA and RIO by eliminating those problem orderings ahead of time instead of relying on the SVM. A breakthrough in re-ordering the -O0 optimizations

Program	Problem	Model	Runtime Change
Cacheprof	Flag Selection	RIO	-1.822%
		BOCA	-2.041%
		FOGA	-1.373%
	Phase Order	RIO	-5.085%
		BOCPA	-5.932%
Sorting	Flag Selection	RIO	-0.481%
		BOCA	-1.449%
		FOGA	-1.896%
	Phase Order	RIO	-6.112%
		BOCPA	-6.803%
Hidden	Flag Selection	RIO	-1.556%
		BOCA	-2.282%
		FOGA	-3.095%
	Phase Order	RIO	-3.125%
		BOCPA	-3.906%
Maillist	Flag Selection	RIO	-1.328%
		BOCA	-15.17%
		FOGA	-2.804%
	Phase Order	RIO	-6.807%
		BOCPA	-7.78%

Table 5.1: Summary of Important Results in Flag Selection (Over -O2) and Phase-Order (Over Default)

would increase the search space from just 9! to 23!, there could be significant room for improvement.

Previous work done in GHC shows that the compiler heuristics for at least some optimizations are insufficient to meet modern day developer needs. Hollenbeck et al. found this was true for inlining in the GHC [10], and Wang et al. found that strictness in the GHC compiler also requires significant developer insight [27], insight that I think the compiler should have. I have shown that there exists several detrimental optimizations in the GHC. If these are generally detrimental optimizations or just program specific detriments has been left unproven, but if identified they could also be areas for modernization just like inlining.

If developers can be assured that changing the phase-order in the GHC will not fundamentally break the compiler, then I think it would be a natural and advantageous next step to make this a permanent feature of the Glasgow-Haskell Compiler. A simple configuration file for the GHC with a designated phase-order or even a command line argument would make phase-order optimization more similar to its

more popular cousin in flag-selection.

BIBLIOGRAPHY

[1] AlgorithmsX. *Random topological sorting with uniform distribution in near linear time*. https://stackoverflow.com. Question answer. (Licensed under CC BY-SA 3.0: <https://creativecommons.org/licenses/by-sa/3.0/>). Apr. 2017. URL: https://stackoverflow.com/a/38552721/17917005 (visited on 05/10/2024).

[2] Fulya Altiparmak et al. "A steady-state genetic algorithm for multi-product supply chain network design". In: *Computers and Industrial Engineering* 56 (2 Mar. 2009), pp. 521–537. ISSN: 03608352. DOI: 10.1016/j.cie.2007.05.012.

[3] Jason Ansel et al. "OpenTuner: An extensible framework for program autotuning". In: *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*. 2014, pp. 303–315. DOI: 10.1145/2628071.2628092.

[4] Amir H Ashouri et al. "A survey on compiler autotuning using machine learning". In: *ACM Computing Surveys (CSUR)* 51.5 (2018), pp. 1–42.

[5] Junjie Chen et al. "Efficient compiler autotuning via bayesian optimization". In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE. 2021, pp. 1198–1209.

[6] Yang Chen et al. "Deconstructing iterative optimization". In: *Transactions on Architecture and Code Optimization* 9 (3 Sept. 2012). ISSN: 15443566. DOI: 10.1145/2355585.2355594.

[7] Keith D Cooper et al. *Compilation Order Matters* *.

[8] Corinna Cortes and Vladimir Vapnik. "Support-Vector Networks". In: *Machine Learning* 20.3 (Sept. 1995), pp. 273–297. DOI: 10.1007/bf00994018.

[9] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. "Exploring Network Structure, Dynamics, and Function using NetworkX". In: *Proceedings of the 7th Python in Science Conference*. Ed. by Gaël Varoquaux, Travis Vaught, and Jarrod Millman. Pasadena, CA USA, 2008, pp. 11–15.

[10] Celeste Hollenbeck, Michael F. P. O'Boyle, and Michel Steuwer. "Investigating magic numbers: improving the inlining heuristic in the Glasgow Haskell Compiler". In: Association for Computing Machinery (ACM), Sept. 2022, pp. 81–94. DOI: 10.1145/3546189.3549918.

[11] Kathy (CalTech) Johnson and Lian Tze (Overleaf) Lim. *Caltech Thesis LaTeX Template (without logo)*. (Licensed under CC BY-SA 4.0: <https://creativecommons.org/licenses/by-sa/4.0/>). 2021. URL: https://www.overleaf.com/latex/templates/caltech-thesis-latex-template-without-logo/xsjkwzcftrym.

[12] A. B. Kahn. “Topological sorting of large networks”. In: *Commun. ACM* 5.11 (Nov. 1962), pp. 558–562. ISSN: 0001-0782. DOI: 10.1145/368996.369025. URL: https://doi.org/10.1145/368996.369025.

[13] P.A. Kulkarni et al. “Exhaustive optimization phase order space exploration”. In: *International Symposium on Code Generation and Optimization (CGO’06)*. 2006, 13 pp.–318. DOI: 10.1109/CGO.2006.15.

[14] Sameer Kulkarni and John Cavazos. “Mitigating the compiler optimization phase-ordering problem using machine learning”. In: *SIGPLAN Not.* 47.10 (Oct. 2012), pp. 147–162. ISSN: 0362-1340. DOI: 10.1145/2398857.2384628. URL: https://doi.org/10.1145/2398857.2384628.

[15] Andy Liaw, Matthew Wiener, et al. “Classification and regression by randomForest”. In: *R news* 2.3 (2002), pp. 18–22.

[16] Luiz G.A. Martins et al. “Clustering-based selection for the exploration of compiler optimization sequences”. In: *ACM Transactions on Architecture and Code Optimization* 13 (1 Mar. 2016). ISSN: 15443973. DOI: 10.1145/2883614.

[17] “MiCOMP: Mitigating the compiler phase-ordering problem using optimization sub-sequences and machine learning”. In: *ACM Transactions on Architecture and Code Optimization* 14 (3 Sept. 2017). ISSN: 15443973. DOI: 10.1145/3124452.

[18] Will Partain. “The nofib benchmark suite of Haskell programs”. In: *Functional Programming, Glasgow 1992: Proceedings of the 1992 Glasgow Workshop on Functional Programming, Ayr, Scotland, 6–8 July 1992*. Springer. 1993, pp. 195–202.

[19] F. Pedregosa et al. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[20] Anupriya Shukla, Hari Mohan Pandey, and Deepti Mehrotra. “Comparative review of selection techniques in genetic algorithm”. In: *2015 international conference on futuristic trends on computational analysis and knowledge management (ABLAZE)*. IEEE. 2015, pp. 515–519.

[21] Gurinder Singh et al. *Comparative Review of Selection Techniques in Genetic Algorithm*. ISBN: 9781479984336.

[22] Michael D Smith. “Overcoming the challenges to feedback-directed optimization (keynote talk)”. In: *Proceedings of the ACM SIGPLAN workshop on Dynamic and adaptive compilation and optimization*. 2000, pp. 1–11.

[23] Burak Tağtekin et al. “FOGA: flag optimization with genetic algorithm”. In: *2021 International Conference on INnovations in Intelligent SysTems and Applications (INISTA)*. IEEE. 2021, pp. 1–6.

[24] GHC Team. *The Glasgow Haskell Compiler User’s Guide*. GHC Project. URL: https://www.haskell.org/ghc/.

[25] *TikZ & PGF (version 2.10)*. http://sourceforge.net/projects/pgf. Till Tantau et al. 2010.

[26] Michael Vollmer et al. “Meta-Programming and Auto-Tuning in the Search for High Performance GPU Code”. In: *Proceedings of the 4th ACM SIGPLAN Workshop on Functional High-Performance Computing*. FHPC 2015. Vancouver, BC, Canada: Association for Computing Machinery, 2015, pp. 1–11. ISBN: 9781450338073. DOI: 10.1145/2808091.2808092. URL: https://doi.org/10.1145/2808091.2808092.

[27] Yisu Remy Wang, Diogenes Nunez, and Kathleen Fisher. “Autobahn: Using genetic algorithms to infer strictness annotations”. In: *Proceedings of the 9th International Symposium on Haskell* (Sept. 2016). DOI: 10.1145/2976002.2976009.

[28] Debbie Whitfield and Mary Lou Soffa. *An Approach to Ordering Optimizing Transformations**.

This template was provided by CalTech [11].