EAFT: EVOLUTIONARY ALGORITHMS FOR GCC FLAG TUNING

BURAK TAĞTEKİN

MEF UNIVERSITY

APRIL 2023

MEF UNIVERSITY
GRADUATE SCHOOL OF SCIENCE AND ENGINEERING
MASTER'S IN INFORMATION TECHNOLOGIES

M.Sc THESIS

EAFT: EVOLUTIONARY ALGORITHMS FOR GCC FLAG TUNING

Burak TAĞTEKİN
ORCID No: 0000-0002-8405-5695

Asst. Prof. Dr. Tuna ÇAKAR

APRIL 2023

ACADEMIC HONESTY PLEDGE

This is to certify that I have read the graduation project and it has been judged to be successful, in scope and in quality and is acceptable as a graduation project Master's Degree in Information Technologies.

Name Surname: Burak TAĞTEKİN

Signature:

ABSTRACT

EAFT: EVOLUTIONARY ALGORITHMS FOR GCC FLAG TUNING

Burak TAĞTEKİN

M.Sc in Information Technologies

Thesis Advisor: Asst. Prof. Dr. Tuna ÇAKAR

April 2023, 53 Pages

The runtime of written codes is a matter of great importance, especially for code that is compiled once and executed multiple times. It is very important for developers to ensure that the resources required by a code are used as efficiently as possible, and that the runtime is as low as possible. Developers who use compilers such as GCC or LLVM to compile and run code written in C or C++ can optimize their code manually and, with certain optimization pointers, are able to make it run faster. This will provide the shorter runtime, but completıng this manual optimization is within the abilities of every developer since determining the right combination from more than 200 flags requires significant expertise.

Many studies have tackled this issue. In this study, Evolutionary Algorithms for GCC Flag Tuning (EAFT) have been developed as a solution to this problem. This Autotuner, which is completely open-source, runs the code provided by the end user according to the specifications also selected by the end user, and searches for the most suitable optimization markers. For the code to be given In line with this study, which specifically addresses the end user, the user can input the code path directly from the Terminal, as well as specify the selection method and the crossover to be used. These choices can be made without the need to alter the code. The genetic algorithm and particle swarm optimization to be used is also presented to the user in EAFT, and unlike in other studies, genetic algorithm contain not one but several models.

Keywords: Compiler, GCC, Evolutionary Algorithms, Genetic Algorithm, Autotuner.

Numeric Code of the Field: 123456

ÖZET

EAFT: EVRİMSEL ALGORİTMALAR İLE GCC İŞARETÇİ OPTİMİZASYONU

Burak TAĞTEKİN

Bilişim Teknolojileri Tezli Yüksek Lisans Programı

Tez Danışmanı: Dr. Öğr. Üyesi Tuna ÇAKAR

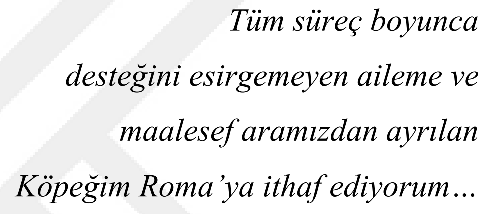
Nisan 2023, 53 Sayfa

Yazılan kodların çalışma süresi, özellikle de bir kez derlenip birden fazla kez çalıştırılacak olanlar için çok büyük önem arz etmektedir. Çalışma süresi boyunca kodun kullanacağı kaynakların verimli hale getirilmesi ya da bekleme sürelerinin azaltılması birçok geliştirici için çok önemlidir. C, C++ gibi kodların derlenip çalıştırılması hususunda GCC ya da LLVM gibi derleyiciler kullananlar bu konuda optimizasyon işini manuel bir şekilde yapıp kodun belirli optimizasyon işaretçileri ile daha kısa sürede çalışmasını sağlayabilir. Bu durum yukarıda bahsi gecen yararları sağlayacaktır ancak seçimi yapmak her geliştirici için o kadar da kolay olmamaktadır zira 200'den fazla flag içerisinden doğru kombinasyonu seçmek uzmanlık isteyen bir alandır.

Bu problemin de önüne geçmek için literatürde birçok çalışma yapılmıştır. Bu çalışma kapsamında ise bu soruna bir çözüm olarak EAFT: Evolutionary Algorithms for GCC Flag Tuning geliştirilmiştir. Tamamen açık kaynaklı olan bu Autotuner, son kullanıcının temin edeceği kodu, yine son kullanıcının seçeceği özellikler doğrultusunda çalıştırıp onun için en uygun olan optimizasyon işaretçilerini arar. Son kullanıcıya özellikle hitap eden bu çalışma doğrultusunda verilecek olan kod için kullanıcı hangi seçim metodunu kullanacağından hangi çaprazlamanın kullanılmasını istediğine kadar birçok noktada direkt olarak Terminal üzerinden seçim yapılabilmesine olanak sağlar. Bu seçimler EAFT içerisinde bir kısım ya da kod değiştirilmeden yapılabilecek kolaylıktadır. Kullanılacak olan evrimsel algoritma da EAFT içerisinde kullanıcının seçimine sunulmuştur ve evrimsel algoritmalar diğer çalışmalardan farklı olarak bir değil birden fazla model içerir.

Anahtar Kelimeler: Derleyici, GCC, Evrimsel Algoritmalar, Genetik Algoritma.

Bilim Dalı Sayısal Kodu: 123456



Tüm süreç boyunca
desteğini esirgemeyen aileme ve
maalesef aramızdan ayrılan
Köpeğim Roma'ya ithaf ediyorum…

TABLE OF CONTENTS

ABSTRACT i
ÖZET ii
LIST OF TABLES vi
LIST OF FIGURES vii
1.INTRODUCTION 1
1.1 Purpose of the Thesis 1
1.2 Literature Review 2
1.3 Overview 6
2.BACKGROUND 7
2.1. Compiler 7
2.2 Interpreters 7
2.3 Optimization 8
2.4 Selecting the Best Flag Sequence 11
2.4.1 Optimization Space 12
2.5 Benefits and Problems 12
2.5.1 Polybench 12
2.5.2 Optimization Effects on Compiler 13
2.5.3 Comparison between O2 and O3 16
3.EVOLUTIONARY ALGORITHMS 21
3.1 Genetic Algorithm 21
3.1.1 Inspiration and First Look 21
3.1.2 Crossover and Mutation 23
3.1.2.1 Partially Mapped Crossover 24
3.1.2.2 Ordered Crossover 25
3.1.2.3 Generalized N-Point Crossover 25
3.1.2.4 Uniform Crossover 26
3.1.2.5 Edge Recombination Crossover 26
3.1.3 Selection Methods 27
3.1.3.1 Roulette Selection 27
3.1.3.2 Tournament Selection 28
3.1.4 Mutation 28
3.1.5 Genetic Algorithm Models 28

3.1.5.1 Generational Model 28
3.1.5.2 Steady State Model 29
3.1.5.3 Down to Size Model 30
3.1.5.4 Ring Model 31
3.1.5.5 Only Mutation Model 32
3.2 Particle Swarm Optimization 32
3.2.1 Formulization of PSO 32
4. RESULTS 34
4.1 Command Line Interface 34
5. DISCUSSION 43
CONCLUSION AND FUTURE WORKS 47
REFERENCES 49

LIST OF TABLES

Table 1.1 Features from [1] 6
Table 2.1 Effects of Optimization Flags 9
Table 2.2 Optimization Flags 10
Table 2.3 Polybench Benchmarks 13
Table 4.1 Benchmark Results 35
Table 4.2 All Flag Opening Results 36

LIST OF FIGURES

Figure 1.1 Input Effect on Assembly 3
Figure 1.2 Branching 4
Figure 2.1 From Source Code to Machine Code 7
Figure 2.2 O2 Optimization Effects of Polybench 15
Figure 2.3 O2 vs O3 Optimization Levels 16
Figure 2.4 O2 vs O3 For Each Benchmark 17
Figure 2.5 Runtime Distribution of Benchmarks with Flags 18
Figure 2.6 Runtime Distribution of All Benchmark with Flags 20
Figure 3.1 Finches Beak [3] 22
Figure 3.2 Optimization Flag Representation in Chromosome 23
Figure 3.3 From Gene to Population 23
Figure 3.4 Partially Mapped Crossover 24
Figure 3.5 Ordered Crossover 25
Figure 3.6 Generalized N-Point Crossover 26
Figure 3.7 Edge Recombination Crossover 27
Figure 3.8 Generational Model 29
Figure 3.9 Steady State Model 30
Figure 3.10 Down to Size Model 31
Figure 3.11 Ring Model 31
Figure 4.1 CLI 34
Figure 4.2 2MM Population Changes 38
Figure 4.3 Mutation Rate Changes 38
Figure 4.4 Crossover Rate 39
Figure 4.5 Seidel-2D Population Changes 39
Figure 4.6 Seidel-2D Crossover Rate 40
Figure 4.7 EAFT vs Opentuner M.M. 40
Figure 4.8 EAFT vs Opentuner TSP 41
Figure 4.9 EAFT vs Opentuner Time Comparison 41

ABBREVIATIONS

CLI	: Command Line Interface
DTS	: Down to Size
ERX	: Edge Recombination Crossover
GA	: Genetic Algorithm
GCC	: GNU Compiler Collection
GM	: Generational Model
OX	: Ordered Crossover
PMX	: Partially Mapped Crossover
PSO	: Particle Swarm Optimization
RM	: Ring Model
SSM	: Steady State Model
TSP	: Travelling Salesman Problem
UX	: Uniform Crossover

1. INTRODUCTION

1.1 Purpose of the Thesis

Programs written in C and C++ languages are widely used today in many different areas, from operating systems to image processing applications. Many software developers prefer these languages for performance reasons.

Programming languages have many features, some of which differ from a language to the next. This thesis will focus on “compile”, one of the features that differ the most between languages. Tools such as Clang or GNU Compiler Collection (GCC) are necessary to run code written in C and C++ languages; these tools are called compilers and their purpose is to convert source code to machine code. Many processes occurring during compilation will be mentioned and detailed in this thesis, but the focus will be on the way developers can intervene in the optimization stages.

In order to convert source code to machine code, a compiler performs a series of operations. These can directly affect the runtime of the code, potentially shortening it when applied correctly. In applications where gaining mere seconds is important—for instance in telecommunication systems—shortening the code execution time is of great importance. The purpose of this thesis is to introduce methods that can be applied to shorten the code runtime, and to present a heuristic search algorithm that can be used for this purpose. In this study, we used GCC 9.3, which contains 212 flags; other versions of GCC may have a different total number of flags.

Heuristic search is a technique in artificial intelligence that aims at providing an approximate solution for a problem, rather than the single best result; it is particularly useful when the set to be searched is very large. Several algorithms can find solutions to the problems faced by computer scientists; in this thesis, Genetic Algorithm (GA) and Particle Swarm Optimization (PSO) will be used to find the most appropriate optimization sequence of the 212 flags contained in GCC 9.3.

1.2 Literature Review

The search for the best optimization flags has been the subject of much research, and different approaches have been employed to try and solve the issue.

Opentuner is a general purpose Autotuner. It can find the most suitable flags using several search algorithms that it implements and runs together [4]. Rather than implementing these algorithms in a generic way, Evolutionary Algorithms for GCC Flag Tuning (EAFT) uses binary search algorithms that fully match the GCC flag optimization. In this way, EAFT ensures that only the flags that will enable the code to run faster will be included. Compared to Opentuner, which is written in Python, GOLANG is used in the EAFT implementation. Not only is Go language faster than Python in terms of multi-threading, but the runtime of the search algorithm is also shorter.

Studies on flag optimization are not concerned exclusively with search algorithms. By making static and dynamic analyses, it is the structure of the code itself that is examined. These analyses are also conducive to making prediction models.

Static analyses can be done on the assembly code of the program with tools such as LLVM-MCA [5]. By using tools such as Ithemal [6] that evaluate the features coming from LLVM-MCA, it is possible to predict the runtime of a code without the need to actually run it. A consequence of only predicting the runtime of a code is that multiple features cannot be determined. The code may be processing the input values it receives from outside, or it may rely on a value it calculates at runtime. For such reasons, only estimating the code's runtime is not always desirable. This issue is known in the literature as the "Halting Problem" [6]. Likewise, the runtime of the code can be increased or shortened simply by changing the input size, without altering the code structure. In this case, the analysis tools will not be able to interpret the situation and will thus not be able to make correct predictions, since they will not detect any change in the code. As shown in Figure 1.1, whether the input given to the square function is 1 or 1,000,000, there is no difference in the LLVM-MCA output.

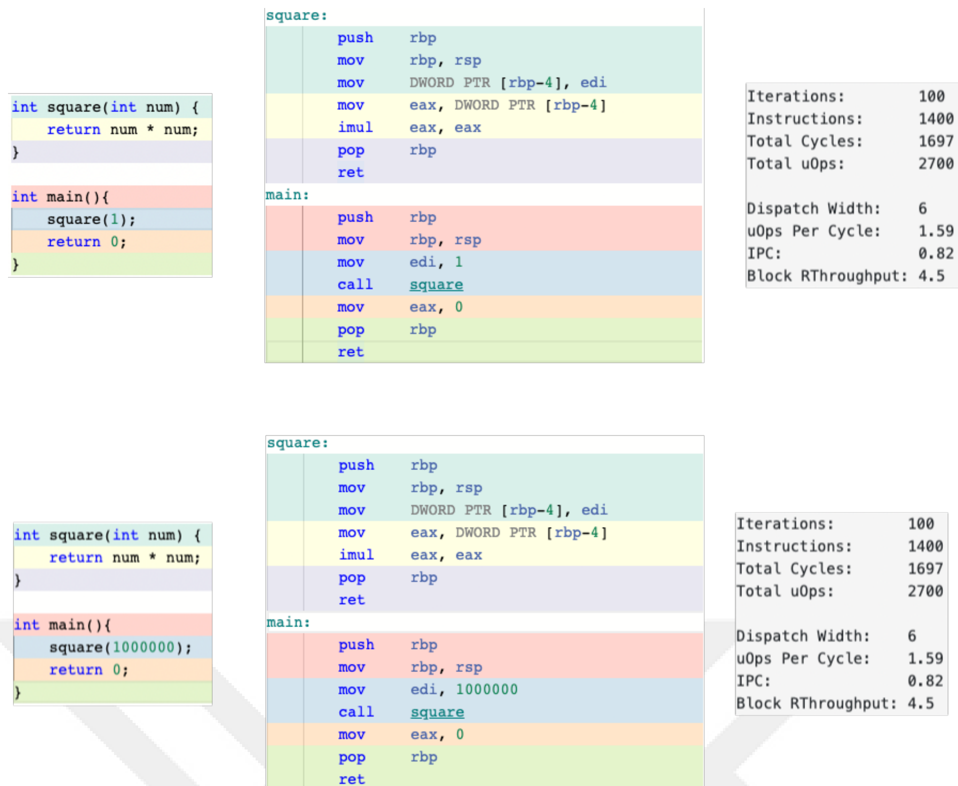


Figure 1.1: Input Effect on Assembly.

Unlike LLVM-MCA, Ithemal [6] analyzes only a basic block code instead of the assembly code of the whole program. There are too many basic blocks in long and complex programs, and if we consider the conditions they contain, we notice that the working structure of the code is highly variable. As shown in Figure 1.2, a simple if / else in the code leads to two different possibilities. This results in lower accuracy values along with an increased complexity of the code.

Cobayn [8] is a tool that aims at predicting the static feature values of the code as a speedup prediction using Bayesian networks. This tool releases static features with Milepost GCC [9]. The fact that the team that developed Milepost worked with versions 4.X.X of GCC and did not make the project compatible with newer GCC versions led to the project eventually becoming obsolete. This GCC versions are outdated for Cobayn using Milepost. On the other hand, Cobayn implemented with Matlab is more complex to use than EAFT, written in Go language. The static features released by Milepost have inspired other works [9]. These features, numbering 56 in total, comprise the number of basic blocks in the assembly code, the total number of edges in the control flow graph, the number of direct calls in the method, the total number of instructions, and the number of methods that return integers or pointers.

These features search for information in the assembly code (as mentioned in the Cobayn), along with the flags that will speed up the code. The order of the flags is also important and that issue has been investigated in several studies, including as part of the Cobayn method [10]. Wang *et al.* [11] did a research on Feature Engineering from static code features. They used machine learning models to identify code structure and attempted compilation processes using models. As the authors mentioned, Machine Learning is not a panacea, one of the main issue being that the code structures significantly differ from a program to next.

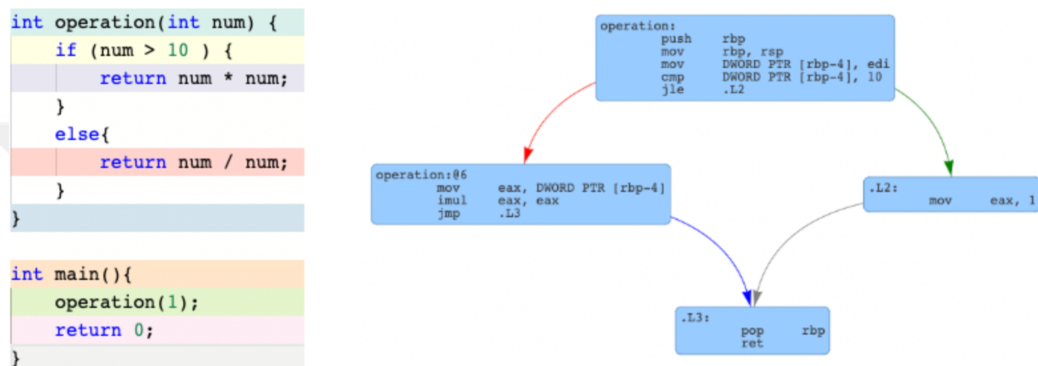


Figure 1.2: Branching

Using a different codebase, Cooper *et al.* [12] also tackled optimization problems. The authors tried to reduce the code size by changing the optimization flags, and reached their goals using genetic algorithms. Their research also showed that a set of optimization flags reduces the code size regardless of the structure of the code.

Zhong *et al*. [13] tried to decrease code runtime with a different heuristic search method, Simulated Annealing. As the authors of the study acknowledged, their GCC version was not one of the most recent. This is important, since GCC performs better on optimization when the compiler version and the architectures are more recent. Zhong *et al*. pointed out that their optimization reduced the code runtime when compared with O3 optimization level, but not more than other tools mentioned in this literature review.

Dubach *et al*. [14] created a series of features through code assembly. In their research, they used the UTDSP Benchmark; ten different problems were considered, some of which provided a maximum speedup of 50%, others a speedup of 1%.

Static features are usually created from assembly to represent the structure of the code. Although it seems reasonable to collect information about flag optimization by removing the embeddings of the code with approaches such as the one described in [15], two problems arise. First, as long as the flags do not change the high-level structure of the code, help can be obtained from the assembly level to see the change. Since other approaches examines the code semantically, the predicted result changes when the names of the functions given in [15] are changed. Since this is an undesirable result, the approach described in [15] is not useful for our study.

The order of the optimization flags to be selected is also important. Ashouri *et al*. [16] achieved a runtime acceleration of 4% by relocating the optimization flags when they compiled the code according to the measurements made in his study. Phase-ordering is tackled right after having selected the optimization flags.

Alongisde static features, the scientific literature is also concerned with the dynamic feature values of the code and the selection of optimization flags. In the 2007 study by Cavazos *et al*. [1], the authors collected the feature values that could be obtained when the code was running. As shown in Figure 1.4, values such as cache hit, branches, and total cycle were used. As a result, by using PathScale compiler in SPEC2000, these values improved by 10%.

In a study by Ashouri *et al*. [17], the code was profiled using Micro-architectural Independent Characterization of Applications (MICA) tool. No other static analysis was used in the study and only the dynamic profiling task was undertaken by MICA. The model the authors built using Bayesian networks achieved an average speedup of 1.5 times in the cBench benchmark. Unlike other studies, [17] considered only eight optimization flags, which corresponds to approximately 6% of the total number of flags.

1.3 Overview

The thesis is organized as follows. Chapter 2 first considers the Compilation, then elaborates on Optimization flags and compilers. Chapter 3 explores in detail the searching algorithms, models, crossover techniques and other elements used in EAFT. The various stages of the autotuning process are illustrated as necessary. Chapter 4 is concerned with the Command Line Interface (CLI). In Chapter 5, a thorough discussion of the results obtained from our experiments and analyze the performance of EAFT on various benchmark problems were provided. Finally, thesis conclude by summarizing findings and suggesting directions for future research.

Table 1.1 Features from [1]

Floating Point	***Branch Stats***	***L1 & L2 Cache***	***TLB Statistics***
Add	Instructions	Data Hit, Miss	Data Lookaside Miss
Multiply	Cond. Mispredict	Instruction Hit, Miss Reads	Instruction lookaside Miss
Total Instructions	Cond. Taken	Load: Store, Miss	Total lookaside Miss
Total Ops		Total: Access, Hit, Miss	
Cycles			

2. BACKGROUND

2.1. Compiler

Briefly, a compiler converts an expression written in a way that can be understood by a human into a statement that can be understood by hardware [18]. In computer science, this conversion is usually done from a high-level language to a lower-level language such as assembly, machine code etc. By high-level language, we refer to languages with high intelligibility, designed to be easy to use, for instance by being free of details such as memory management. Technically, purification is the most basic criterion to determine whether a language is high-level. The compiler is responsible for making this language easier to execute by a machine, at the cost of lesser intelligibility by humans [2].

2.2 Interpreters

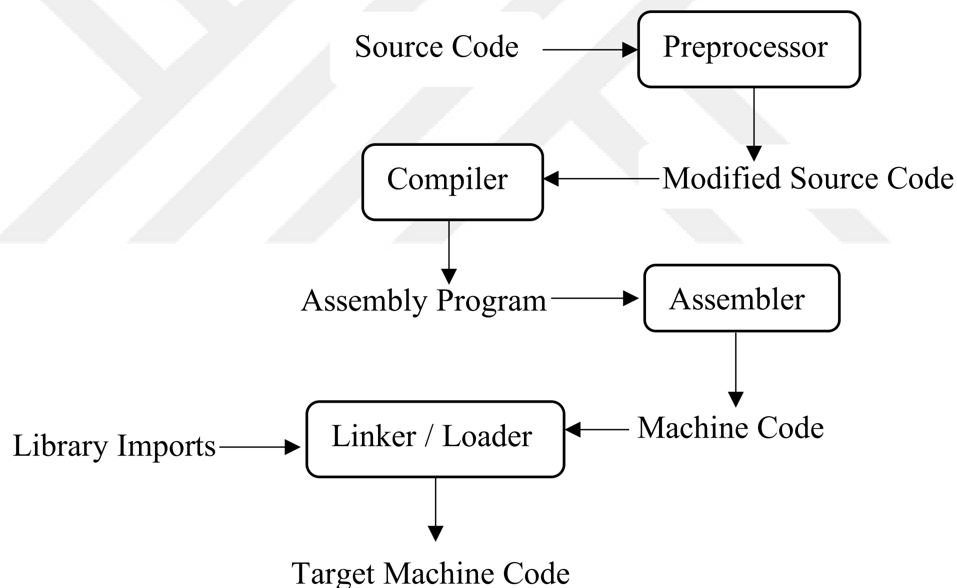


Figure 2.1: From Source Code to Machine Code.

Like Interpreters, Compilers do not deal with the whole code at once, instead they progress step by step in the desired direction and provide results along the way so that the program does not give a generic error in case of errors present in the unused parts of the code. Interpreters usually transform code into middleware such as bytecode or Intermediate Representation. This incremental progression gives better debugging chances than compilers, but also leads to performance loss. Today, commonly used languages such as Python, Perl, or Ruby, are run by Interpreters. Despite significant

differences, Compilers and Interpreters also have a lot in common, such as the ability to tokenize the code and to perform specific analysis. The translation of the code into the target language within the compiler is the result of several steps (see Figure 2.1).

2.3 Optimization

The optimization parameters to be given to the compiler, the main subject of this study, directly affect some of the transformations that will be applied in the conversion to the target code. Most of the programs written in the 1950s contained code written in languages close to assembly level and developed by developers who knew the requirements of the hardware they were working with. While the systems used today are significantly better than the hardware of that period, the number of developers writing low-level code has decreased considerably [18]. As mentioned above, code written in high-level languages relies heavily on the developer's ability to make few mistakes and work efficiently, rather than on hardware compatibility. The optimization difference can be reduced thanks to the optimization flags to be provided to the selected compiler [19]. Selecting the right optimization flags requires considerable experience. There are predefined optimization flags in GCC; version 9.3 contains more than 200 of them. Knowing the role of every single flag is a near impossible task for most developers. In this situation, many software developers get help from predefined optimization flags. Some default flags are shown in Table 2.1. These flags have different effects on the code. A "+" symbol in a column means that the value is greater—for example, in row -O0, the code's Execution Time is longer. This is actually the price to pay for the desired optimization. Increasing the optimization level may be a way to reduce this time, but it should be remembered that a change in the optimization level will lead the values of other metrics to change accordingly. For example, shortening the Execution Time of the code will cause a greater memory usage. This is not a good solution for a system that does not have enough memory. Likewise, the Compile Time of the code also increases over time. Compile Time can be ignored for code that is compiled once and executed many times. Since the main purpose of this study is to shorten the Execution Time of the code, other metrics will be disregarded.

Table 2.1 Effects of Optimization Flags.

Option	Execution Time	Code Size	Memory Usage	Compilation Time
-O0	+	+	-	-
-O1	-	-	+	+
-O2	-		+	++
-O3	--		+	+++
-Os		-		++
-Ofast	--		+	+++

O1: Shortens the execution time of the code without making an optimization that would cause the compilation time to be excessively long.

O2: Applies further optimization than O1. It increases the compilation time as well as shortens the code execution time.

O3: Building up on O2, O3 tries to shorten the execution time of the code by making further optimizations.

Optimization levels progress cumulatively. For example, O2 uses the flags that the O1 optimization level has opened, and adds further optimizations, as seen in Table 2.2. In total, there are 111 flags in Table 2.2, but the figure does not illustrate all options. There are differences according to the GCC version used [20]. GCC 11, for instance, has a total of 232 flags. With the correct selection from these 232 flags, the code can be run faster than with the predefined optimization flags. It must also be noted that using the same flags for every problem or every code does not provide the same optimization; it is therefore necessary to specify a different set of flags for each code.

As mentioned previously, understanding the functions of optimization flags and using them correctly requires expertise. This can get complicated for GCC 11 and the 200+ flags it contains [21]. With the increase in the size of a project, the burden on the person in charge of flag selection also increases. This person will need to keep track of where improvements are made in the structure of the code. This is also a very unefficient and unsustainable situation, especially for a large team.

Table 2.2 Optimization Flags.

O1	O2	O3
auto-inc-dec	align-functions	gcse-after-reload
branch-count-reg	align-jumps	inline-functions
combine-stack-adjustments	align-labels	ipa-cp-clone
compare-elim	align-loops	loop-interchange
cprop-registers	caller-saves	loop-unroll-and-jam
dce	code-hoisting	peel-loops
defer-pop	crossjumping	predictive-commoning
delayed-branch	cse-follow-jumps	split-paths
dse	cse-skip-blocks	tree-loop-distribute-patterns
forward-propagate	delete-null-pointer-checks	tree-loop-distribution
guess-branch-probability	devirtualize	tree-loop-vectorize
if-conversion	devirtualize-speculatively	tree-partial-pre
if-conversion2	expensive-optimizations	tree-slp-vectorize
inline	gcse	unswitch-loops
unctions-called-once	gcse-lm	vect-cost-model
ipa-profile	hoist-adjacent-loads	version-loops-for-strides
ipa-pure-const	inline-small-functions	
ipa-reference	indirect-inlining	
ipa-reference-addressable	ipa-bit-cp	
merge-constants	ipa-cp	
move-loop-invariants	ipa-icf	
omit	ipa-ra	
rame-pointer	ipa-sra	
reorder-blocks	ipa-vrp	
shrink-wrap	isolate-erroneous-paths-dereference	
shrink-wrap-separate	lra-remat	
split-wide-types	optimize-sibling-calls	
ssa-backprop	optimize-strlen	
ssa-phiopt	partial-inlining	
tree-bit-ccp	peephole2	
tree-ccp	reorder-blocks-algorithm=stc	
tree-ch	reorder-blocks-and-partition	
tree-coalesce-vars	reorder-functions	
tree-copy-prop	rerun-cse-after-loop	
tree-dce	schedule-insns	
tree-dominator-opts	schedule-insns2	
tree-dse	sched-interblock	

Table 2.2: Optimization Flags (continued)

O1	O2	O3
tree	sched-spec	
orwprop	store-merging	
tree	strict-aliasing	
re	thread-jumps	
tree-phiprop	tree-builtin-call-dce	
tree-pta	tree-pre	
tree-scev-cprop	tree-switch-conversion	
tree-sink	tree-tail-merge	
tree-slsr	tree-vrp	
tree-sra		
tree-ter		
unit-at-a-time		

Selecting the right flag set poses two challenges. The first one, already mentioned, is to select the right flags from the entire pool (232 flags for GCC 11) to compose the most appropriate flagset. The second one is to provide these selected flags to the compiler in the correct order. The order of the selection of the optimization flags is important as it will affect the execution time of the code, as revealed in the study of Ashouri *et al*. [16]. This situation, known as “Phase-Ordering” in the literature, will not be further investigated in the framework of this thesis, whose scope is primarily focused on the selection of the the right flags.

2.4 Selecting the Best Flag Sequence

The selection of the appropriate flags for the code whose runtime we aim at reducing creates a cluster that contains a certain number of flags. As mentioned above, the order of the flags to be provided to the compiler from this set will affect the optimization structure of GCC. Ashouri *et al*. [8] have shown that these flags will affect the result even without phase-ordering.

2.4.1 Optimization Space

For the compiler, a flag can have two states: On or Off. This will be represented by a 0 or a 1 on the genetic algorithm, as shown in formula 2.4.1.

$$SelectionSet = \{0, 1\}$$

(2.4.1.)

A marker was needed to indicate whether the optimization flags were toggled on or off. Markers prefixed with “fno” are represented as closed, and markers prefixed with “f” are represented as open. In the GCC *-O2 -fno -unsafemath -optimizations -finline -functions* statement, the *-unsafe -math -optimizations* flag needs to be turned off, and inline-functions needs to be turned on. As mentioned above, this is a binary search problem. If ten flags were used, the probability of selection best optimization flag sequence would be close to like 2^{11}. Considering that recent GCC versions have more than 200 flags, this probability quickly grows to exponential levels, as each additional flag doubles the total search set. There are also parameter fields in the flags. These flags, which accept values in a certain range, will not be discussed in this thesis. While developing EAFT, tests and necessary developments have been made for the optimization flags that will work on GCC.

2.5 Benefits and Problems

Compiler optimization may not make as noticeable a difference as the development of CPU structures used to make. Within the scope of the thesis, this subject is also discussed and examined. Before providing the reader with the results of our research, we will present the benchmark used in this study, Polybench.

2.5.1 Polybench

Polybench is a benchmark tool written in C language with different problems, and that contains different predefined input options [22]. These 30 codes perform different operations such as image processing, physics simulation, dynamic programming, and statistical operations. The benchmark’s details are presented in Table 2.5.1. Polybench was selected because it is already used in several publications, especially for compiler optimization. Since the upper limits of the loops can be defined by the user, the latter can directly determine the working time. This gives the user the

option to try the benchmark with different inputs, which means that it is useful to evaluate more than one state of an algorithm, and it thus leads to a good optimization series.

2.5.2 Optimization Effects on Compiler

In this section, we will examine the effectiveness of the optimization levels on the benchmark. A review by Ezhil *et al*. [23] gives a preview of the effectiveness of optimization flags on the cBench benchmark. Let us dig a little deeper into this with Polybench. In the study by Perez *et al*., the authors experienced a 40% improvement in runtime with optimization flags. In their experiment, they used the GCC version 7.1, a different version than the one used in this study.

Table 2.3 Polybench Benchmarks.

Benchmark	**Description**
2mm	2d Matrix Multiplications (D=A.B; E=C.D)
3mm	3d Matrix Multiplications (E=A.B; F=C.D; G=E.F)
Adi	Alternating Direction Implicit Solver
Atax	Matrix Transposition and Vector Multiplication
Bicg	BicG Sub Kernel of BicGStab Linear Solver
Cholesky	Cholesky Decomposition
Correlation	Correlation Computation
Covariance	Covariance Computation
Doitgen	Multiresolution kernel analysis
Durbin	Toeplitz Solver
Dynprog	Dynamic Programming in 2D
Fdtd-2d	2-D Finite Different Time Domain Kernel
Fdtd-apml	FDTD Using Anisotropic Perfectly Matched Layer
Gauss-filter	Gaussian Filter
Gemm	Matrix-Multipl C=alpha.A.B+beta.C
Gemver	Vector Multiplication and Matrix Addition

Table 2.3 Polybench Benchmarks (continued)

Gesummv	Scalar, Vector and Matrix Multiplication
Gramschmidt	Gram-Schmidt Decomposition
Jacobi-1d	1-D Jacobi Stencil Computation
Jacobi-2d	2-D Jacobi Stencil Computation
Lu	LU Decomposition
Ludcmp	LU Decomposition
Mvt	Matrix Vector Product and Transpose
Reg-detect	2-D Image Processing
Seidel	2-D Seidel Stencil Computation
Symm	Symmetric Matrix-Multiply
Syr2k	Symmetric Rank-2k Operations
Syrk	Symmetric Rank-k Operations
Trisolv	Triangular Solver
Trmm	Triangular matrix-multiply

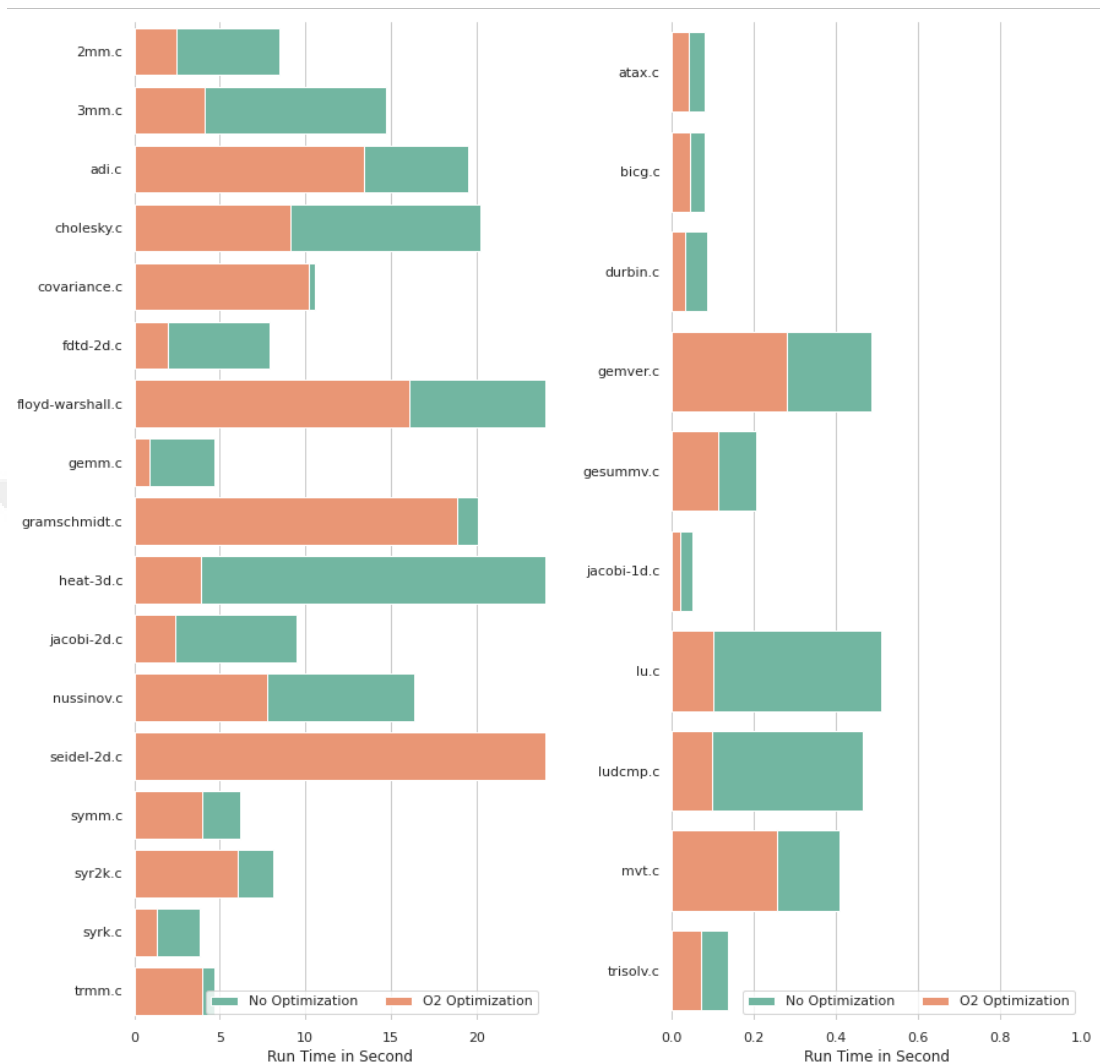


Figure 2.1: O2 Optimization Effects of Polybench.

The graph on Figure 2.2 shows the difference in compilation runtime under normal conditions and with O2 optimization; it reveals that the optimization levels did not provide the same improvement in runtime for each code. In this study, we attempted to reduce the measurement errors by running each benchmark five times with the same optimization flags, substracting the highest and lowest values, and taking into account the average of the remaining three values. In this way, the compilation runtime differences that may occur because of the specifications of the computer on which the measurement is made are reduced to a minimum. In Figure 2.2, the codes with relatively high operating times (between 0 and 20 seconds) are shown on the left side, while the other codes with a runtime between 0 and 1 second are on

the right side. Seidel-2d, trmm and covariance optimization markers were least affected by optimization, while the runtimes of 3mm, head-3d, and fdtd-2d codes were greatly shortened. As mentioned in the Optimization section (section 2.3), shortening the runtime of the code by increasing its optimization levels has side effects. For example, O3 optimization level reduces the code's runtime more than O2, while significantly increasing the size of the code [24]. Since all optimization levels have trade-offs like increasing compilation time when try to decrease run time of a given code, within the scope of this study the results will be compared exclusively with the O2 optimization level.

2.5.3 Comparison between O2 and O3

In the previous section, the effects of the O2 optimization marker on the Polybench data were examined. Does O3 optimization always provide a lower runtime when compared to O2 optimization, which is deemed to be the most effective by GCC ? The answer to this question is to be found in Figure 2.3. The formula

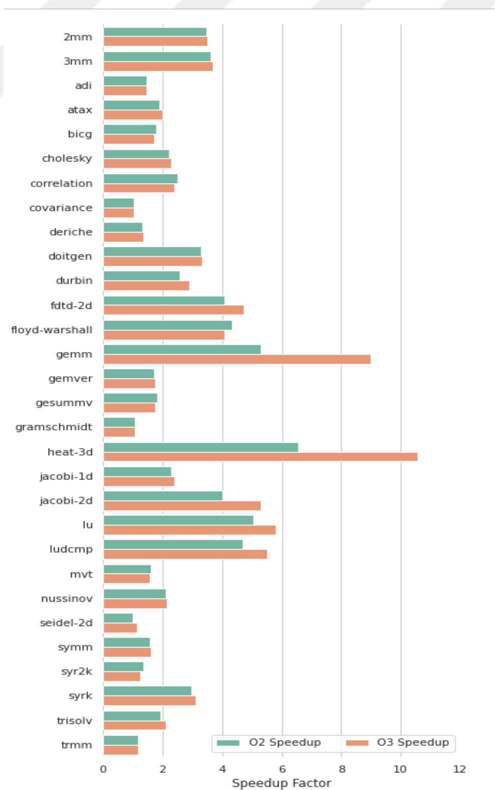


Figure 2.2 O2 vs O3 Optimization Levels.

$SpeedupFactor = \frac{t_{O2}}{t_{runtime}}$ can be used to express the results of the O2 optimization level displayed on Figure 2.3. Likewise, the speedup factor was also calculated for O3.

As shown on Figure 2.3, the O2 optimization marker gave better results for 9 out of 30 benchmark codes. For most of the remaining benchmarks, the difference in runtime was barely noticeable. This is the case with the basic optimization levels. A possibly clearer way to examine the differences in runtime is to calculate how much the optimization speeds up each code individually—this is shown on Figure 2.4. The green dots in the graph show the runtime of the code without optimization, while the orange dots show the O2 and the blue dots, the O3 runtime. The Y axis in Figure 2.4 also represents the runtime. Code such as seidel-2d, already tackled above when

Figure 2.3 O2 vs O3 For Each Benchmark.

examining Figure 2.3, could not be significantly optimized. Several factors could account for this, but one of the most straightforward reason is that the optimization flags do not work effectively because of the complex structure of the code. This situation is also mentioned in the Cole study [21]. Upon examination of the runtimes through the Polybench codes, we saw that for some codes the predefined optimization levels were insufficient. In line with this inadequacy, we aim at finding a better optimization by turning on and off the optimization flags in the optimization levels, rather than using the GCC predefined optimization levels. In order to find the most appropriate flags, a genetic algorithm, which is one of the heuristic search algorithms, will be used. The examination of the effectiveness of heuristic search algorithms, or the way the benchmarks runtimes in Polybench can be reduced with the specified flags, will make for a good introduction before moving on to the explanation of the study itself. But first, the distribution of the runtime for a few benchmarks will be examined using violin plots (Figure 2.5).

It is necessary to give an important detail here. The graphs in Figures 2.5 represent the optimization flags deemed the most useful according to previous studies [4].

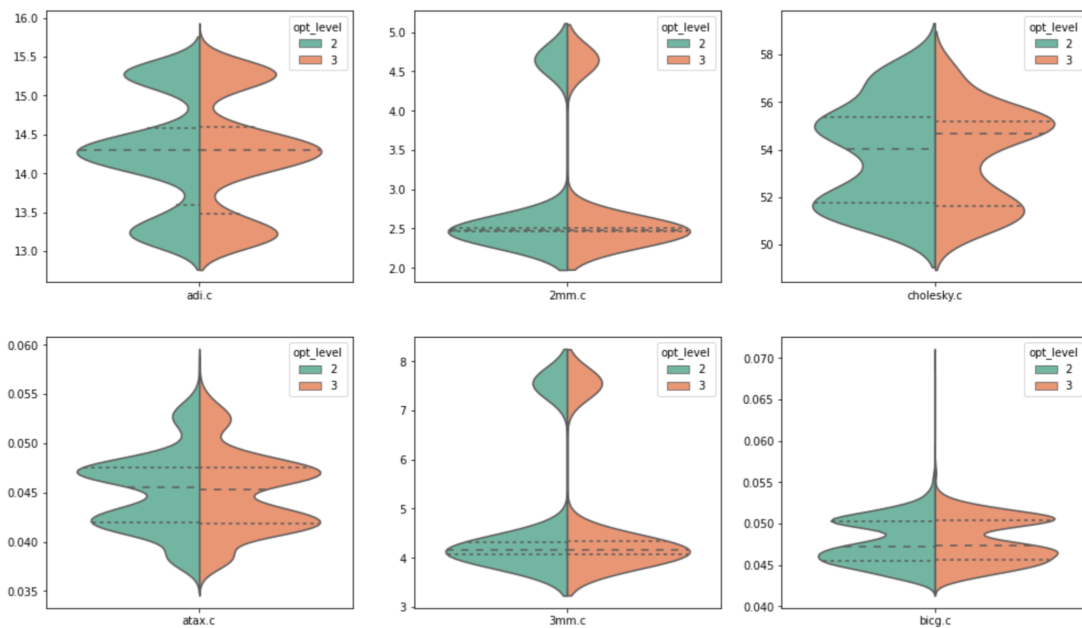


Figure 2.4 Runtime Distribution of Benchmarks with Flags.

Figure 2.5 shows the extent to which the code can be improved when all possibilities are taken into account. Each graph shows the number of codes included in the O2 and O3 optimization levels, and which ones runs faster. Unfortunately, it is not always possible to consider all the possible combinations of code and optimization flags, as shown in Figure 2.6. This can be explained by the multiplicity of optimization levels, as well as by the length of code runtime. For example, running all combinations of eight optimization flags at O2 level for a code whose average working time is 10 minutes would take up to 42 hours—almost two days.

Figure 2.6. gives the distribution of all possible combinations of flags. We expect the distribution of optimizations made using O2 and O3 to be very similar. It was shown in Table 2.2 that the optimization markers also progress cumulatively, and that some of the flags that are added to O2 are actually equal to those of the O3 optimization level. Differences between the distributions is expected too, since all of the optimization markers tried here are not equal to all flags that O2 and O3 have in common. This situation is rarely witnessed simply because each flag does not have the same effect on every code; that is the reason why the distribution in the Cholesky code differs drastically from the others. If we consider the 2mm code, we see that while it peaks at 5 seconds, it is most of the time running in 2.5 seconds

Figure 2.6 Runtime Distribution of All Benchmark with Flags.

3. EVOLUTIONARY ALGORITHMS

Several evolutionary algorithms were implemented in EAFT; they will be examined in this chapter.

3.1 Genetic Algorithm

3.1.1 Inspiration and First Look

Many systems are made by imitating living beings in nature. One of them, in the field of computers science, is the evolutionary algorithms that imitate the evolutionary development processes of living beings, such as the reactions of living beings to the events in nature [25], and use it as a tool. Researchers interested in the adaptation processes of living beings to their environment have imitated natural selection and turned it into a search algorithm [26]. It is obvious that in imitating this evolutionary process, researchers were inspired by Darwin's theory of evolution [27]. The survival of living beings in their natural environment or the changes they undergo to pursue their goals also forms the basis of the genetic algorithm. We will illustrate our meaning through the example of the polar bear. Polar bears lived in polar regions with their black fur in the early periods [28]. Because of this, they were strongly contrasting with the color of the ice, making them more easily distinguishable than other living beings whose predominant color was white. If we assume that the most important factor of survival for polar bears, is the objective function, we can postulate that this factor is low for black bears. Let us suppose that one of them caused its fur to turn white as a result of a mutation in its genetic structure; presumably this change increased his chances of survival. In this case, this white bear, which may live longer than its competitors, will have a higher chance of mating, and the offspring from this bear will resemble its ancestors and also be white, with similarly longer chances of survival than the other bears whose fur remained black [29]. Another example, using the finch birds described in Darwin's Beagle diaries and later included in his works, may bring further clarification. The beaks of the finches on other islands took a different from the beaks of the birds on the mainland, and this change was brought about by the need to access different types of food sources [30].

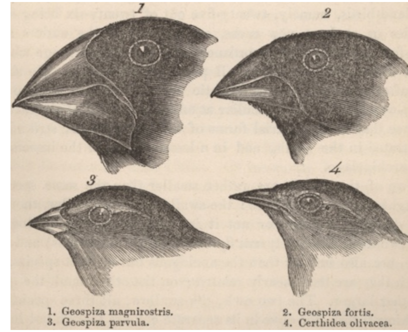


Figure 3.1 Finches Beak [3]

The Darwin's work, can be thought of as the ability of living beings to find food or improve their chances of survival with adapting environment. This underlies the genetic algorithm that seeks to find solutions to very different problems, and for this reason, genetic algorithm is used in machine learning, optimization, and solving problems [31]. Although other algorithms exist—for instance evolutionary algorithms—the most popular algorithm is the genetic algorithm [32]. The genetic algorithm helps us find the best solution within a very large search set, without having to go through every single option. This will save time, as explained in the Optimization section (section 2.3). A consequence of the genetic algorithm's constantly producing better results is also something that humanity has used for thousands of years: crossover. The fact that the offspring produced by two individuals has unique characteristics, with genes inherited from both parents, gives each living being a unique place in the population. Each feature given by the parents to the offspring is called a gene, and a set of these genes is called a chromosome [32]. The characteristics given by these genes affect the behavior of an individual or creature, its reactions in a given environment, its chances of survival, and many other aspects. These features affect the Fitness value, which is what we seek to improve. In this thesis, we will determine which genes are the best to calculate a shorter code runtime; for this, we will sequence the genes from the individual resulting from the crossover. Each gene can take the value of 0 or 1, and the chromosome structure formed by all the genes represents the optimization flags in the GCC. The aspect of this structure is illustrated in Figure 3.2. Each chromosome represents an individual, and a assemblage of chromosomes is called a population. The representation of chromosome and gene is illustrated in Figure 3.3.

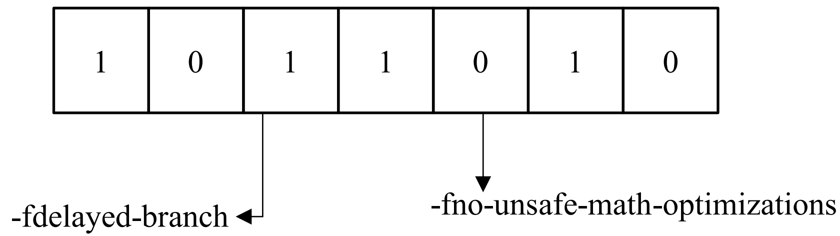


Figure 3.2 Optimization Flag Representation in Chromosome.

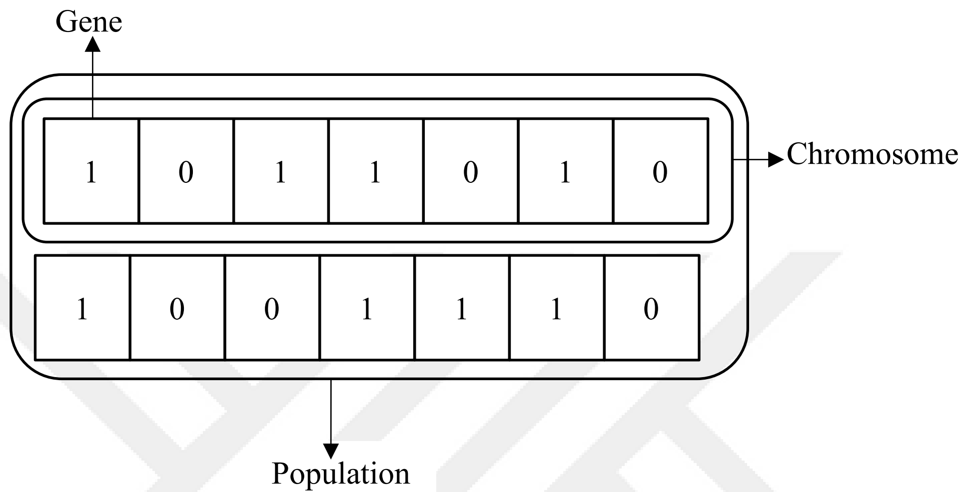


Figure 3.3 From Gene to Population.

3.1.2 Crossover and Mutation

So far, we have considered the origins of the genetic algorithm rather than the way it works. Crossover is the method that determines how the offspring of two individuals will receive its genes. The quality of the method is further improved by the application of numerical methods, such as Average Crossover, that consider the average of the gene values from the parents [33]. This situation is not suitable for the scenario mentioned above, because the values 1 and 0 shown in the representation are used as categories, not as numbers. Another important criterion is the preservation of the uniqueness of the genes as a result of crossover. The absence of a duplicate gene is important for some of the problems. A case study [34] showed that since each gene represents a unique value, the individual created by crossover also preserves this uniqueness. Although this situation is of limited importance in EAFT, a crossover method that preserves the situation has been implemented. The crossover methods implemented in EAFT are presented in the following sections.

3.1.2.1 Partially Mapped Crossover

Partially Mapped Crossover (PMX) is a permutational crossover method in which gene uniqueness is preserved. It is a crossover method that produces two offspring from two parents. It consists of multiple stages, illustrated in Figure 3.4. Step 1: a random range is selected for crossover. This range should not cover the same index value, nor all chromosomes. Later, this value is also selected for the other parent. In step 2, the genes contained in the selected range are swapped. After this substitution, the crossover is not yet complete since there are duplicate genes. In order to eliminate

STEP 1

1	2	3	4	5	6	7	8
2	8	1	4	3	7	5	6

STEP 2

1	2	1	4	3	7	7	8
2	8	3	4	5	6	5	6

STEP 3

1	4	3	7
3	4	5	6

1 -> 3 -> 5
4 -> 4
7 -> 6

STEP 4

5	2	1	4	3	7	6	8
2	8	3	4	5	6	1	7

Figure 3.4 Partially Mapped Crossover.

these duplicate genes, the mapping process, which gives its name to the crossover, is performed (step 3): the chromosomes not included in the initial selection are changed according to this mapping. At the end of the process, in step 4, two offspring are produced.

3.1.2.2 Ordered Crossover

Ordered Crossover (OX), like PMX, is a crossover method where uniqueness is preserved. With this method, one offspring is produced from two parents. The genes to be introduced into the offspring are taken from a random index selected from either parent (step 1). After these genes are transferred, the same genes from parent 2 are discarded, and the remaining genes are transferred to the offspring, in order [35]. An example of this crossover is shown in Figure 3.5.

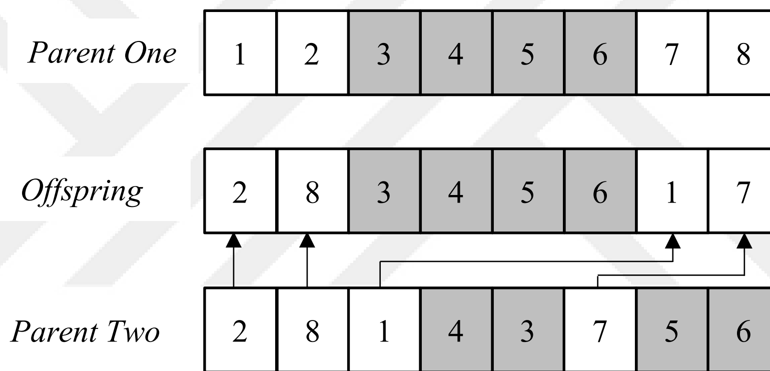


Figure 3.5 Ordered Crossover.

3.1.2.3 Generalized N-Point Crossover

Unlike PMX and OX, N-Point Crossover is not a permutational crossover method; instead, it includes a lighter method of computational power. The crossover starts with a randomly selected number N, which can only be as high as the total number of genes in the chromosome. After the number N is determined, the chromosomes are randomly divided into N parts. After splitting both parents' chromosome, the genes are swapped, thus creating two offspring [36]. As seen in Figure 3.6, the gene 1 occurs more than once in offspring 1. Likewise, genes 3 and 8 appear more than once in offspring 2. This situation is not problematic for EAFT and in the context of this study, because this situation has been implemented especially for the use of different types of crossover methods.

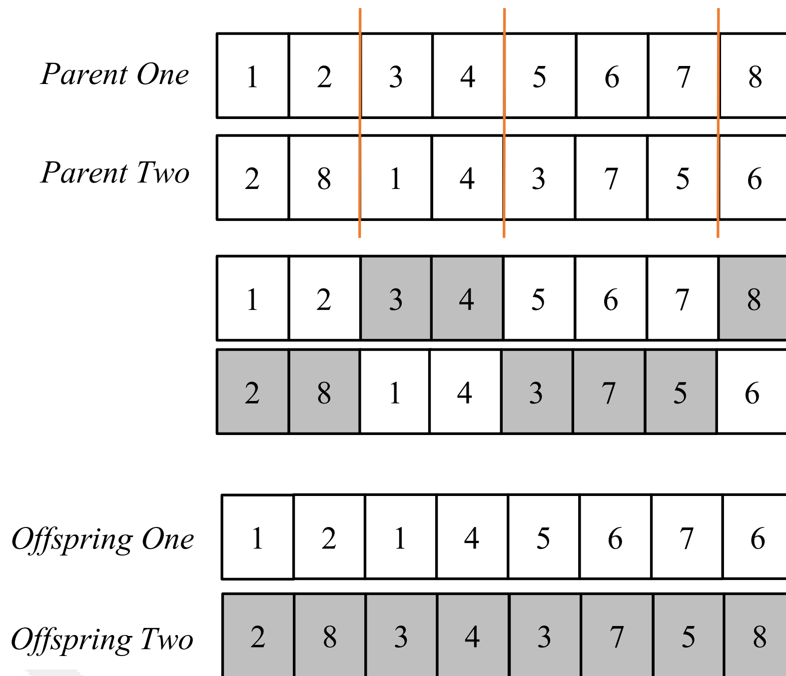


Figure 3.6 Generalized N-Point Crossover.

3.1.2.4 Uniform Crossover

Unlike other methods, Uniform Crossover (UX) uses no index value. The genes to be transferred from each parent are randomly switched to create an offspring with a certain probability [37]. The threshold value can be input as a parameter; it is 0.5 by default.

3.1.2.5 Edge Recombination Crossover

Edge Recombination Crossover (ERX) is a crossover method that aims at minimizing the empty edges, since they affect the performance of the algorithm. The edges between the nodes remain important, as exemplified by the Travelling Salesman Problem (TSP) [33]. In the first step, the neighbors of each gene from both parents are listed. The algorithm starts by selecting a random gene, which is transferred to the offspring. This transferred gene is removed from all other neighbor lists, then the genes in its neighbor list are considered: whichever gene has the least number of neighbors is selected, and in case of equality, a random choice is made [38]. The newly selected gene is added to the offspring and the algorithm continues in the same way until the sequence is completed. A simulation of this crossover method is illustrated in Figure 3.7.

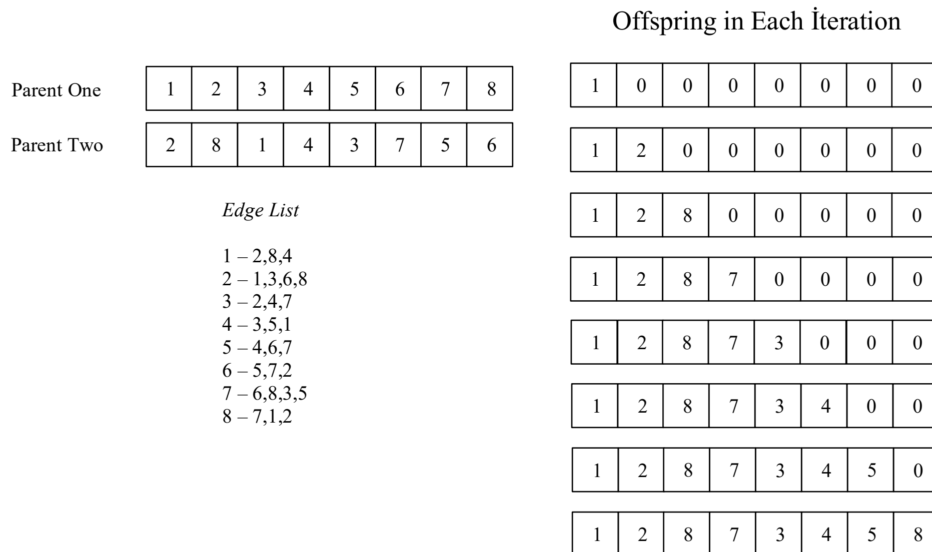


Figure 3.7 Edge Recombination Crossover.

3.1.3 Selection Methods

Selection methods come from the true nature of creatures. Darwin's research showed that selection criteria are a part of evolution [39]. This method or process is known as "Natural Selection." Darwin's works state that differences or variations in phenotypes between individuals are a key element of natural selection. In genetic algorithm, a population comprises many individuals, and some or all of them, depending on the model type, will produce offspring. The questions that need to be asked are: how do we determine which individual will mate with another individual? Should everyone in a population produce an offspring? In any given population, individuals have a different probability of finding mates to produce offspring. According to Darwin, the fittest individuals have a higher chance to find a mate [40]. In genetic algorithm, this probability will depend on two basic things: selection probability and fitness value. In EAFT, two popular selection methods—the "Roulette" and the "Tournament"—have been implemented.

3.1.3.1 Roulette Selection

This selection is based on a real-world game: Roulette [41]. In this game, every number on the roulette wheel has the same probability to come out, but genetic algorithm implementation is a bit different and more in line with Darwin's work, as mentioned above. In this selection method, individuals that have a higher fitness value have a higher degree of agency, or in other words have a higher chance to mate [42].

3.1.3.2 Tournament Selection

Tournament selection is a selection method where tournaments occur between randomly chosen individuals. The algorithm starts with the selection of a number N of individual from a given population. After the selection, each individual's probability to win is determined based on their fitness value. For example, if 5 out of 100 selected randomly in a population. Calculate probability distributions of these individuals and run several tournaments. Number of this tournaments is game changer metric because if the number of tournaments is high, the individuals that have small chances of winning will probably not mate and if it is small this is going to be like random selection from population [43].

In both selection methods, the fittest individual will be the one with the highest theoretical probability of mating.

3.1.4 Mutation

Mutation is based on biological mutation [44]. It ensures that the diversity of chromosomes is preserved, and that genes lost in the process are recovered. Mutation rate is a parameter in EAFT and can be changed in CLI.

3.1.5 Genetic Algorithm Models

In EAFT, more than one genetic algorithm model as well as a wide variety of crossover methods are proposed to the user. By using these models rather than traditional genetic algorithm models, our aim is to find better sequences in the GCC optimization flag set [45]. As in Flag Optimization with Genetic Algorithm (FOGA), only one model was excluded.

3.1.5.1 Generational Model

The Generational Model (GM) is the one of the most common genetic algorithm models in literature [46]; whenever a genetic algorithm is described, it is usually a generational model [47]. Initially, its aim was to produce a number N of offspring and to replace the population with these new children [48] (exceptionally, they could be the first generation if there has been no generation before). The flowchart in Figure 3.8 illustrates the process. In the process of generating new individuals,

selection and crossover are typically performed at the beginning, followed by the application of mutation towards the end.

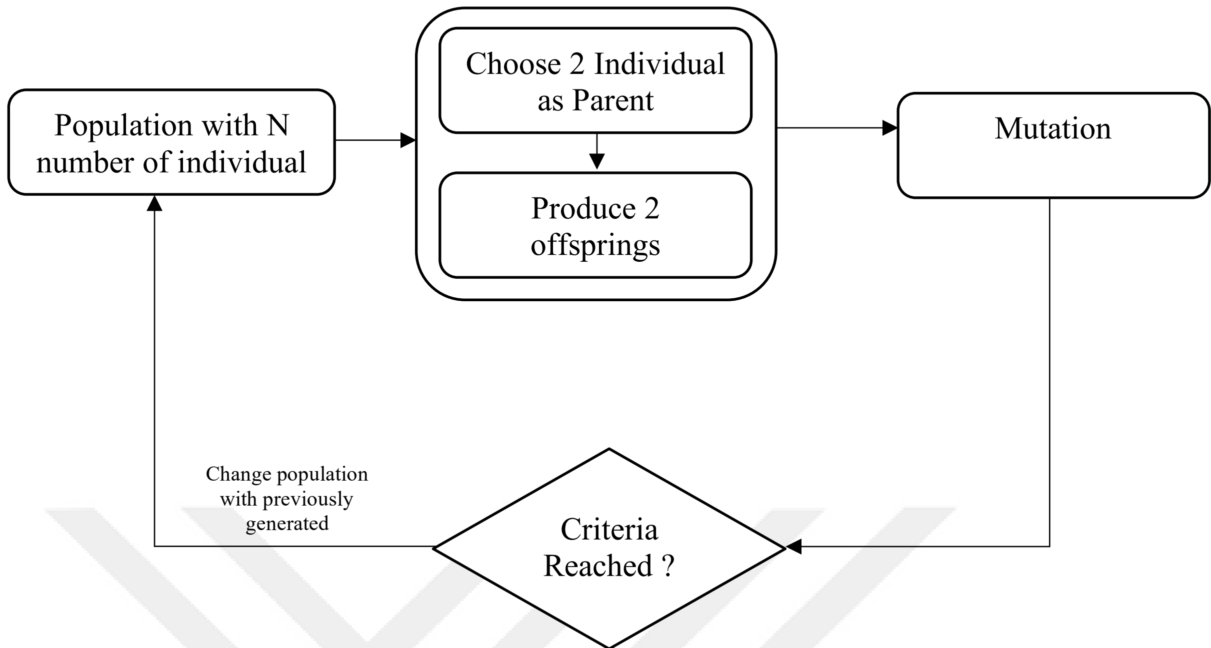


Figure 3.8 Generational Model.

3.1.5.2 Steady State Model

Steady State Model (SSM) is a bit different than the Generational Model. In this model, only one new member joins the population with each iteration [49]. With this implementation, not all new offspring creates a new population since there are many similarities and overlap between children and parents. The other difference is that only two parents are selected in the whole population to generate a new offspring. Not all individuals in the population end up creating a new offspring. After selecting two parents, the crossover is applied and as a result, if two parents generate two children, only the best two will be chosen from this four-member family. This is the main idea behind the Steady State Model [50]. The process is then repeated until the acceptance criteria are reached.

The flowchart in Figure 3.9 shows the additional steps of SSM. As mentioned previously, the new step compares the fitness value of all the members of a family (two offspring and two parents) to find the two best members. A generation population that contains exclusively the new offspring may not be ideal in all situations. In SSM,

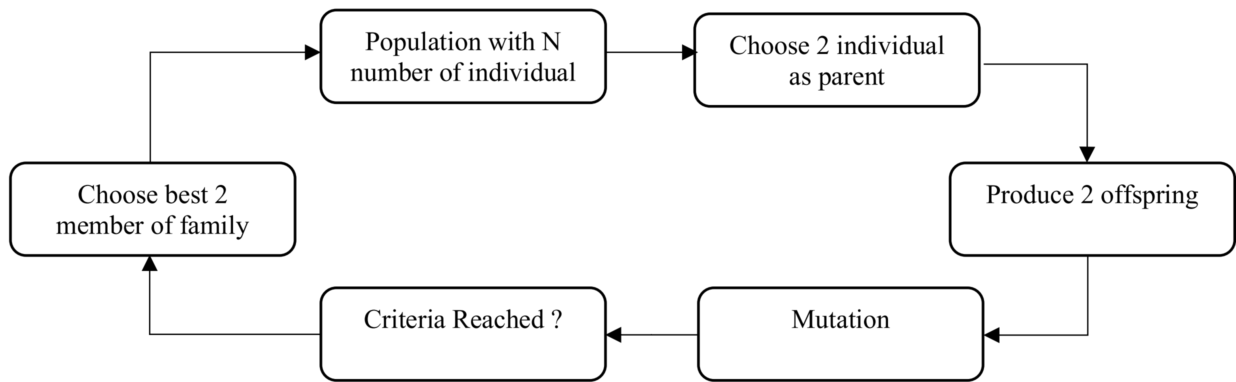


Figure 3.9 Steady State Model.

the population always contains the individuals with the highest fitness value, whereas in GM, the new offspring takes the place of the parents' independently from the fitness value of the individual. Each situation has pros and cons, and the choice of the one to use depends on the end user of EAFT. Some studies have considered the types of problems that could be solved with this type of model, and they have concluded that rule-based systems could use SSM [26]. These two models, as well as the models that will be presented below, are all optional for the end user.

3.1.5.3 Down to Size Model

Down to Size (DTS) model uses different strategies to select the best individuals. In contrast to the other two models, DTS generates two population members. In this model, there are two selection steps. The first step of the model selects the parents who will create the offspring, then counts the offspring as part of the population. If the number N of parents in the population P generates offspring M, the size of population P at the end of this step is M + N. This model leads to the initial population nearly doubling after a round of offspring generation. Following the application of mutation in the genetic algorithm, the newly generated individuals undergo another round of selection, and their acceptance is determined based on predefined criteria. The selection methods are mentioned in selection section. With this method, EAFT gives the end user the choice of selection method. The flowchart in Figure 3.10 illustrates this method's process.

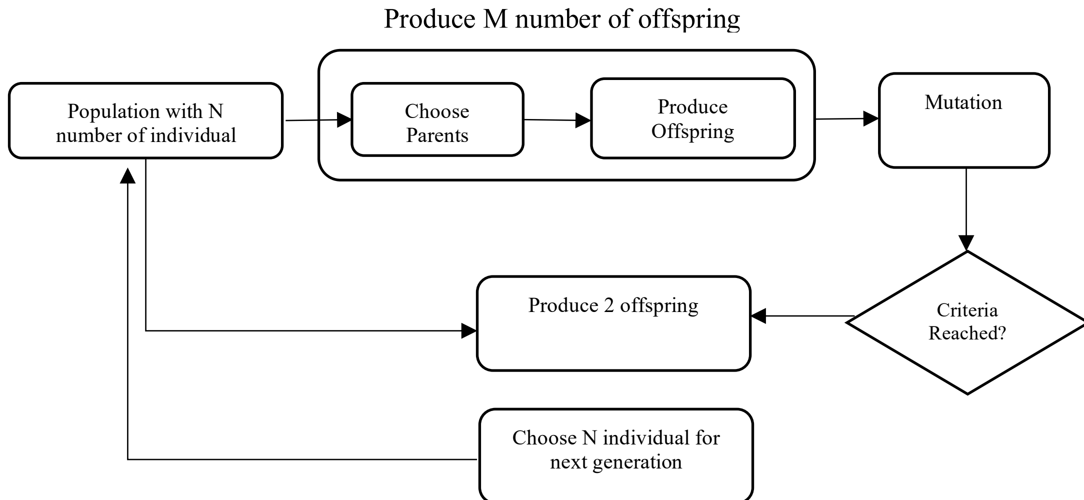


Figure 3.10 Down to Size Model.

3.1.5.4 Ring Model

The Ring Model (RM) is a topological model. In RM, individuals in the population are all neighbors. The selection is completed in a different way than in the other models: whereas other models used selection methods to choose their mates, in this model, an individual can choose its mate only within its neighbors [51]. Figure 5.11 shows how the process works. After an individual is selected, it generates an

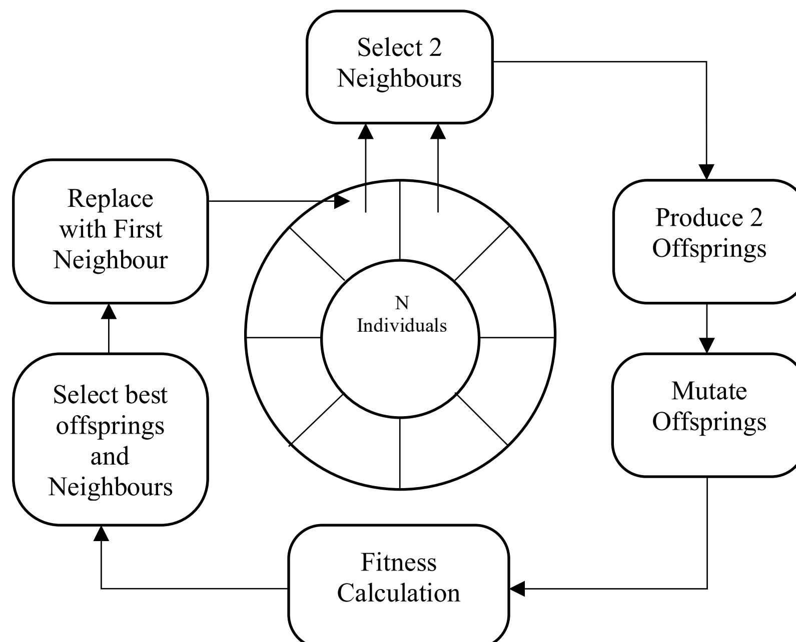


Figure 3.11 Ring Model.

offspring with its neighbor. Then, depending on the fitness value, the offspring may replace the parent. The model then continues until converge its criteria.

3.1.5.5 Only Mutation Model

The Only Mutation model is the simplest of all models. It aims to change individuals only through mutation; there is no crossover in this model. If the mutation rate is increased too much, the model will generate random chromosomes and the individual's fitness values will not converge as desired; for that reason, it is important to choose the correct mutation rate. Conversely, if the mutation rate is too low, then no change will take place in the chromosomes, and the fitness value will not change visibly either.

3.2 Particle Swarm Optimization

Particle Swarm Optimization (PSO) is another evolutionary algorithm option that EAFT users can select. It is also an evolutionary algorithm whose origin is to be looked for in the movements and attitudes of living beings in their natural environment—in particular, it is inspired by the social behavior of bees, fish, or birds [52]. In this algorithm, an individual is represented in the same way as in the genetic algorithm: as a chromosome. A particle is a candidate solution. Each particle represents an optimization flag sequence. In the scientific literature, PSO is mostly used with continuous values but, as mentioned, the compiler optimization problem has a binary representation, and values are always binary. Further, some PSO implementations for binary problems exist, like those presented by Khanesar *et al.* [53]. In EAFT, the Eberhart version of PSO [54] was implemented with binary modifications based on the approach used by Eberhart *et al.* [55].

3.2.1 Formulization of PSO

Unlike in the genetic algorithm, in the PSO the global and personal best values of individuals matter. As formulated by Eberhart [54], each particle is in position $X_i = (x_{i_1}, x_{i_2}, x_{i_3} \dots, x_{i_d})$ and has a velocity of $V_i = (v_{i_1}, v_{i_2}, v_{i_3}, \dots, v_{i_d})$. In this situation, particles' best values are $P_{i_{best}} = (p_{i_1}, p_{i_2}, p_{i_3} \dots, p_{i_d})$ and the global bests are represented as $P_{g_{best}} = (p_{g_1}, p_{g_2}, p_{g_3} \dots, p_{g_d})$. In order to calculate the velocity and position of a particle in a swarm, we use the following formula:

$$v_i(t+1) = w.v_i(t) + c_1\phi_1\big(p_i - x_i(t)\big) + c_2\phi_2\left(p_g - x_i(t)\right)$$

(3.1)

$$x_i(t+1) = x_i(t) + v_i(t+1)$$

(3.2)

c1 and c2 are positive constants, and ϕ1 and ϕ2 are random values between 0 and 1. In addition to (3.1), function (3.3) can be used to calculate the velocity of particles as probability.

$$V_{i_j}(t) = \mathrm{sig}\left(v_{ij}(t)\right) = \frac{1}{1 + e^{-v_{ij}(t)}}$$

(3.3)

$$x_{ij}(t+1) = \begin{cases} 1, & r_{ij} < \mathrm{sig}(v_{ij}(t)) \\ 0, & \text{Else.} \end{cases}$$

(3.4)

r_{ij} is a random value between [0, 1]. According to changes in velocity, function (3.2) will be used to determine the position of a particle, instead of function (3.4).

4. RESULTS

4.1 Command Line Interface

EAFT is a tool designed for end users. It differs from other autotuners in several ways, one of which is CLI, which allows end users to choose options such as the evolutionary algorithm, the crossover, the mutation rate etc. without changing any line of code. The Callback Notification field contains information on the population, such as where to find the best fitness value, or the population's unique ID. A unique ID could then be used to search the population in JSON. EAFT stores all iteration results in a JSON file for analysis or other future use. The algorithm first runs with O2 and O3 optimization levels as baseline and compares each fitness value with these baseline. Figure 4.1 shows the CLI for a population of 5; in that problem, the best fitness value is the same as the lowest (0.21), and has an improvement of 0.41. The values for O2 and O3 are also provided, in the List field. In this problem, the algorithm's O2 runtime is 0.37 second and the best candidate solution for a population of 5 has a runtime of 0.21 second. In order to follow the algorithm's process, the Hall of Fame field was added. It shows the best individual in each population. The last field, Current Fitness Value, provides in real-time the fitness values that have just been calculated. Hall of Fame provides an overall view of the general process, while Current

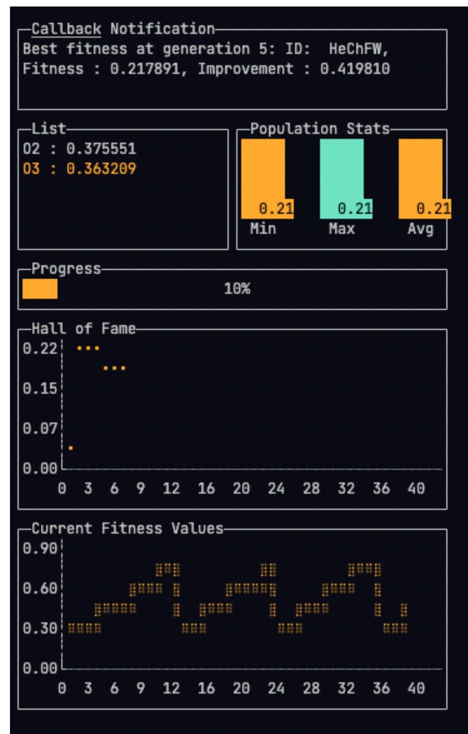


Figure 4.1: CLI.

Fitness Value shows only part of it but in greater detail. Finally, the Population Stats field shows the population statistics.

Table 4.1 Benchmark Results

PROBLEM	**O3**	**GM**	**SSM**	**DTS**	**RM**	**PSO**
2mm	0.9979	0.6028	0.9867	0.9796	**0.5237**	0.7441
3mm	1.0002	0.3709	0.9743	0.9756	**0.3351**	0.3875
deriche	0.9658	0.9779	0.9912	0.9931	**0.9255**	0.9706
cholesky	0.8242	0.8077	0.8458	0.8016	**0.7994**	0.814
jacobi-2d	0.6272	0.2553	0.6297	0.6197	**0.5347**	0.6281
durbin	1.0056	0.8839	0.938	0.8906	**0.832**	0.9019
mvt	1.0088	0.9519	0.9624	0.9735	**0.9028**	0.9746
heat-3d	0.6013	0.4912	0.5189	0.5074	**0.4427**	0.4607
atax	1.0018	0.9671	0.9823	0.9628	**0.9372**	0.9817
doitgen	0.9646	0.5688	0.9809	0.4122	**0.3835**	0.8203
floyd-warshall	1.0862	0.892	0.9573	0.9515	**0.8423**	0.9171
correlation	0.99	0.9405	0.9897	0.9832	**0.8204**	0.9124
gesummv	1.0397	0.9668	0.969	0.9676	**0.8767**	0.9611
bicg	0.9671	0.8289	0.9013	0.9353	**0.759**	0.8304
trisolv	0.964	0.9078	0.9517	0.9532	**0.8833**	0.8912
trmm	0.9895	0.9672	1.0254	0.9641	**0.9541**	0.9766
jacobi-1d	0.9494	0.9241	0.968	0.9467	**0.9228**	0.9308
syr2k	0.7944	0.7324	0.9216	0.803	0.7183	**0.6775**
gemver	1.004	0.9434	0.9694	0.9569	**0.8677**	0.9273
nussinov	0.9894	0.8669	0.9292	0.9186	**0.8554**	0.8763
ludcmp	0.8371	0.6923	0.8302	0.7295	**0.6777**	0.6924
covariance	0.9851	0.7938	0.9884	0.9792	**0.6933**	0.9428
adi	0.94	0.804	0.9435	0.8542	**0.7972**	0.811
syrk	0.8276	0.6804	0.6723	0.7015	**0.653**	0.6686
seidel-2d	0.8834	0.8206	0.8931	0.8293	**0.8012**	0.8279
symm	0.9445	**0.8459**	0.9729	0.8659	0.8627	0.8602
fdtd-2d	0.6852	0.6279	0.687	0.6464	**0.6252**	0.6295
lu	0.8536	0.8374	0.8681	0.8479	**0.7731**	0.8495
gramschmidt	0.9734	0.9473	0.9712	0.96	**0.9366**	0.9387
gemm	0.6142	0.518	0.547	0.5151	**0.4739**	0.5096

Table 4.2: All Flag Opening Results

Problem	*All Flags Runtime*	*Just Level Runtime*	*Level*	*Change Ratio*
adi	18.895	8.814	-O2	2.144
adi	19.006	7.741	-O3	2.455
gramschmidt	6.646	1.852	-O2	3.589
gramschmidt	6.538	1.819	-O3	3.594
deriche	0.471	0.248	-O2	1.896
deriche	0.471	0.257	-O3	1.829
fdtd-2d	2.233	1.325	-O2	1.686
fdtd-2d	2.219	0.944	-O3	2.352
heat-3d	5.634	1.771	-O2	3.181
heat-3d	5.724	1.104	-O3	5.185
2mm	8.057	2.271	-O2	3.548
2mm	8.145	2.269	-O3	3.590
ludcmp	23.075	12.708	-O2	1.816
ludcmp	23.314	9.670	-O3	2.411
trmm	4.362	0.790	-O2	5.519
trmm	4.398	0.806	-O3	5.459
durbin	0.081	0.081	-O2	**1.000**
durbin	0.087	0.080	-O3	1.092
cholesky	22.465	10.948	-O2	2.052
cholesky	21.875	9.349	-O3	2.340
atax	0.160	0.090	-O2	1.780
atax	0.101	0.084	-O3	1.208
floyd-warshall	56.024	10.616	-O2	5.278
floyd-warshall	57.004	10.609	-O3	5.373
jacobi-2d	3.812	1.100	-O2	3.467
jacobi-2d	3.734	0.781	-O3	4.781
correlation	7.364	1.419	-O2	5.191
correlation	7.210	1.421	-O3	5.076
symm	3.267	1.098	-O2	2.977
symm	3.319	1.099	-O3	3.020
3mm	12.343	3.484	-O2	3.543
3mm	12.555	3.473	-O3	3.615
syr2k	4.360	1.172	-O2	3.720
syr2k	4.248	0.765	-O3	5.555
doitgen	3.801	0.603	-O2	6.303
doitgen	3.793	0.592	-O3	6.412
syrk	1.121	0.482	-O2	2.324
syrk	1.093	0.350	-O3	3.120

Table 4.2: All Flag Opening Results (continued)

Problem	***All Flags Runtime***	***Just Level Runtime***	***Level***	***Change Ratio***
doitgen	3.801	0.603	-O2	6.303
doitgen	3.793	0.592	-O3	6.412
syrk	1.121	0.482	-O2	2.324
syrk	1.093	0.350	-O3	3.120
jacobi-1d	0.078	0.085	-O2	**0.909**
jacobi-1d	0.071	0.078	-O3	**0.914**
gemver	0.124	0.100	-O2	1.242
gemver	0.126	0.087	-O3	1.455
mvt	0.130	0.095	-O2	1.372
mvt	0.120	0.091	-O3	1.318
covariance	7.385	1.416	-O2	5.216
covariance	7.218	1.415	-O3	5.102
bicg	0.111	0.102	-O2	1.086
bicg	0.103	0.101	-O3	1.027
nussinov	6.017	1.952	-O2	3.082
nussinov	6.233	1.957	-O3	3.184
gemm	1.400	0.618	-O2	2.265
gemm	1.411	0.351	-O3	4.020
seidel-2d	0.561	0.251	-O2	2.236
seidel-2d	0.561	0.217	-O3	2.579
trisolv	0.089	0.083	-O2	1.067
trisolv	0.098	0.087	-O3	1.118
gesummv	0.096	0.088	-O2	1.091
gesummv	0.093	0.083	-O3	1.123
lu	31.538	12.436	-O2	2.536
lu	31.712	10.785	-O3	2.940

The values in the “All Flags Runtime” column are calculated with the statement *gcc -O2 -fx -fy -fz*, 199 total available flags. The values in the “Just Level Runtime” column are calculated with the statement *gcc -O2.* The values in the “Change Ratio” column are the result of the division of All Flags Runtime by Just Level Runtime.

Upon analysis of the benchmark results (Table 4.1), it appears that the Ring Model found most of the best results and showed the best performance according to the O2 working time. In order to better understand the results, Table 4.2 shows the runtime changes when all the optimization flags are turned on. In all problems but two, turning on the flags resulted in an extension of the code's runtime.

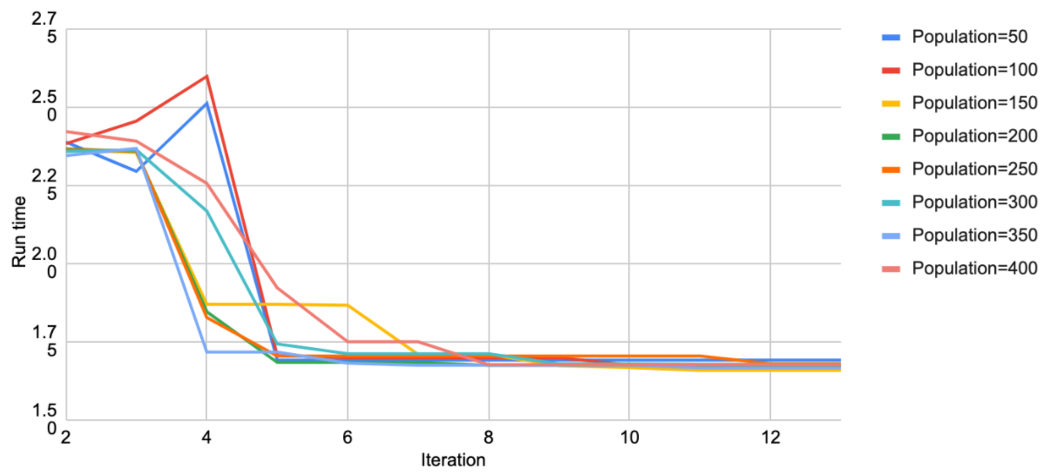


Figure 4.2: 2MM Population Changes on Runtime.

Figure 4.2 shows how different population numbers for the EAFT fine-tuning process of the 2MM Benchmark code affected the optimal code runtime value. After the fourth iteration, the runtime for population values of 50 and 100 showed an increase—an undesirable situation—before decreasing like all the other population values. The 350-population value, unlike the others, reached its lowest value already in the fourth iteration. Although the 200 and 250 population values did not reach their lowest value by the fourth iteration, they still decreased significantly during the study.

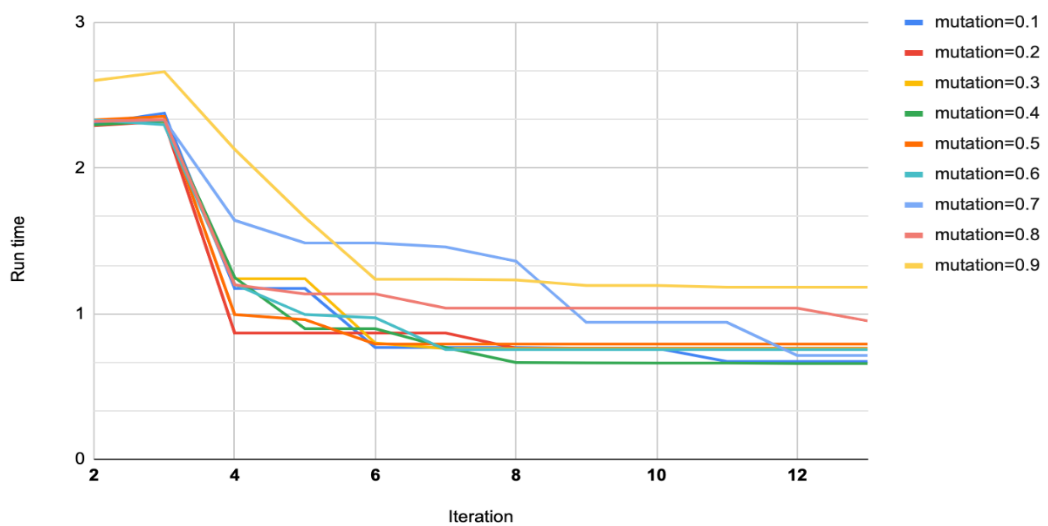


Figure 4.3: Mutation Rate Effect on Run Time for 2MM.

We have also examined the fine-tuning of EAFT in mutation rate. Values between 0.1 and 0.9 are plotted in Figure 4.3: they show how a 0.2 mutation value resulted in a shorter runtime than the other values.

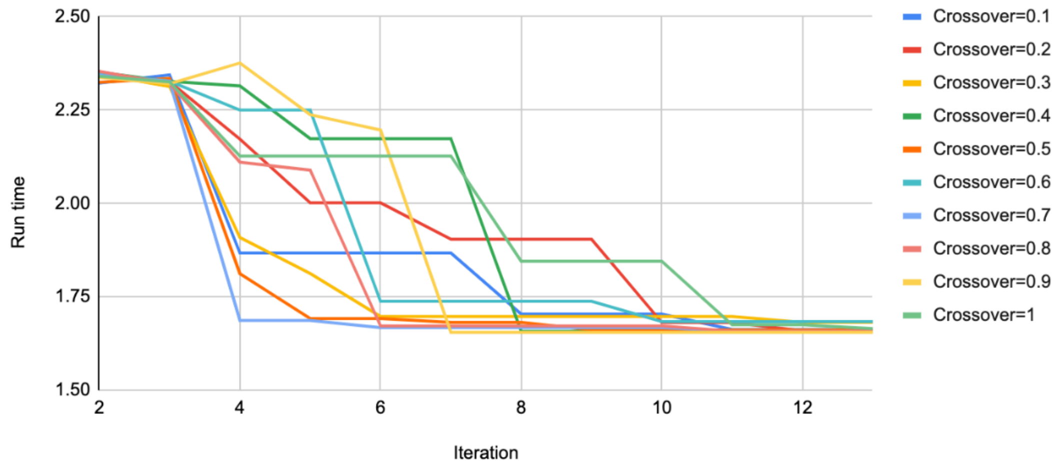


Figure 4.4: Crossover Rate on Run Time for 2MM.

The last parameter examined for EAFT fine-tuning is Crossover Rate. One of the main goals of EAFT is to find the best runtime and to keep it as short as possible. Since the crossover rate determines the possibility individuals have to produce offspring, it also indirectly affects the runtime of EAFT. Although most results are close to the same runtime value—a Crossover rate of 0.7—further shortening the runtime of EAFT and reaching the optimal value in a relatively shorter time, is preferable than the others.

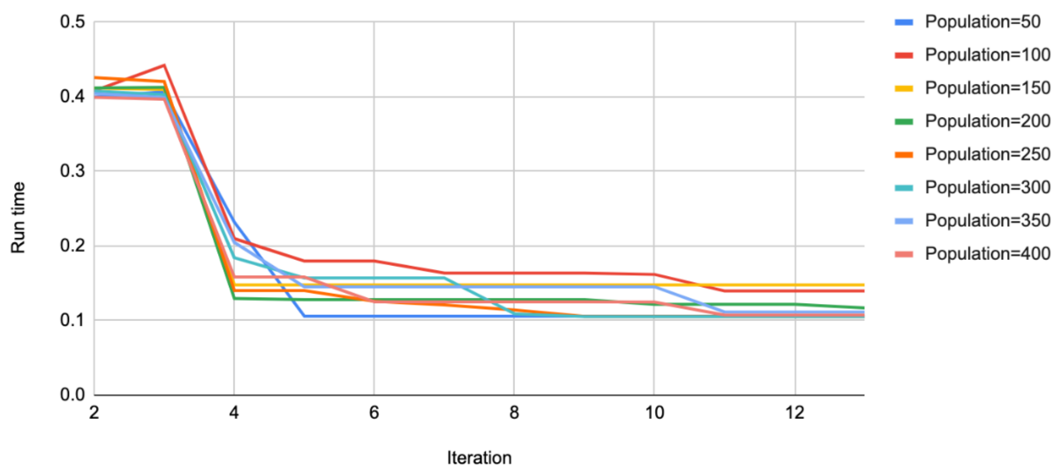


Figure 4.5: Seidel-2D Population Changes.

For the Seidel-2D code, we tried to fine-tune the Population Number, as shown in Figure 4.5. Although the 200 and 250 population numbers seem more reasonable in terms of use, a population of 50 reached a lower runtime in an earlier iteration than the other values. This situation may change depending on the structure of the code and is interpreted as a very unreasonable situation.

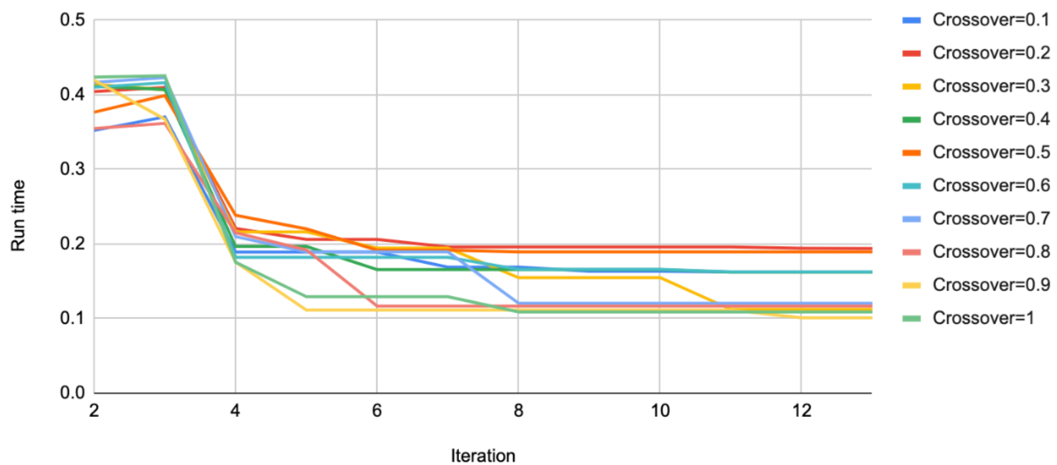


Figure 4.6: Seidel-2D Crossover Rate.

A Crossover Rate fine-tuning has also been tested in the Seidel-2D benchmark. For this problem, 0.9 and 1 crossover values achieved the best results in the fifth iteration, followed the 0.5 crossover value in the following iteration.

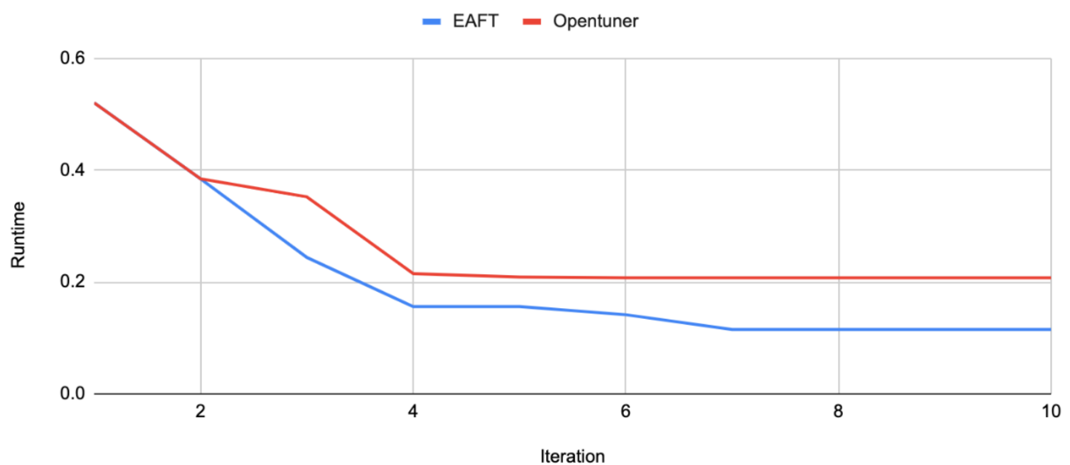


Figure 4.7: EAFT vs Opentuner on Matrix Multiplication Code.

To better understand the results, we compared EAFT and Opentuner, since both do similar work. Figure 4.7 shows how much uptime the Matrix Multiplication

(MM) C++ code in Opentuner obtained with two tools. From the seventh iteration onwards, EAFT and Opentuner progressed in parallel, with EAFT achieving a faster result.

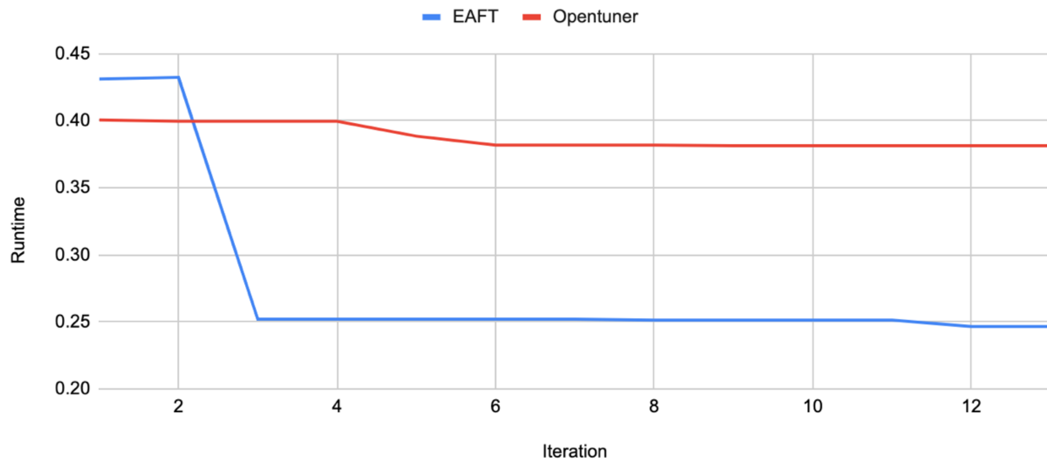


Figure 4.8: EAFT vs Opentuner on TSP Code.

We then compared the runtime of TSP_GA, a C++ code included in Opentuner. With this code, the runtime difference between Opentuner and EAFT was very wide as early as the third iteration, with EAFT achieving a much better result; the results continued to progress in parallel afterwards.

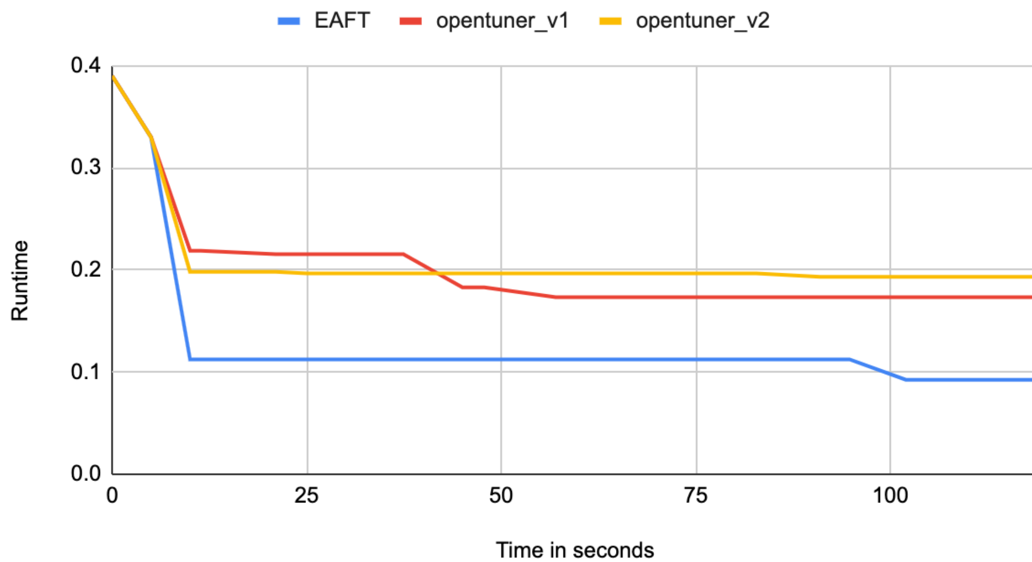


Figure 4.9: Opentuner vs EAFT Time Comparison.

In Figure 4.9, the Opentuner code was run twice for the same problem (TSPGA), named v1 and v2 in the legend. In both versions, EAFT achieved a better result than Opentuner.

5. DISCUSSION

EAFT is a tool for users who wants to shorten the runtime of the programs they use. To best guide the users, the research and implementation presented above was worked on a benchmark. These results are valid exclusively in Polybench. As mentioned in the Compiler section (2.1), most codes are different and no single solution is valid for all types of code. All the results are presented in Table 4.1. To better explain the results, the O2 runtime of all problems in the benchmark was set to 1. Then, all remaining working times of the problems were multiplied by this ratio. In this way, using 2mm as an example, the runtime of O3 was 1% faster than O2's. Likewise, O2 had 48% less uptime than RM O2. The resulting RM thus found is the best value for that problem, and the best results for the other problems are emphasized in bold in the table. When we consider all the problems, it seems that the Ring Model is the model that provides the best result in the vast majority of cases. It is possible that this situation is due to the similarity of the structure of all the benchmarks. As explained before, all problems, although they are very different from each other, are eventually mathematical solutions. Exceptionally, Ring Model did not provide the best result: for instance, PSO found the best result in syrk2k, and GM did so in symm. There were also cases where the O3 runtime was longer than that of O2. For most of the cases, RM had better runtimes than O2. This is a benefit of using several different models. For most problems, other models have achieved results close to those of RM. For 2MM, the difference between the results of GM and RM is minimal. Likewise, the best model for TRMM is only 5% faster than O2.

Table 4.1 shows how much each model improved over the runtime values of O2. The difference between these results and those of O2 is due to the fact that they were recompiled and run with specific combinations of optimization flags. This situation brings up the following question: how fast will the code runtime be if all the flags are turned on? The answer to this question is presented in Table 4.2. Let us first say a word about the columns of the table.. The "All Flags Runtime" column was compiled by opening all the optimization flags in GCC-11, where this experiment was performed, then specifying an optimization level (indicated in the "Level" column) before running the flags. All optimization flags presented here are taken from the results returned after the *gcc-11 --help=optimizers* command, according to the guidelines provided on the GCC official website. The value in the "Change Ratio"

column is simply the result of the value in the “All Flags Runtime” column divided by the value in the “Just Level Runtime” column; this “change ratio” is, in other words, the answer to the following question: “what is the runtime difference, expressed as a ratio, between compiling the codes at O2 and O3 levels with, and without, all optimization flags?” The code compiled with all the optimization flags turned on naturally performed less well than when compiled with the standard O2 or O3 level, prolonging the runtime of the code we were instead trying to shorten. All the codes were slowed down, except for Jacobi 1D and Durbin, the latter only at O2 level. It is not uncommon for the use of all possible optimization flags to result in a slower program than if we had use only the O2 optimization level. This can happen for several reasons:

1. Over-optimization: when all the possible optimization flags are used, the compiler may perform optimizations that end up actually slowing down the program. This is known as over-optimization [56].
2. Optimization conflicts: some optimization flags may conflict with each other, resulting in unexpected behavior and slower program performance.
3. Resource consumption: some optimization flags may require more memory or processing power, leading to slower overall performance.
4. Compiler bugs: using all possible optimization flags may create bugs or issues in the compiler, which can result in slower program performance.

It is important to note that optimization flags can have different effects depending on the specific code being compiled; as mentioned above, using all optimization flags may occasionally result in faster performance, like Jacobi 1D and Durbin.

Fine-tuning Genetic Algorithm is important for further usages. It could improve the performance of Genetic Algorithm to find better solutions in a shorter time. In order to improve the performance of the Genetic Algorithm, it is first necessary to determine which code(s) provide the best improvement from the benchmark. 2MM and Seidel-2d are suitable for this purpose. Afterwards, we need to examine the effects that changing a single feature have on the result, keeping the other features constant. Initially, this situation was examined only to find the best result, while the runtime of EAFT was ignored.

The first experiment with fine-tuning was to change the population number. In this experiment, presented in Figures 4.2 and 4.5, we considered the population

number between 50 and 400 in increments of 50. In this experiment, the values for the Crossover Rate and the Mutation Rate were 0.9 and 0.01, respectively. Increasing the population size could initially lead to better solutions, since there is a higher diversity of individuals and hence the exploration of the search space is increased. However, this improvement in performance is often short-lived, as the population may converge to a suboptimal solution after a certain number of generations. This convergence occurs because of a trade-off between the computational complexity and the diversity of the population. As the population size increases, the computational complexity of the algorithm also increases, which can make it more difficult to efficiently explore the search space. Furthermore, as the population size increases, the probability of generating duplicate individuals also increases, which in turn reduces the diversity of the population. Thus, it is crucial to choose an appropriate population size based on the complexity of the problem and the available computational resources. In general, the optimal population size is problem-specific and can be determined through empirical experimentation. As a result of our trials on Polybench with EAFT, we came to the conclusion that the best result was a population size of 350.

Another experiment examined the effect of changes in the crossover rate. In this experiment, Crossover Rate values between 0.1 and 1 were tried one by one; the results are presented in Figures 4.4 and 4.6. This optimal crossover rate, 0.8, led to the best performance of the Genetic Algorithm. The reason for this optimal crossover rate can be attributed to the balance between exploration and exploitation of the search space. When the crossover rate is too low, the algorithm may converge prematurely and end up not exploiting the good features of different individuals in the population. Conversely, if the crossover rate is too high, it may result in too much exploitation of the search space and a lack of diversity in the population. In our experiments, the optimal crossover rate of 0.7 allowed for sufficient exploitation of the search space while maintaining a diverse population. This optimal rate allowed the algorithm to combine the good features from different individuals in the population, while also avoiding premature convergence toward a suboptimal solution. In conclusion, the optimal crossover rate for a genetic algorithm is problem-specific and can be determined through empirical experimentation. In our case, the best crossover rate of 0.7 allowed for a balance between exploitation and exploration of the search space, resulting in the highest quality solutions.

Finally, the effects of different mutation rates on the performance of EAFT were examined; the results of this experiment, conducted with rates ranging from 0.1 to 1, are presented in Figure 4.3. The results indicate that the best mutation rate for the given problem was 0.2. This optimal mutation rate led to the best performance of the genetic algorithm, as it resulted in the highest quality solutions found during the experiments. The reason for this optimal mutation rate can be attributed to the balance between exploration and exploitation of the search space. When the mutation rate is too low, the algorithm may converge prematurely and end up not exploring new regions of the search space. Conversely, if the mutation rate is too high, it may result in too much random variation and a lack of convergence toward the optimal solution. In our experiments, the optimal mutation rate of 0.2 resulted in a diverse population that allowed for the exploration of new regions of the search space, while also providing enough stability to converge toward the optimal solution. In conclusion, the optimal mutation rate for a genetic algorithm is problem-specific and can be determined through empirical experimentation.

Let us assess the performance of EAFT with the C++ codes included in Opentuner, using the optimal results. We first ran these codes on Opentuner, then ran them again with EAFT, and we examined the graph that shows which code reached the most optimal result. When analyzed for the MatrixMultiply code, Figure 4.7 shows that EAFT and Opentuner initially progressed in close proximity to each other, before diverging in a more pronounced way after the fourth iteration. Opentuner could not find a better result after the fourth iteration. Regarding the TSP_GA code, Opentuner found a better result in the first two iterations, before being overtaken by EAFT from the third iteration onwards. The results are presented in Figure 4.8. Finally, we compared the runtimes of Opentuner and EAFT. Unlike the other experiments, this comparison was not based on iteration but is presented with the runtime of these tools on the x-axis, expressed in seconds. EAFT was able to outperform Opentuner not only in iteration but also in runtime. These results are shown in Figure 4.9.

CONCLUSION AND FUTURE WORKS

In this thesis, a genetic algorithm was employed to optimize the selection of EAFT and GCC optimization flags, with the aim of achieving the best runtime performance for a given code. The algorithm was designed to explore a vast search space to find the most optimal parameters within it. In order to do this, we employed several models and crossover methods that have not been used in other studies. Among these methods, the different Genetic Algorithm models were identified as the most critical. This approach led to the identification of more effective optimization parameters and enabled the algorithm to produce better runtime performances.

One of the most significant challenges encountered in this study was to find the right optimization markers. Optimization markers are metrics used to measure the performance of different optimizations for a given program. Finding an optimization marker that provides consistent results across different benchmarks was difficult. This was due to the fact that the optimal parameters for one benchmark may not necessarily be optimal for another. However, the study fine-tuned the approach to identify the optimal metrics for EAFT. This approach improved the accuracy of the optimization and enabled the algorithm to produce more consistent results.

The values obtained in this study were compared with the results of another library, Opentuner. The results indicated that EAFT produced better results than Opentuner. This finding highlighted the effectiveness of the genetic algorithm approach in optimizing the selection of EAFT and GCC optimization flags. Moreover, the study showed that fine-tuning the approach can lead to better performance results.

Future improvements to this study could involve exploring different genetic algorithm models and crossover methods to improve the efficiency and accuracy EAFT. Expanding the research to cover a broader range of programming languages and architectures could also be worth considering. Furthermore, the study could investigate the impact of different hardware configurations on the performance of the genetic algorithm. Another aim of future work could be the improvement of the output format of EAFT. The tool currently produces output in JSON format, which may not be suitable for all users. Therefore, future work could involve designing an approach to include the results of EAFT into a database. This would enable more flexibility in

the way users can access and analyze the optimization results, and it could facilitate the integration of EAFT into a CI/CD pipeline for the automated optimization of code.

REFERENCES

[1] J. Cavazos, G. Fursin, F. V. Agakov, E. V. Bonilla, M. F. P. O'Boyle, and O. Temam, "," in *Int. Symp. Code Generation and Optimization (CGO'07)*, 2007, pp. 185–197.

[2] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques, and tools*, 2nd ed. Boston, MA, USA: Pearson Addison-Wesley, 2007.

[3] C. Darwin, *The voyage of HMS Beagle*, 1910.

[4] J. Ansel *et al.*, "Opentuner: An extensible framework for program autotuning," in *Proc. 23rd int. conf. Parallel Architectures and Compilation*, 2014, pp. 303–316.

[5] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Int. Symp. Code Generation and Optimization (CGO'04)*, 2004, pp. 75–86.

[6] C. Mendis, A. Renda, S. Amarasinghe, and M. Carbin, "Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks," in *36th Int. Conf. Machine Learning (ICML)*, 2019, pp. 4505–4515.

[7] L. Burkholder, "The halting problem," *ACM SIGACT News*, vol. 18, no. 3, pp. 48–60, 1987.

[8] A. H. Ashouri, G. Mariani, G. Palermo, E. Park, J. Cavazos, and C. Silvano, "Cobayn: Compiler autotuning framework using bayesian networks," in *ACM Transactions on Architecture and Code Optimization (TACO)*, 2016, vol. 13, no. 2, pp. 1–25.

[9] G. Fursin *et al.*, "Milepost gcc: Machine learning enabled self-tuning compiler," *Int. J. Parallel Programming*, vol. 39, no. 3, pp. 296–327, 2011.

[10] E. Park, J. Cavazos, and M. A. Alvarez, "Using graph-based program characterization for predictive modeling," in *Proc. 10th Int. Symp. Code Generation and Optimization*, 2012, pp. 196–206.

[11] Z. Wang and M. O'Boyle, "Machine learning in compiler optimization," in *Proc. IEEE*, 2018, vol. 106, no. 11, pp. 1879–1901.

[12] K. D. Cooper, P. J. Schielke, and D. Subramanian, "Optimizing for reduced code space using genetic algorithms," in *Proc. ACM SIGPLAN 1999 workshop Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 1999, pp. 1–9.

[13] S. Zhong, Y. Shen, and F. Hao, "Tuning compiler optimization options via simulated annealing," in *2nd Int. Conf. Future Information Technology and Management Engineering (FITME)*, 2009, pp. 305–308.

[14] C. Dubach, J. Cavazos, B. Franke, G. Fursin, M. F. O'Boyle, and O. Temam, "Fast compiler optimisation evaluation using code-feature based performance prediction," in *Proc. 4th Int. Conf. Computing Frontiers*, 2007, pp. 131–142.

[15] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," in *Proc. ACM Programming Languages* (POPL), 2019, vol. 3, pp. 1–29.

[16] A. H. Ashouri, G. Palermo, J. Cavazos, and C. Silvano, "The phase-ordering problem: An intermediate speedup prediction approach," in *Automatic Tuning of Compilers Using Machine Learning*. Cham, Switzerland: Springer, 2018, pp. 71–83.

[17] A. H. Ashouri, G. Mariani, G. Palermo, and C. Silvano, "A bayesian network approach for compiler auto-tuning for embedded processors," in *2014 IEEE 12th Symp. Embedded Systems for Real-Time Multimedia (ESTIMedia)*, pp. 90–97.

[18] W. M. Waite and G. Goos, *Compiler construction*. New York, NY, USA: Springer, 1984.

[19] D. Whitfield and M. L. Soffa, "An approach to ordering optimizing transformations," in *Proc. 2nd ACM SIGPLAN Symp. Principles & Practice of Parallel Programming*, 1990, pp. 137–146.

[20] J. Chen, N. Xu, P. Chen, and H. Zhang, "Efficient compiler autotuning via bayesian optimization," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pp. 1198–1209.

[21] K. A. Hoste and L. Eeckhout, "Cole: compiler optimization level exploration," in *Proc. 6th annu. IEEE/ACM int. symp. Code Generation and Optimization*, 2008, pp. 165–174.

[22] T. Yuki, "Understanding polybench/c 3.2 kernels," in *Int. workshop Polyhedral Compilation Techniques (IMPACT)*, 2014, pp. 1–5.

[23] P. Ezhil *et al*., "Experimental analysis of optimization flags in GCC," *Turkish J. Computer and Mathematics Education (TURCOMAT)*, vol. 12, no. 7, pp. 1875–1879, 2021.

[24] M. T. Jones, "Optimization in GCC," *Linux J.*, vol. 2005, no. 131, p. 11, 2005.

[25] X. Yu and M. Gen, *Introduction to evolutionary algorithms*. London, U.K.: Springer, 2010.

[26] J. H. Holland, "Genetic algorithms," *Scientific American*, vol. 267, no. 1, pp. 66–73, 1992.

[27] S. Mirjalili, "Genetic algorithm," in *Evolutionary Algorithms and Neural Networks: Theory and Applications*. Springer, 2019, pp. 43–55.

[28] B. Kurtén, “The evolution of the Polar Bear, *Ursus maritimus* Phipps,” *Acta zoologica Fennica*, vol. 108, pp. 3–30, 1964.

[29] J. A. Cahill, “Polar bear taxonomy and evolution,” in *Ethology and Behavioral Ecology of Sea Otters and Polar Bears*, R. W. Davis and A. M. Pagano, Eds. Cham, Switzerland: Springer, 2021, pp. 207–218.

[30] D. Lack, *Darwin's finches*. Cambridge, U.K.: Cambridge Univ. Press, 1983.

[31] T. Bäck and H.-P. Schwefel, “An overview of evolutionary algorithms for parameter optimization,” *Evolutionary Computation*, vol. 1, no. 1, pp. 1–23, 1993.

[32] S. Mirjalili, J. Song Dong, A. S. Sadiq, and H. Faris, “Genetic algorithm: Theory, literature review, and application in image reconstruction,” *Nature-Inspired Optimizers: Theories, Literature Reviews and Applications*, S. Mirjalili, J. Song Dong, and A. Lewis, Eds. Cham, Switzerland: Springer, 2020, pp. 69–85.

[33] A. J. Umbarkar and P. D. Sheth, “Crossover operators in genetic algorithms: a review,” *ICTACT J. Soft Computing*, vol. 6, no. 1, pp. 1083–1092, 2015.

[34] B. Tağtekin, M. U. Öztürk, and M. K. Sezer, “A case study: Using genetic algorithm for job scheduling problem,” 2021, *arXiv*:2106.04854.

[35] P. Moscato *et al.*, “On genetic crossover operators for relative order preservation,” *C3P Report*, vol. 778, p. 825, 1989.

[36] W. M. Spears and K. A. De Jong, “An analysis of multi-point crossover,” in *Foundations of Genetic Algorithms*, G. J. E. Rawlins, Ed. San Mateo, CA, USA: Morgan Kaufmann Publishers, 1991, pp. 301–315.

[37] G. Syswerda *et al.*, “Uniform crossover in genetic algorithms,” in *Proc. 3rd Int. Conf. Genetic Algorithms (ICGA)*, 1989, vol. 3, pp. 2–9.

[38] Z. H. Ahmed, “Genetic algorithm for the traveling salesman problem using sequential constructive crossover operator,” *Int. J. Biometrics & Bioinformatics (IJBB)*, vol. 3, no. 6, p. 96, 2010.

[39] C. Darwin, *Charles Darwin's natural selection: being the second part of his big species book written from 1856 to 1858,* R. C. Stauffer, Ed. London, U.K. and New York, NY, USA: Cambridge Univ. Press, 1987.

[40] G. J. Balady, “Survival of the fittest—more evidence,” *The New England J. of Medicine*, vol. 346, no. 11, pp. 852–854, 2002.

[41] N. M. Razali and J. Geraghty, “Genetic algorithm performance with different selection strategies in solving tsp,” in *Proc. World Cong. Engineering 2011 (WCE)*, vol. 2, Hong Kong, China: International Association of Engineers, pp. 1–6.

[42] A. Lipowski and D. Lipowska, “Roulette-wheel selection via stochastic acceptance,” *Physica A: Statistical Mechanics and its Applications*, vol. 391, no. 6, pp. 2193–2196, 2012.

[43] B. L. Miller, D. E. Goldberg, “Genetic algorithms, tournament selection, and the effects of noise,” *Complex Systems*, vol. 9, no. 3, pp. 193–212, 1995.

[44] M. Kumar, D. Husain, N. Upreti, and D. Gupta, “Genetic algorithm: Review and application,” *Int. J. Information Technology and Knowledge Management,* vol. 2, no. 2, pp. 451–454, 2010.

[45] B. Tağtekin, B. Höke, M. K. Sezer, and M. U. Öztürk, “Foga: Flag optimization with genetic algorithm,” in *2021 Int. Conf. INnovations in Intelligent SysTems and Applications (INISTA)*, pp. 1–6.

[46] M. Mitchell, *An Introduction to Genetic Algorithms*. Cambrige, MA, USA: MIT Press, 1998.

[47] H. G. Cobb and J. J. Grefenstette, “Genetic algorithms for tracking changing environments,” in *Proc. 5th Int. Conf. Genetic Algorithms*, S. Forrest, Ed., 1993, pp. 523-530.

[48] G. Syswerda, “A study of reproduction in generational and steady-state genetic algorithms,” in *Foundations of genetic algorithms*, G. J. E. Rawlins, Ed. San Mateo, CA, USA: Morgan Kaufmann Publishers, 1991, pp. 94–101.

[49] F. Vavak and T. C. Fogarty, “Comparison of steady state and generational genetic algorithms for use in nonstationary environments,” in *Proc. IEEE Int. Conf. Evolutionary Computation*, 1996, pp. 192–195.

[50] D. Whitley and T. Starkweather, “Genitor ii: A distributed genetic algorithm,” *J. Experimental & Theoretical Artificial Intelligence*, vol. 2, no. 3, pp. 189–214, 1990.

[51] D. Whitley, S. Rana, and R. B. Heckendorn, “Island model genetic algorithms and linearly separable problems,” in *AISB Int. Workshop Evolutionary Computing*. Berlin, Germany; Heidelberg, Germany: Springer, 1997, pp. 109–125.

[52] R. Poli, J. Kennedy, and T. Blackwell, “Particle swarm optimization,” *Swarm Intelligence*, vol. 1, no. 1, pp. 33–57, 2007.

[53] M. A. Khanesar, M. Teshnehlab, and M. A. Shoorehdeli, “A novel binary particle swarm optimization,” in *2007 Mediterranean conf. Control & Automation*, 2007, pp. 1–6.

[54] R. Eberhart and J. Kennedy, “Particle swarm optimization,” in *Proc. IEEE Int. Conf. Neural Networks (ICNN)*, vol. 4, 1995, pp. 1942–1948.

[55] J. Kennedy and R. C. Eberhart, “A discrete binary version of the particle swarm algorithm,” in *1997 IEEE Int. Conf. Systems, Man, and Cybernetics: Computational Cybernetics and Simulation*, vol. 5, pp. 4104–4108.

[56] A. Hashimoto and N. Ishiura, "Detecting arithmetic optimization opportunities for C compilers by randomly generated equivalent programs," *IPSJ Trans. System LSI Design Methodology*, vol. 9, 2016, pp. 21–29.