AUTOMATIC CODE OPTIMIZATION USING GRAPH NEURAL NETWORKS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR
THE DEGREE OF
MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

By
Melih Peker
January 2023

Automatic Code Optimization Using Graph Neural Networks
By Melih Peker
January 2023

We certify that we have read this thesis and that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Özcan Öztürk(Advisor)

Ercüment Çiçek

Süleyman Tosun

Approved for the Graduate School of Engineering and Science:

Orhan Arıkan
Director of the Graduate School

ABSTRACT

AUTOMATIC CODE OPTIMIZATION USING GRAPH NEURAL NETWORKS

Melih Peker
M.S. in Computer Engineering
Advisor: Özcan Öztürk
January 2023

Compilers provide hundreds of optimization options, and choosing a good optimization sequence is a complex and time-consuming task. It requires extensive effort and expert input to select a good set of optimization flags. Therefore, there is a lot of research focused on finding optimizations automatically. While most of this research considers using static, spatial, or dynamic features, some of the latest research directly applied deep neural networks on source code. We combined the static features, spatial features, and deep neural networks by representing source code as graphs and trained Graph Neural Network (GNN) for automatically finding suitable optimization flags.

We chose eight binary optimization flags and two benchmark suites, Polybench and cBench. We created a dataset of 12000 graphs using 256 optimization flag combinations on 47 benchmarks. We trained and tested our model using these benchmarks, and our results show that we can achieve a maximum of 48.6% speed-up compared to the case where all optimization flags are enabled.

Keywords: Code optimization, Graph Neural Networks, GCC, Compilers.

ÖZET

ÇİZGE YAPAY SİNİR AĞLARI KULLANILARAK OTOMATİK PROGRAM ENİYİLEMESİ

Melih Peker
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Danışmanı: Özcan Öztürk
Ocak 2023

Derleyiciler yüzlerce eniyileme seçeneği sunar ve iyi bir eniyileme dizisi seçmek yoğun çaba ve uzman girdisi gerektirdiği için karmaşık ve zaman alan bir iştir. Bu nedenle, literatürde eniyilemeri otomatik olarak bulmaya odaklanan birçok araştırma mevcuttur. Bu araştırmaların çoğu statik, uzamsal veya dinamik özellikleri kullanırken son zamanlarda, derin sinir ağlarını kaynak koduna doğrudan uygulamaya yönelik araştırmalara ağırlık verilmektedir. Biz de bu tez çalışmasında, kaynak kodunu çizge biçiminde temsil ederek statik özellikleri, uzamsal özellikleri ve derin sinir ağlarını birleştirdik. Otomatik olarak en uygun eniyileme seçeneklerini bulmak için ise bir Çizge Yapay Sinir Ağı (GNN) modeli eğittik.

Modeli test etmek için sekiz adet ikili eniyileme seçeneği ile Polybench ve cBench adında iki adet veri kümesi seçtik. 47 program ve 256 adet eniyileme seti kombinasyonu kullanarak 12000 çizgeden oluşan bir veri seti oluşturduk. Modelimizi bu çizgeleri kullanarak eğittik ve test ettik. Sonuçlar, tüm eniyileme seçeneklerinin etkinleştirildiği duruma kıyasla maksimum %48,6 hız artışı elde edebileceğimizi gösterdi.

Anahtar sözcükler: Program eniyilemesi, Çizge Yapay Sinir Ağları, GCC, Derleyiciler.

Acknowledgement

I want to express my sincere and warmest gratitude to my advisor Prof. Özcan Öztürk, for his continuous support during my studies. His guidance and advice carried me through all the stages of writing my thesis, and his patience and faith in me sustained me this far. As a thoughtful instructor, he introduced me to the recent breakthrough in computer science and guided me during my master's degree. I could not dream of a better advisor and mentor for my master's degree. Also, I would like to thank my committee members, Prof. Süleyman Tosun and Prof. Ercüment Çiçek, for their excellent comments and suggestions. Furthermore, this work has been supported in part by the Scientific and Technological Research Council of Turkey (TÜBİTAK) 1505 program through Teydeb 5210111 project.

Finally, my deepest gratitude to my supportive wife, Burcu, and my whole family for their continuous support and understanding when undertaking my research and writing my thesis. Their support during my studies helped me to complete my thesis during sleepless nights.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Compiler optimizations allow programmers to create more efficient object codes by applying various transformations to the source code at compile time. Those optimizations have four main goals: reducing execution time, minimizing compile time, consuming fewer resources, and minimizing the code size [2]. Such optimizations generally minimize resource usage so that the resulting programs run faster. To maximize efficiency, most compilers have lots of optimization options. For instance, the GCC compiler has more than 200 predefined optimization options [2], and LLVM has 150 optimization options [3]. Choosing the best optimizations for a program is very important for utilizing the hardware, minimizing the cost of resources, and creating scalable deployments. Since there are many optimization options and it is essential to choose the best ones for efficiency, selecting the proper optimizations for a particular program is very complex.

To make it easier, GCC offers six predefined optimization levels; $-O1$, $-O2$, $-O3$, $-Os$, $-Ofast$, and $-Og$ [2]. These levels selectively choose and utilize some of the compiler optimization options. Although it is easier to choose between these levels rather than 200 flags, it is still hard to choose the best one. Moreover, using all optimizations or choosing the most aggressive level (for GCC, $-O3$) does not guarantee the best performance [4]. For instance, when we compare the execution time of benchmarks using all the flag combinations and $-O3$

baseline, we observe that neither $-O3$ nor enabling all the optimization flags gives the best possible run time. Therefore, as shown in Figure 1.1, using predefined optimizations usually is not good enough to bring the best achievable application-specific performance [5].

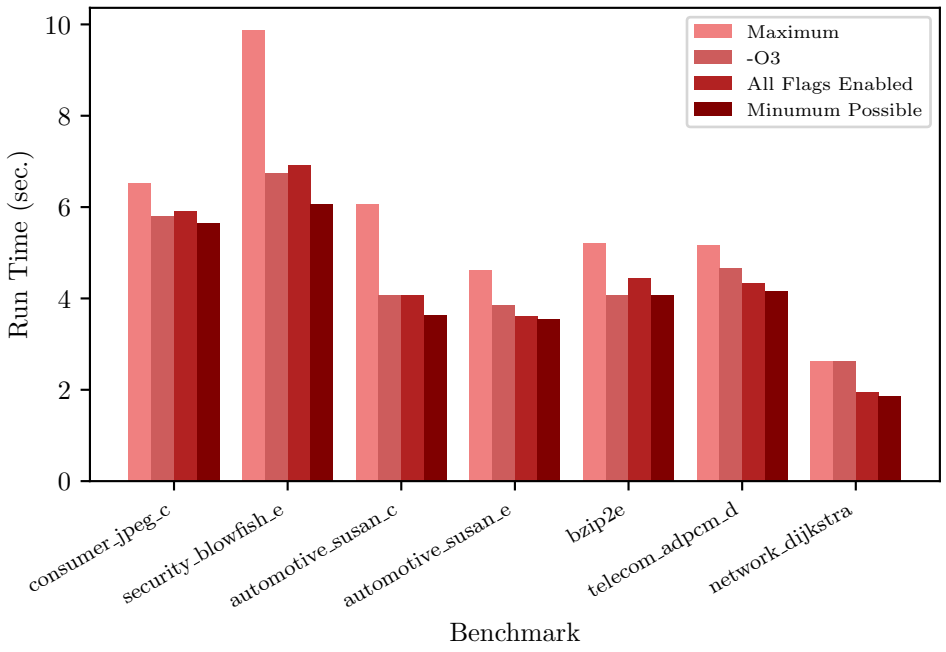

Figure 1.1: Run time of some benchmarks with $-O3$, with all optimization flags enabled along with maximum and minimum.

Since it is hard to know whether a program will benefit from an optimization flag or a combination of optimization flags, choosing the best set of flags requires expert knowledge and many attempts. Because more than 200 optimization flags exist, there are more than 2^{200} possible combinations, and it is nearly impossible to find the best one by trying all of them.

For a long time, researchers have tried to find a fast, robust, and easy method for selecting the best optimization flags. The traditional way of choosing the best optimizations includes finding hot spots in code by testing it and analyzing the assembly code [6]. This approach requires expert knowledge and time. For automatic tuning of the compiler optimizations, some researchers proposed using

genetic algorithms to generate random sets and then find the best group by mutating those sets according to a cost function [6][7]. Some researchers proposed employing *iterative search* algorithms to narrow down the search space and then to construct the best optimizations from that space [8][9][10]. Although those methods provide promising results, like achieving better run time performance than $-O3$ [10], they are not fast enough. Finding the best optimization from those large sets still requires countless executions for finding the best one.

In recent years, learning-based models have been used for modeling complex systems and understanding complex concepts like languages and images [11]. Since machine learning (ML) models are proven to solve complex problems, ML-based approaches are also used to understand source codes [5][12][13]. Although extracting features and learning patterns from source codes is not that straightforward, many research efforts employ machine learning for automatic code optimization tasks [5][12]. One of the famous ones, Milepost GCC, used machine learning to find the best optimizations by manually extracting features and learning patterns from those features [5]. Different research groups focused on spatial information of the source code and extracted spatial features from data flow graphs (DFG) and control flow graphs (CFG) for representing source codes as a fixed feature vector [12] [14]. While most of these researches used manual feature extraction, some researchers find similarities between language and source codes and use natural language processing (NLP) models for representing source code as tokenized feature vectors [15] [16]. Representing source codes as fixed-size vectors is a very sophisticated task, and the authors aimed to tackle these challenges.

In this thesis, we tried to combine all of the benefits of the previous work and create a system that benefits from both hand-crafted features, spatial information, and automatically extracted features. We specifically utilized Graph Neural Networks (GNNs) on program CFGs to extract features and predict the best compiler flags. Our specific contributions can be listed as follows:

- We extract meaningful hand-crafted features from the source code to represent the program as a fixed-size feature vector. We dive deep into the basic block level to extract the most meaningful information and create feature vectors for each basic block.

- We utilized the spatial information for preserving and using CFG features such as loops, jumps, calls, and other characteristics.

- We generate graphs by combining hand-crafted features with spatial information for representing the source code as directed graphs with node features. We create a scalar representation framework for source codes by representing source codes as graphs.

- We created a GNN model and trained the model using the graphs that we generated. Using our generated graphs, the GNN model suggests the best compiler flags for source codes.

- Models have been tested in different domains and with other benchmarks to show the applicability in different scenarios. Our approach is not discipline-specific and can be applied to various cases. For instance, the GNN model trained using a math benchmark [17] can be used on a benchmark with much more complex programs [18].

- We find optimization sequences that lead to run faster times than $-O2$ and $-O2$ + all selected flags enabled cases. The $-O3$ flag is avoided in some critical cases due to increased source code size (which leads to prolonged run time under certain circumstances - e.g., on a CPU with exceptionally small L1 instruction cache) [2][6][19][20]. Therefore, our proposed method's goal is to find better optimization sequences than enabling all selected flags on top of $-O2$ optimization.

The rest of the thesis is organized as follows. The next chapter discusses the related work, whereas Chapter 3 gives the necessary background. Chapter 4 explains the details of our GNN-based approach. Chapter 5 discusses the experimental setup and results, and Chapter 6 concludes the thesis.

Chapter 2

Related Work

In recent years, learning-based ML models proved successful in modeling complex systems and understanding complex concepts [11]. Therefore, much of the literature's current work focuses on using ML for an automatic compiler optimization problem.

Learning-based models learn patterns between input and output from training data and try to find similar patterns in test data to predict the outcome [11]. These approaches usually incur expensive one-time training but are inexpensive when applied to new programs. Therefore, machine learning models need a good source code representation to find similarities between them. Symbolic encoding should be used in the applications to see the similarities between code snippets [15] [16]. Most ML models need fixed-size feature vectors as input for finding the similarities between programs using linear and non-linear combinations between those features. Therefore, researchers mainly focus on finding a good source code representation. Research efforts in this domain can be classified into three main categories, as seen in Figure 2.1.

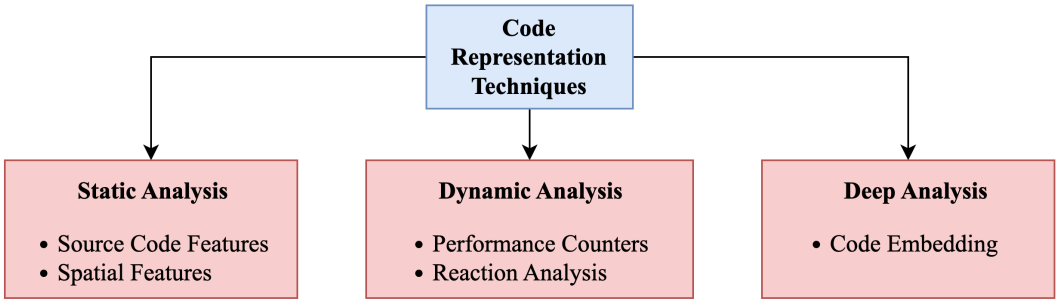

Figure 2.1: Code representation techniques used in literature.

2.1 Static Analysis

Static analysis extracts features directly from the source code or the compiler intermediate representation (IR). These features are extracted **without** running the code itself, which is why they are called static features. Some examples of these features are the number of branches, number of loops, variable types, number of arithmetic operations, and other static properties.

2.1.1 Source Code Features

The source code features are directly extracted from the source code itself. One of the earliest research on source code features is done by Agakov et al. [21]. They tried to automatically focus on the spots likely to improve performance in their work. Even though they used an iterative search algorithm, they created the nearest neighbor (NN) classifier for narrowing the search space. The predictive model then defines a suitable region of interest (ROI) to search. For the NN implementation, they represented the code with 36 different source code features.

Some examples of those features are:

- Is the loop nested?
- Number of branch instructions in a loop
- Number of divide instructions in a loop
- Number of call instructions in a loop
- Number of generic instructions in a loop
- Number of array instructions in a loop
- Number of memory copy instructions in a loop
- Number of int variables in a loop
- Does the loop have branches?
- Does the loop have regular control flow?

In their implementation, the authors used principal component analysis (PCA) to reduce the dimensionality of the dataset. It turns out that most of these raw code features can be used together to create combined features. For example, one can divide the number of load instructions by the number of total instructions to get the memory load ratio [22]. As a result of their analysis, they got a 5D feature vector and used it to train the NN model.

While seeking a solution for the phase ordering problem, Kulkarni and Cavazos worked on extracting per-method-based features in dynamic compilers such as Java [23]. They tried to find an order for the optimizations specific to each piece of code. Specifically, they extracted 26 features for each method in the program, such as *the number of bytecodes in a method*, *the number of words allocated for local variables*, *fraction of bytecodes that are used for compare operations (e.g., lcmp, dcmpl)*, *fraction of the bytecodes that are used for primitive and long computations*, etc [23].

2.1.2 IR Features

Most of the static feature extraction process uses the compiler intermediate representation (IR) to avoid using irrelevant information such as dead code. IR is an important target for optimization since it contains useful structural and execution information about the application. Namolaru et al. proposed a state-of-the-art model for predicting suitable optimization flags using IR features [24]. In this model, they used GCC's intermediate representation [25] to extract static features. They viewed the program as a labeled graph represented by Datalog notation [24]. Furthermore, they proposed to use the relational view of the program by considering several features such as *CFGs*, *Call Graphs*, *Loop Hierarchy*, *DDGs*, etc.

They extracted 56 static features from the relations that comprise the relational representation of the program. Some examples are given as follows:

- Number of basic blocks in the method
- Number of basic blocks with more than two successors and more than two predecessors
- Number of basic blocks with the number of instructions in an interval, such as [15, 500]
- Number of binary integer operations in the method
- Number of binary floating point operations in the method
- Number of instructions in the method
- Average number of instructions in basic blocks
- Number of instructions that do pointer arithmetic in the method
- Number of indirect references via pointers ("*" in C)
- Number of calls that return a pointer

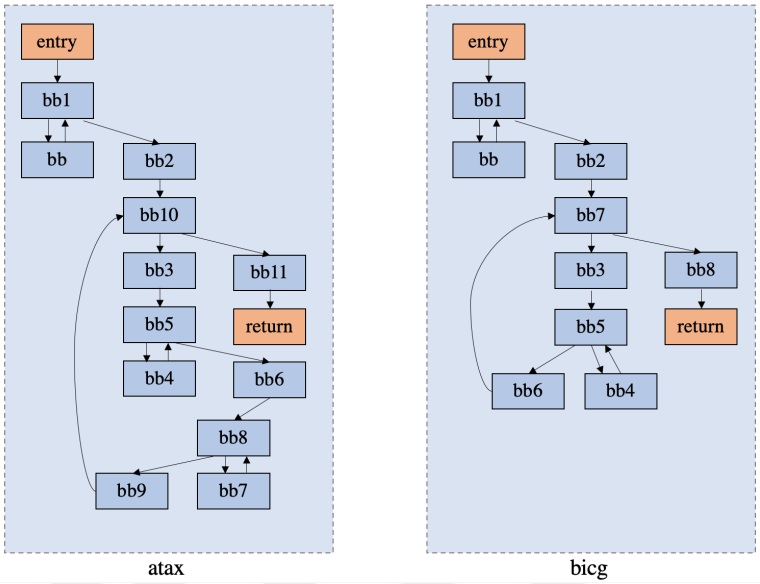

Figure 2.2: Similarities between the CFG of different algorithms.

Milepost uses the above features as input to the network and is considered an essential baseline for ML-based approaches in the literature [10][12][14][26].

2.1.3 Spatial Features

One year after the Milepost, Malik showed that spatial-based features are better for characterizing the programs instead of using only the source code features [12]. These features show how different instructions are distributed within a program, which is crucial for instruction scheduling and register allocation. More specifically, the author uses DFGs, where each node represents an instruction, and each edge shows the data dependency between two instructions. This way, the spatial information of the program can be extracted and used in training.

In 1997, Koseki et al. proposed using dependence graphs to predict optimal unrolling factors for nested loops [27]. They specifically utilized the flow-dependence, reuse, and output dependencies, to predict proper unrolling factors for loops.

Thanks to the research that demonstrated the benefits of using dependence graphs in characterizing loops, they are also used as features for representing source codes. For instance, Figure 2.2 shows an example of the similarity between the CFG graphs of similar algorithms. Similar programs may have similar CFG structures, so the CFG information can be used as an indicator when characterizing the programs.

In 2012, Park et al. proposed using CFG for feature extraction in code representation [14]. They used MinIR [28] to analyze the intermediate representation (see Section 3.3) and extracted a 10-dimensional feature vector for each node in the CFG. MinIR generates CFG as a directed graph and uses CFG to extract features like $bb0 \rightarrow bb1, bb4 \rightarrow bb3$, etc. The authors combined the list of directed edges with a 10D feature vector to train a graph-based Support Vector Machine (SVM) kernel (see Figure 2.3).

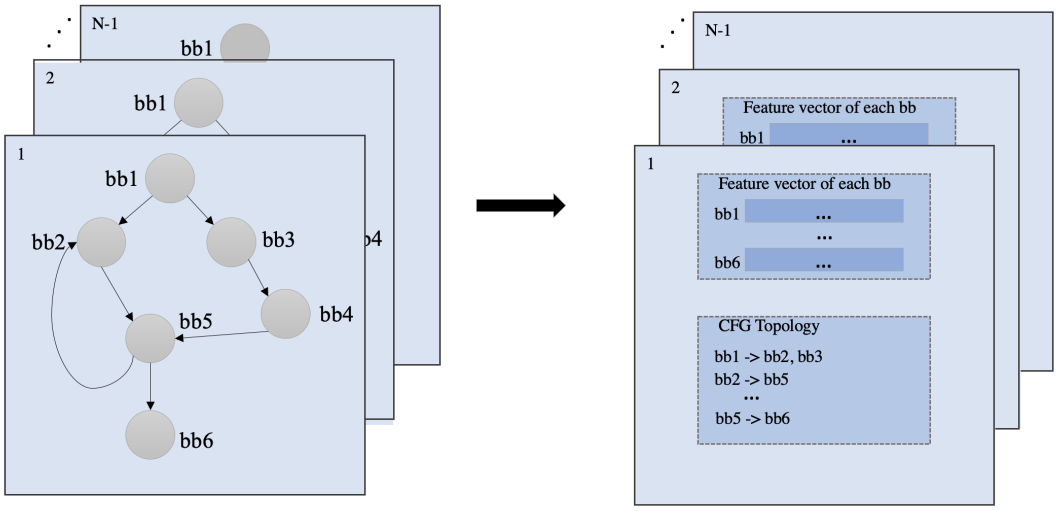

Figure 2.3: Use of graph based features.

2.2 Dynamic Analysis

Static code features may contain information about the code segments that are rarely executed, which can confuse the machine learning model [22]. Some program information, such as the loop bounds, depends on the program input, which can only be obtained during the execution time. Also, static code features may not precisely capture the application behavior in the run time environment (such as resource contention and I/O behavior) [29]. This highly depends on the computing environment, such as the number of available processors and co-running workloads [22].

To address such run time changes, dynamic analysis is usually preferred. The run time information contains behaviors on specific hardware components with particular inputs, hot code, loop counts, etc. Although it is more complex than collecting static features, the dynamic analysis may represent the source code better in specific environments and execution scenarios [29].

In 2007, Cavazos et al. used performance counters to extract useful features from programs. Static features are great for finding a correlation between multimedia kernels on embedded processors but those features performed poorly on more general applications [29]. While static features can characterize local code constructs such as loops, they provide a poor global characterization once aggregated over many such code sections. Therefore, they used **performance counters** to represent general-purpose applications [29].

Performance counters have been extensively used for performance analysis in explaining program behavior by capturing events. These events can describe several characteristics of the running program, such as cache hits and misses and branch prediction statistics [29]. One of the advantages of performance counters is that they can capture how the target program behaves on specific hardware and avoid the irrelevant information that static code features may bring [22]. Using those counters, authors collected 60 events related to performance, and those events are used as features for an ML model [29].

Although the performance counters are successful, they lack in terms of being dependent on specific hardware. Researchers and developers implemented methods and tools to overcome this limitation. Intel Pin [30] is an example that allows developers to analyze an application at the instruction level without the need for detailed knowledge of the underlying instruction set. The Intel Pin API is designed to be architecture independent whenever possible by making analysis tools source compatible across different architectures using the same instruction set, i.e., X86. For example, Microarchitecture-independent Workload Characterization of Applications (MICA) [31] uses Pin for analyzing and characterizing programs using run time statistics.

Fursin and Temam proposed another dynamic analysis method based on the programs' reactions to the specific optimization sequences [32]. First, they created a collective optimization framework in which users can run their program with different optimization combinations in different architectures and send the statistics of their run time to a repository. As the collective repository expands with the run time statistics and the optimization combinations, it turned into a structure that users search for **similar** programs and optimization flags that give the best performance so that they will know the best optimization flags that their program can benefit [32].

Finding similarities between programs is the challenging part. To find similar programs, they used reactions to some optimization combinations. The response is the performance change when a specific optimization flag is applied to a program [32]. Suppose we have two optimization flag sequences (1 and 2), and when we apply these sequences to a program, the performance changes are specified as C_1 and C_2. Let us assume that the performance change of the target program P is $C_1 > C_2$, the performance change for another program, $P\prime$, is $C_1 > C_2$, and the change for another program, $P\prime\prime$, is $C_1 < C_2$. Then, $P\prime$ gets a score of 1, and $P\prime\prime$ receives a score of 0 since the performance change using the same optimization flags is similar for P and $P\prime$. The program with the best score is considered the matching program. The more pairs (reactions to optimizations) are compared, the more accurate and reliable the program matching is [32].

Although dynamic analysis can help find handy features, it may be affected by the context switches happening in the system. For instance, a different program running after the population of performance counter registers may alter the measurements and cause using inaccurate values during the dynamic analysis. Therefore, the dynamic analysis may suffer noise in some environments [22].

2.3 Deep Analysis

Manual feature extraction methods require expert input, which is even more complex at the source code level. Specifically, static features are not always enough to represent a program, and extracting dynamic features is exhaustive since it requires run time information [15]. Moreover, finding relations between hand-crafted features is a complex procedure. Therefore, recent studies focused on automatic feature extraction methods using deep neural networks.

In 2015, Peng et al. proposed representing programs as vectors for training deep models [15]. To feed the codes to the deep network, they used language models similar to those used in NLP problems. This is meaningful since programming languages, like natural languages, have considerable repeatability and predictability. Therefore, although code analysis and natural language processing seem far from one another, they have many similarities, such as extracting semantics from code/text, summarizing functions/paragraphs, classifying code/articles, and comparing code/text similarity. However, instead of directly tokenizing the characters, as in the case of NLP, they used Abstract Syntax Trees (AST) and tokenized the nodes in the AST for training. Each node in ASTs (e.g., ID, Constant) is mapped to a real-valued vector, which can provide high-level abstract features. The tree structural nature of ASTs can also capture the programs' structural information. In this vector representation, the similarity is defined based on the following intuition: similar symbols have similar usages. ID and Constant can be an operand of a binary operator; both For and While are blocks of codes, etc. Therefore, similar symbols in some aspects should have similar values in corresponding feature dimensions. When they used an NN model for

```
#define DTYPE float
#define ALPHA(a) 3.5f * a
inline DTYPE as (DTYPE x)
{
    return ALPHA(x);
}
__kernel void saxpy( __global DTYPE *input1, __global DTYPE *input2, const int nelem)
{
    unsigned int idx = get_global_id(0); // ax + y
    if(idx < nelem){
        input2[idx] += ax(inpu1[idx]);
    }
}
```

(a)

```
inline float A(float a){
    return 3.5f * a;
}
__kernel void B(__global float* b, __global float* c, const int d){
    unsigned int e = get_global_id(0);
    if (e < d){
        c[e] += A(b[e]);
    }
}
```

(b)

Figure 2.4: Source rewriting example: (a) Original source code, (b) Source code after the source rewriting process is applied.

classification using those vectors, they achieved more than 95% accuracy, which shows vector representation is suitable for characterizing the codes.

In 2017, Cummins et al. proposed using deep neural network models for feature extraction to find GPU thread coarsening factors [33]. Neural networks extract a fixed-size vector that characterizes the entire sequence. This is comparable to the hand-engineered feature extractors used in prior works. Still, it is a learned process that occurs entirely and automatically within the network's hidden layers.

They needed to create a fixed-size input vector to feed neural networks with source codes. Firstly, they removed semantically irrelevant information (such as comments) from the source code of the target program and passed it to a language model (see Figure 2.4). Using word embedding, the language model converted the arbitrary length stream of code into a fixed length vector of real values. Finally, the fixed length vector (tokenized source code) is then used for training an LSTM-based language model [33]. The similarities between source codes and

```
define void @mark_visted(%struct.node*)
%2 = alloca %struct.node*, align 8
store %struct.node* %0, %struct.node** %2, align 8
%3 = load %struct.node*, %struct.node** %2, align 8
%5 = load i32, i32* %4, align 8
%6 = icmp ne i32 %5, 0
br i1 %6, label %7, label %8
```

(a)

```
define void <@ID>({ i32, opaque**, i32 }*)
<%ID> = alloca { i32, opaque**, i32 }*, align 8
store { i32, opaque**, i32 }* <%ID>, { i32, opaque**, i32 }** <%ID>, align 8
<%ID> = load { i32, opaque**, i32 }*, { i32, opaque**, i32 }** <%ID>, align 8
<%ID> = load i32, i32* <%ID>, align 8 <%ID> = icmp ne i32 <%ID>, <%INT>
br i1 <%ID>, label <%ID>, label <%ID>
```

(b)

Figure 2.5: Preprocessing step example: (a) Original LLVM IR, (b) Preprocessed LLVM IR.

natural languages showed that natural language processing models could also be used for source code analysis.

Similar work is proposed by Zhang et al., where they compare and analyze the similarity between source codes and binary codes for detecting the reused open source codes [34]. In their work, they used a famous NLP model, word2vec [35], for characterizing the source codes, and using this characterization, they cluster similar applications. They used *clang* compiler and LLVM IR for representing binaries from various architectures in the same format. This process unifies the source code and binaries into the same intermediate form while preserving semantic information. With the help of some preprocessing, they simplified the LLVM IR representation. They also used CFGs as well. Although directed graphs are not sentences, they treated directed basic blocks as adjacent sentences. In the preprocessing stage, user-defined variables are renamed as specific tags plus numbers; for example, local variables are renamed as %ID, and global variables are renamed as @ID. This naming method ensures a fixed vocabulary, and the control flow is specified by labels (see Figure 2.5). These tags are helpful for deep learning methods to understand the source codes better.

In 2018, Mendis et al. used a data-driven approach to predict the clock cycles a processor takes to execute a block of assembly instructions in a steady state

(the throughput). More specifically, they used the assembly code to implement the data-driven technique. First, they applied *canonicalization* to convert the assembly input into a more structured form and map the compiled assembly block to a list of instructions. Each instruction consists of a list of tokens representing its operation code, source operands, and destination operands, separated by distinguished delimiter tokens. After that, they use *embedding* to convert instructions into a real-valued vector in a high-dimensional space. Similar to previous methods, it maps tokens into embedding vectors for the LSTM network. Using those embeddings, they showed that they could predict throughput better than **llvm-mca** and **IACA** [16].

2.4 GNN-based Code Optimization

In an effort to find suitable optimization flags automatically, we have discussed different methods. While some methods use only simple predefined static code features, others treat source code as natural language and tokenize source codes as they classify texts. Although there are various techniques in the literature, a common property shared by these approaches is the use of static source code features.

As we discussed in Section 2.1.2, most of the successful efforts in literature use the Milepost project as a baseline and try to outperform it using better features or combining additional information [10][12][14][26].

Although the recent work proposes using deep learning directly on the source code without dealing with intermediate representation, compiler passes, etc., the features generated by those deep learning methods are not interpretable. This problem makes those approaches a black box that cannot be adapted to specific domains and cannot be improved using different static features [22].

Dynamic analysis can be a great option if the goal is to optimize a code for specific hardware. However, dynamic analysis is not valid if the code optimization targets different architectures **without** running the code.

One of the most successful and interesting efforts among the prior efforts is Park et al.'s work. As discussed in Section 2.1.3, Park et al. proposed using CFG for feature extraction. They used not only the static code analysis but also the spatial information of the program for finding suitable optimization flags. They used graph-based SVM kernels for learning patterns from the connections between the basic blocks. Although using graph kernels is an excellent way to learn spatial information from directed graphs, they are limited in expressiveness. They highly depend on hand-crafted feature maps, which require expert input and may change as train data accumulates [36].

Considering the wide set of successful efforts in the literature, we propose using static source code features and deep analysis to learn more complex patterns than conventional methods.

In addition to the static features, we also use the spatial features generated by the CFG. Although our method looks like it is in the static spatial features category, we automatically use GNNs for learning patterns on source code. To the best of our knowledge, this is the first approach directly using the GNNs for automatic code optimization. Therefore, we treat our method as a pioneer bridge between static analysis and deep analysis since we both use the static source code features and learn complex spatial patterns using GNNs.

2.5 Program Features for Automatic Code Optimization

In this section, we compared and classified the commonly used features in the literature. Commonly used parts for automatic code optimization can be classified into four main categories as shown in Table 2.1. As discussed in Section 2.4, static source code features are one of the most commonly used features in the literature. Information about loops, basic blocks, arithmetic operations, etc., is beneficial for representing and classifying the source code.

Tree-graph-based features give insight into the execution order of the program. These features show the loop structures, branching nodes, etc., which offers a great idea about the program structure and data flows. This category is divided into two subsections since while some work uses CFGs for tree-graph-based features, some use DFGs for extracting spatial information.

Dynamic features can be used if the task optimizes code for specific hardware. Reaction analysis, loop counts, and cache hit/miss information are valuable for particular hardware but not helpful for different architectures.

Finally, deep analysis methods use abstract syntax trees or word embedding to analyze the source code. Those methods do not use hand-crafted features. Instead, they extract their own features.

In Table 2.2, we classified the efforts and their respective contributions to the field. These efforts can be classified into three main categories.

The first category is conventional methods, which include iterative compilation, adaptive compilation, and genetic algorithms for finding a good subset of possible optimization flags. Those methods create the subsets either by starting with an empty set and extending this set or starting with all flags and narrowing the set iteratively.

Table 2.1: Common features used in machine learning techniques for automatic code optimization.

Commonly Used Features	Source Code Features	Loop Information	[5] [13] [21] [24] [26] [27] [29]
		Basic Block Information	[5] [8] [13] [14] [21] [24] [26]
		# of Basic Blocks	[5] [8] [13] [14] [21] [24] [26]
		# of Floating Point Operations	[5] [8] [13] [21] [23] [24] [26]
		# of Memory Operations	[5] [21] [24] [26] [37]
		# of Integer Instructions	[5] [8] [13] [21] [23] [24] [26]
		# of Array Instructions	[5] [13] [21] [23] [24] [26]
		# of Calls in Method	[5] [14] [21] [24] [26] [37]
	Tree-Graph Based Features	CFG Topology	[13] [14] [23] [24] [27]
		DFG Topology	[12] [24] [27]
	Dynamic Features	Loop Counts	[29] [38]
		Hot Code	[12] [29] [38]
		Cache Hit/Miss Information	[29] [37] [38]
		Reactions to Flags	[32]
	Deep Analysis	Abstract Syntax Tree	[33] [39]
		Word Embeddings	[15]

The second category uses hand-crafted features, including static and dynamic features, to find suitable optimization flags. There are also efforts to use static and dynamic features in a hybrid manner. While some methods use supervised learning, others use unsupervised learning methods to not deal with data labeling.

The approaches in the last category use automatic features such as deep analysis for automatic code optimization. Furthermore, they do not use any predefined or hand-crafted features.

Table 2.2: Efforts in the literature and their respective contributions in the field.

	Iterative Compilation	Adaptive Compilation	Genetic Algorithms	Supervised Learning	Unsupervised Learning	Static Features	Dynamic Features	Hybrid Features	Automatic Features	Narrows Search Space	Extends Search Space	Function Level	Phase Ordering
[5]	✓			✓		✓			✓	✓			
[6]		✓	✓							✓			
[7]	✓	✓	✓								✓		
[8]	✓		✓	✓		✓					✓		
[9]	✓	✓		✓						✓	✓		
[10]	✓									✓	✓		
[12]					✓	✓							
[13]				✓		✓						✓	
[14]				✓		✓	✓	✓		✓	✓	✓	
[15]				✓	✓				✓				
[16]				✓					✓				
[19]		✓			✓					✓			
[20]	✓									✓			
[21]					✓	✓							
[22]	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓		
[23]						✓				✓			
[24]				✓		✓							
[26]				✓		✓				✓	✓	✓	
[27]					✓								
[29]	✓	✓		✓			✓						
[32]				✓	✓		✓						
[33]				✓	✓				✓				
[37]	✓			✓			✓						
[38]							✓						
[39]									✓				
[40]	✓												
[41]		✓			✓					✓		✓	

Chapter 3

Background

3.1 Compiler Optimizations

Eliminating unnecessary instructions in object code, or replacing a sequence of instructions with a faster sequence of instructions that does the same thing, is usually called "code improvement" or "code optimization." [42]. Code optimizations can create more efficient machine code by applying various transformations to the source code at compile-time [43]. The code optimizations have four main goals [44]:

- Reducing execution time
- Consuming fewer resources
- Minimizing compile time
- Minimizing code size

Compilers such as GCC and LLVM have hundreds of high-level optimizations to provide many options to the users on optimization settings [2][3]. In the following sections, we detail some of those optimizations related to our approach.

3.1.1 Function Inlining

This optimization is designed to reduce the function call overhead. When a function is called, instead of jumping and switching lines, the call function is substituted by the whole code [2]. The specific GCC flag used for this purpose is `-finline-functions`, whereas it is `-inline` for LLVM [45].

3.1.2 Loop Unrolling

Besides the actual loop command, loop control instructions also take time on loops. Therefore, this optimization unrolls loops for eliminating loop control instructions like branches and jumps, which are critical for pipeline performance [2][42]. An example of this optimization is given in Listing 3.1 and Listing 3.2. This kind of optimization is enabled by `-funroll-loops`, and `-loop-unroll` in GCC and LLVM, respectively [2][45].

```
1 for(int i = 0; i < 4 i++)
2 {
3     result[i] = a[i] + b[i];
4 }
```

Listing 3.1: Original code.

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

Listing 3.2: Fully unrolled code.

3.1.3 Auto Vectorization

To speed up the repeated instructions, auto-vectorization optimization can be applied. This optimization is designed for detecting single instruction multiple data

(SIMD) statements and completing these instructions in chunks instead of handling them one-by-one [2][45]. For instance, the loop in Listing 3.1 is vectorizable since it contains repeated summation operations. The auto-vectorization option can be enabled by `-ftree-vectorize` in GCC and `-bb-vectorize` in LLVM.

3.1.4 Function Alignment

The CPU caches the function in run time for running it without reading it from memory again. However, to cache the function properly, the function should not cross the alignment boundary [2]. Function alignment optimization can be used to ensure that the function is in the alignment boundary. This way, the start of functions is aligned to the next power-of-two greater than or equal to a machine-dependent value, n (maximum 65536), by skipping some bytes. For instance, if n is 32, this option aligns functions to the next 32-byte boundary [2]. The function alignment option can be used with `-falign-functions` flag in GCC.

3.2 GCC Optimization Levels

Many optimization options make it hard to choose the best possible optimization sequences among 200 flags. Therefore, GCC provides the users with six predefined optimization levels to make selecting optimization levels easier than dealing with individual optimization flags. Those optimization levels are as follows:

- **-O1**: Includes basic optimizations for reducing code size and execution time, preserving the compilation time. This level includes 47 optimizations such as `-fmerge-constants, -freorder-blocks, -ftree-ccp`, etc. [2].
- **-O2**: Enables all the flags in $-O1$ and enables nearly all supported optimizations that do not involve a space-speed tradeoff such as `-falign-functions, -falign-jumps`, etc. Compared to $-O1$, $-O2$

increases both compilation time and the performance of the generated code [2][4]. Some flags included as part of this optimization level are `-fipa-bit-cp`, `-fipa-cp`, `-fipa-icf`, `-findirect-inlining`, `-fpartial-inlining`, `-fthread-jumps`, etc. [2].

- **-O3**: Enables all optimizations specified by $-O2$ and also turns on 12 more flags that are heavy in both compile time and memory usage (such as `-funswitch-loops, -fpredictive-commoning`). It is not guaranteed that the code generated is better in terms of performance compared to $-O2$ [2][4]. However, in some cases, it provides a rapid increase in performance when compared to $-O2$. Some flags included in this level are `-ftree-loop-distribution, ftree-partial-pre, -funswitch-loops,` etc. [2].

- **-Os**: Enables all $-O2$ optimizations that do not typically increase the code size. Furthermore, it also enables optimizations designed to reduce the code size. This level optimizes code for the size and is useful for environments with limited storage space, such as CPUs with small cache sizes in embedded systems [2][4]. This level includes all flags in $-O2$ except `-falign-functions, -falign-jumps, -falign-labels, -falign-loops, -fprefetch-loop-arrays` flags since these flags increases the code size [2].

- **-Ofast**: Disregards the strict requirements of standard float operation semantics. Enables all $-O3$ flags and optimizations that are not valid for all standard-compliant programs (`-ffast-math, -fno-protect-parens` and `-fstack-arrays`) [2]. In this level, the order of some of the floating point operations may be changed (since it enables `-ffast-math`), which may cause problems in scientific operations.

- **-Og**: Enables flags that do not interfere with debugging for optimizing the debugging experience. Optimizes code to reasonable levels while maintaining debugging access [2]. Using this flag is better than using no optimizations while debugging since it provides faster compile time and no inference with debugging experience [4].

3.3 CFG

A control flow graph, *CFG*, is a directed graph with basic blocks as nodes and possible execution orders of those basic blocks as directed edges [46]. CFG is like a graphical representation of a program; edges in the graph represent how the execution of the program can potentially proceed, and nodes of the graph represent statements.

3.3.1 Basic Block

Each unit in the CFG is called a *basic block*. In a single basic block, a set of operations always execute together [46]. Basic blocks have only one entry and one exit point [47]. The basic block representation of the simple C code given in Listing 3.3 is shown in Figure 3.1.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      int list[10];
7      for(int i = 0; i < 10; i++)
8      {
9          list[i] = i + i;
10     }
11     return 0;
12 }
```

Listing 3.3: Simple C code with a loop.

In this program, for loop is represented in the CFG as a single basic block since it has one entry and one exit point. As can be seen in Figure 3.1, the single basic block is represented in the GCC-generated assembly code.

```
0x100003f90:  mov  eax, dword ptr [rbp - 4]
0x100003f93:  lea  edx, [rax + rax]
0x100003f96:  mov  eax, dword ptr [rbp - 4]
0x100003f99:  cdqe
0x100003f9b:  mov  dword ptr [rbp + rax*4 - 0x30], edx
0x100003f9f:  add  dword ptr [rbp - 4], 1
0x100003fa3:  cmp  dword ptr [rbp - 4], 9
0x100003fa7:  jle  0x100003f90
```

Figure 3.1: Basic block of the loop in GCC-generated assembly code.

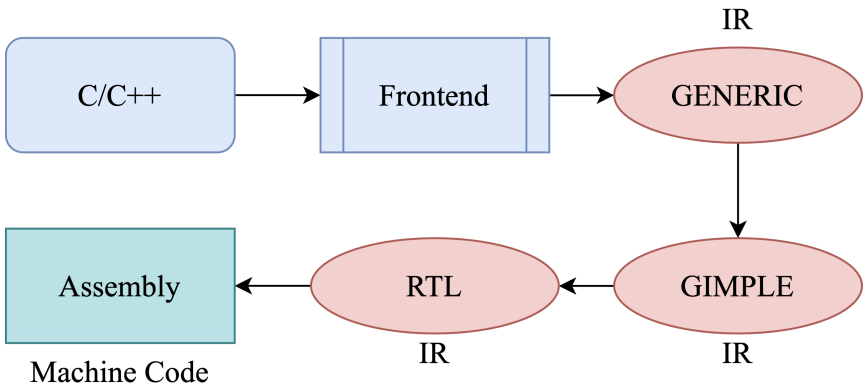

Figure 3.2: GCC compilation flow: 1) Source Code, 2) Intermediate Representation, and 3) Assembly Code.

CFG of a program can be generated from the compiler's intermediate representation (IR). GCC pipeline has several IRs ranging from high level (GENERIC, GIMPLE) to low level, close to the assembly (RTL). As seen in Figure 3.2, when a source code is compiled using GCC, there are several steps before generating the assembly code. In these intermediate steps, GCC generates internal representations, including basic blocks, possible execution patterns, etc., for applying transformations or passes [43]. Therefore, the CFG can be built using RTL and GIMPLE IR of GCC [48].

An example CFG with its basic blocks and execution order is given as a directed graph shown in Figure 3.3. In this representation, edges show the execution order, and the edge started from `0x100003f90` block to the `0x100003f90` block shows the for loop given in Listing 3.3.

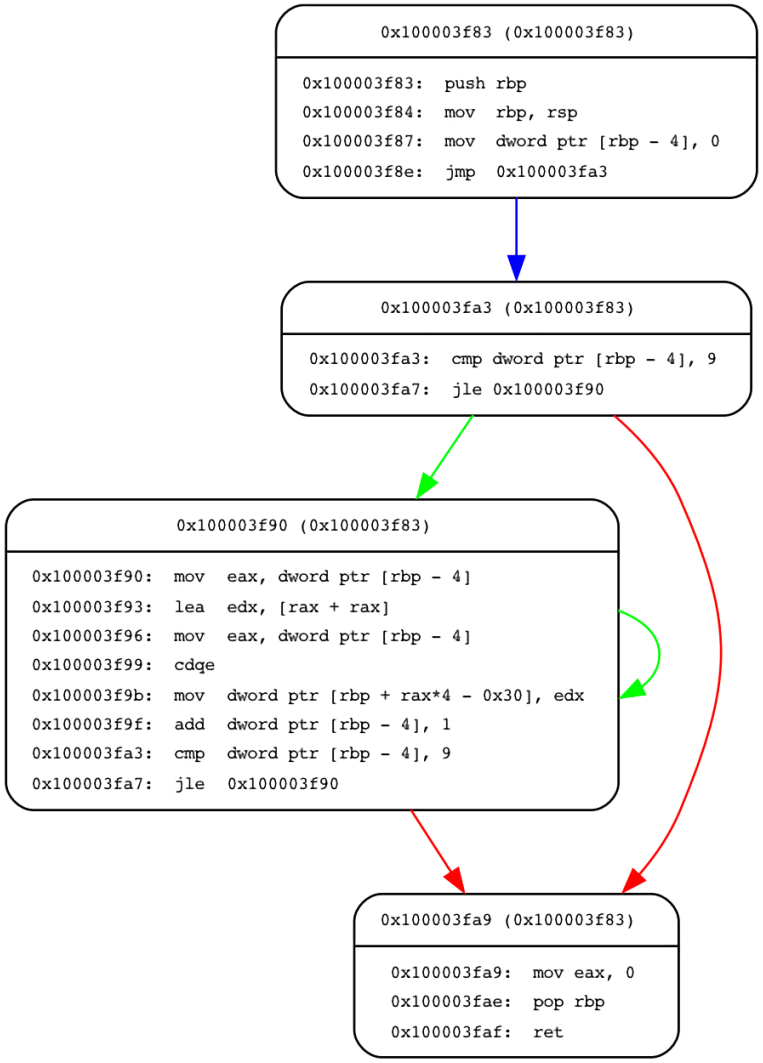

Figure 3.3: CFG of the simple C program given in Listing 3.3. The CFG is created using ANGR [1].

3.4 Neural Networks

Neural Networks (NN) are collections of neurons organized in a sequence of multiple layers [49]. Neurons receive input as activation of other neurons from the previous layer and perform a simple computation such as a weighted sum of the input followed by a nonlinear activation function [49]. Thanks to their structure, neural networks are proven to be a robust method for solving complex problems like image classification and natural language processing [11].

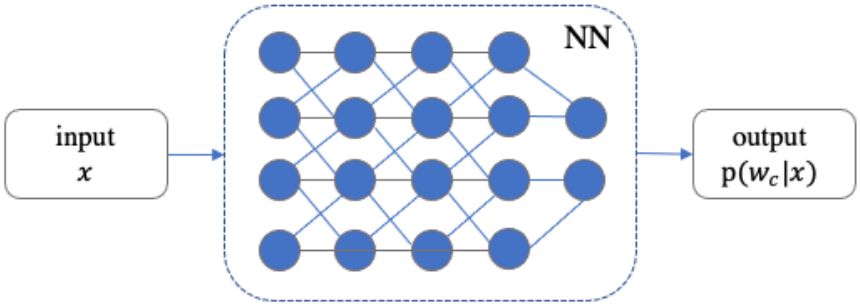

Figure 3.4: Example of a Neural Network (NN).

Figure 3.4 shows an example of a fundamental neural network. The NN has many interconnected neurons which learn the patterns from the input data. Learning is achieved by updating the weights of each neuron (weights are used for calculating the weighted sum on each neuron) during propagation until NN finds the best combination of values that best maps the inputs to the output [50].

The NN indicates the probability of association between input x and specific concept w_c [49]. Therefore, using NN, we can classify images, process natural languages, and classify complex structures like source code.

3.5 Graph Neural Networks

Graph Neural Networks (GNNs) are the types of neural networks used in graph applications. Unlike graph kernels, their multi-layered architecture and non-linear activation functions make them better for finding complex relations between nodes and edges [51]. There are three widely used GNN application types: node classification, link prediction, and graph classification.

3.5.1 Node Classification

This type of GNN has labeled nodes, each with features. The goal is to find the unlabeled nodes by looking at the features [51]. An example of this type of GNN is finding protein-protein interactions (PPI). In PPI, each graph represents a human tissue, and each node represents a protein function. Features of each node are derived from the positional gene set, and labels of the nodes are the gene ontology set. GNN model predicts the gene ontology set by considering the node features and edges [52].

3.5.2 Link Prediction

Link prediction aims to find edges between two arbitrary nodes. This type of GNN is widely used in social network applications [53]. An example of this type of prediction is the citation network. In this task, nodes represent the scientific publications, and edges between the nodes indicate the citations. Using link prediction, GNN can predict whether there is a citation between two publications [54].

3.5.3 Graph Classification

GNNs can be used for classifying a whole graph as well [51]. Using node features, edge weights, and graph structures, **graph convolutions** can be used for learning patterns and classifying graphs according to those patterns.

Graph convolution is similar to convolution in convolutional neural networks (CNN). In CNNs, neighboring pixels of images are convolved with fixed-size kernels (3×3, 5×5) to find the image's local features. The idea is similar in graphs as well. Neighboring nodes are convolved with fixed-size kernels so that the relationship between those nodes can be represented easily [51]. In Figure 3.5, the similarity between image convolution and graph convolution is given. As can be seen, while image convolution finds the correlation between pixels, graph convolution finds the correlation between graph nodes.

Moreover, convolutional kernels can recognize identical features independently of their spatial locations since they are shift and translation-invariant [55]. Convolution in graphs is mathematically defined as follows:

$$h_i^{(l+1)} = \sigma\Big(\sum_{j \epsilon N(i)} \frac{1}{c_{ij}} h_j^{(l)} W^{(l)}\Big) \tag{3.1}$$

where $h^{(l)}$ is the vector of activations of node i in the l^{th} neural network layer. $N(i)$ is the set of neighbors of node i, c_{ij} is the product of the square root of node degrees, $W^{(l)}$ is a layer-specific weight matrix, and σ is an activation function [56].

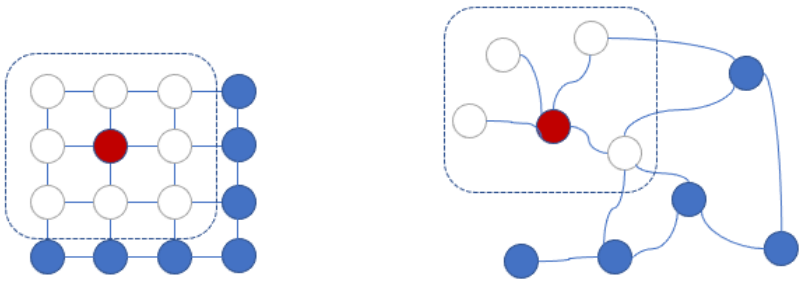

Figure 3.5: Comparison between convolution in images (left) and convolution in graphs (right).

To reduce the dimensions of the graph in the network, graph coarsening layers are used in GNN [55]. The graph coarsening layers reduce the number of parameters to learn in GNN, and also these layers allow the network to represent the whole graph as a 1D feature vector at the end. This feature vector is then fed into a dense layer. Finally, a classification layer is used to classify the graph [55].

Chapter 4

Our Approach

Our main objective is to create a general machine-learning-based automatic code optimization framework that finds suitable optimization flags for a given source code in a reasonable amount of time without executing it. Even though we use the GCC compiler in our implementation, our approach can be applied to any compiler framework with slight modifications.

4.1 Overview

In recent years, research on automatic compiler optimization has leveraged machine learning algorithms [10][12][14][26]. Since compiling and running code several times to find suitable optimization flags is not feasible, most of the research towards proposing appropriate optimization flags is focused on finding these flags without actually running the code itself. In this context, static source code analysis should be applied to propose suitable optimization flags for an application. There are several static code analysis methods proposed by lots of researchers in the past [5][8][12][13][14][23][26]. These methods relied on extracting hand-crafted features from source code, and those features are fed into machine learning models

for learning patterns from source code. Those patterns are then used to find similarities between different applications so that the automatic code optimization framework proposes similar optimization flags to similar programs.

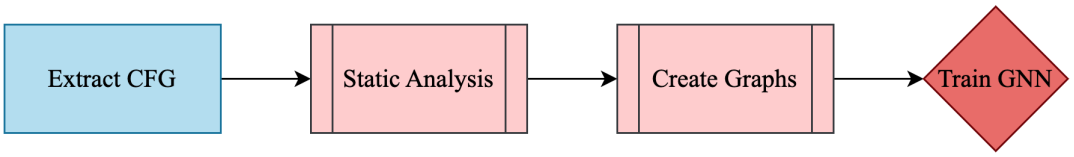

Figure 4.1: High level view of our approach.

As opposed to the efforts in the literature, we represent source codes using not only the static features but also the programs' spatial information. After we generate the features, we use neural networks to learn complex patterns of the source codes and propose the best possible optimization flags using the neural network model. Our implementation is composed of four main steps, as shown in Figure 4.1. Individual steps of the approach can be summarized below:

- Extract the CFG from the intermediate representation.
- Create node features using the static analysis at the basic block level.
- Generate directed graphs using the node features and CFG edges.
- Train a graph neural network for learning patterns from spatial information of the CFG.

We will explain these steps in the following sections in detail.

4.2 Dataset Generation

Unlike conventional machine learning models, deep neural networks automatically extract and find features from input data; they do not require hand-crafted features for training [57]. Thus, to learn what is essential and what is not, they need a vast amount of data for learning patterns [57].

Since GNN is also a deep neural network (it has a multi-layered architecture consisting of several hidden layers), we need to create a large dataset for training our model. Our dataset generation is composed of multiple tasks:

- Creating source code pool,
- Choosing set of optimization flags,
- Saving run times for each combination of flags,
- Analyzing source codes and creating graphs.

4.2.1 Creating Source Code Pool

We use C source codes to train our model. We use computationally intensive benchmark datasets to see the effect of optimization flags. Also, we use a comprehensive set of benchmarks from different domains to understand the capabilities of our model.

One of the most used benchmarks in compiler research is the Polybench benchmark suite due to its clean structure and computationally heavy applications [17]. The benchmark consists of 30 different applications with different features.

In addition to Polybench, we also use cBench [5] as part of our source code pool. In 2008, Fursin started as an initiative for creating an extension to MiBench [18] and created Collective Benchmark set, *cBench* and used cBench for training Milepost GCC in 2011 [5]. The dataset has more than 30 programs, which all

have very long and complex algorithms. Thus we included cBench in our work as well.

Unlike many other works, we used multiple datasets to evaluate our models in different domains.

4.2.2 Selecting Optimization Flags

There are more than 200 optimization flags in GCC. Furthermore, most of them are not binary, requiring arguments. For example, `-flto-compression-level` flag requires an integer value between 0-19 for defining the compression level of the intermediate language objects [2]. Therefore, training with all the optimization flags would not be feasible. As a result, we select a suitable subset of all the available flags. To do this, we compile our benchmarks with a single optimization flag at a time and record the execution time. We choose the most effective binary flags and use those flags as an initial set. According to the collected results in our benchmark set, we observe that the following flags provide the most significant performance improvements:

- `-fivopts`
- `-funsafe-math-optimizations`
- `-finline-functions`
- `-fguess-branch-probability`
- `-ftree-loop-optimize`
- `-funroll-all-loops`
- `-ftree-vectorize`
- `-funroll-loops`

On average, compiling, running, and analyzing a program in the Polybench set takes roughly 1 minute, and for the cBench, that exceeds 5 minutes (because of the size of applications in the cBench). Even with eight flags, there are $2^8 = 256$ possible binary flag combinations, and generating a dataset for Polybench takes five days, and for the cBench, it takes 26 days. Therefore, we work with a good set of optimization flags.

The order of the flags in the optimization set, i.e., phase order, also affects the performance of the optimization as well [23]. However, most of the studies in the literature focused on selecting optimization flags problems and phase ordering problems separately. Therefore, we focused on selecting optimization flags; phase order is also out of the scope of our work.

4.2.3 Benchmark Run Times

After we choose the flags, we compile and run the programs using the combinations of these optimization flags. We recorded the run time of each possible combination. After creating the dataset, we know which combination takes how long to run.

For example, Table 4.1 shows the effects of different optimization flags on the execution time of the *telecom_gsm* benchmark. As seen from this table, execution time ranges from 3.39 to 5.16, a 34% difference.

To observe whether using all the flags gives the best results, we showed all the possible run times using two baselines, namely the original $-O2$ and **enhanced** $-O2$, meaning that all of the test flags are enabled with the $-O2$. We used violin graphs to visualize all run times and baselines in a single chart. These graphs show the maximum, minimum, and mean run times using three horizontal lines. Also, the shaded area behind these lines offers the distribution of the run times generated by all 256 combinations.

Finally, we added two baselines for observing the effect of using no additional flags and all eight flags. The violin graphs in Figure 4.2 and 4.3 visualize how combinations of our binary flags changes at run time.

As can be seen from Figures 4.2 and 4.3, using all flags does not always yield the best possible result. As discussed in previous sections, some flags affect one another, resulting in performance degradation, such as the *covariance* benchmark in Figure 4.2. Therefore, our model tries to find the best possible combination instead of using all flags.

Also, we get a compilation error in some programs when we enable all the flags (*consumer_tiffmedian* in cBench suite). We could not give the enhanced $-O2$ baseline in such cases.

Table 4.1: Execution time of *telecom_gsm* application when compiled using different flag combinations.

-fivopts	-funsafe-math-optimizations	-finline-functions	-fguess-branch-probability	-ftree-loop-optimize	-funroll-all-loops	-ftree-vectorize	-funroll-loops	Time (sec.)
								5.03
✓								4.69
	✓							4.73
		✓						4.61
			✓					4.59
				✓				4.58
					✓			4.28
						✓		4.33
							✓	4.12
✓	✓							4.49
✓		✓						4.39
✓			✓					4.04
✓				✓				4.22
			✓		✓			**3.39**
				✓	✓			**5.16**
						✓	✓	4.11
		✓		✓		✓	✓	4.28
✓	✓	✓	✓	✓	✓	✓	✓	4.41

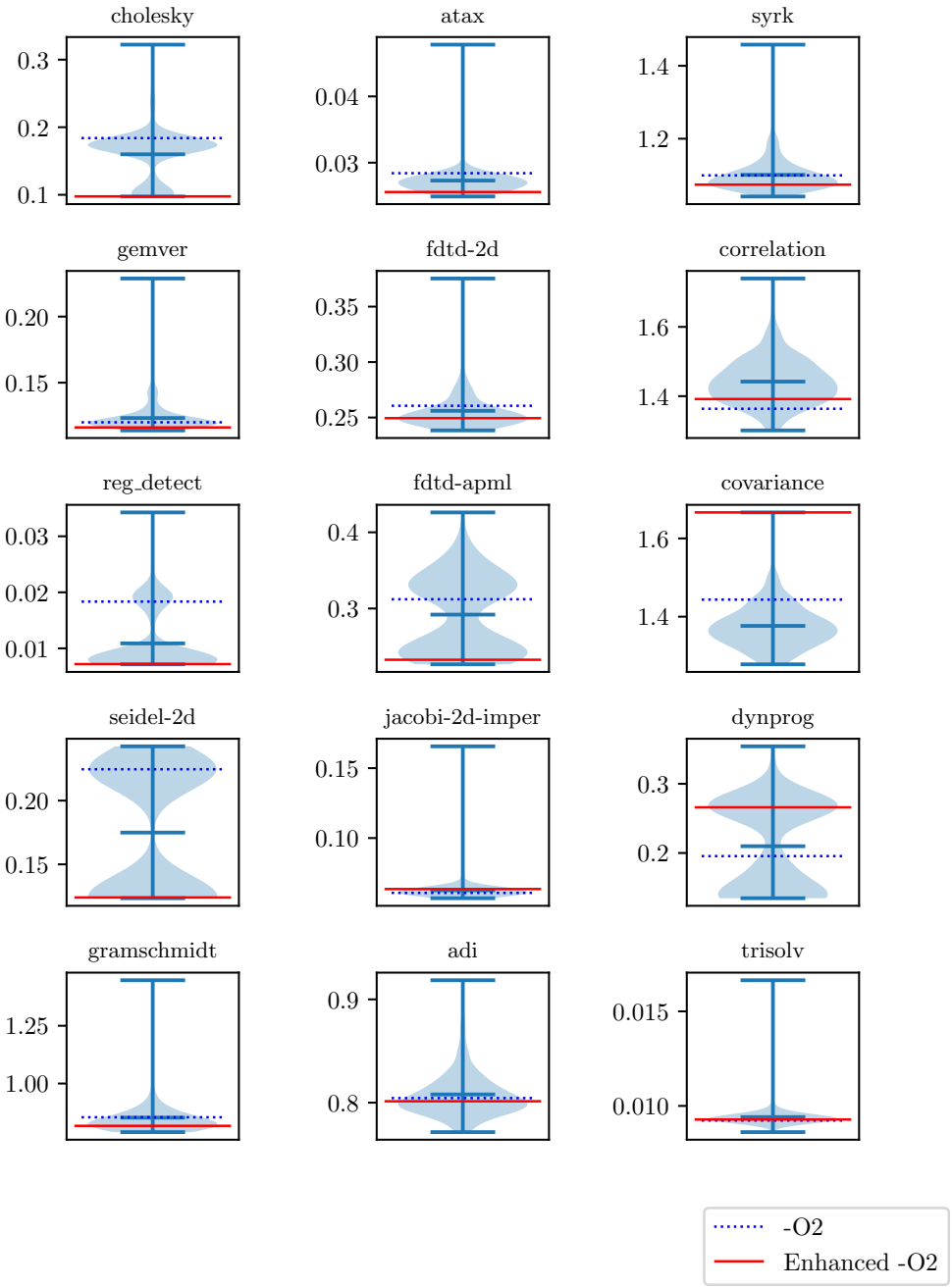

Figure 4.2: Run times of our optimizations with $-O2$ and enhanced $-O2$ baseline for some of the programs in the Polybench benchmark suite.

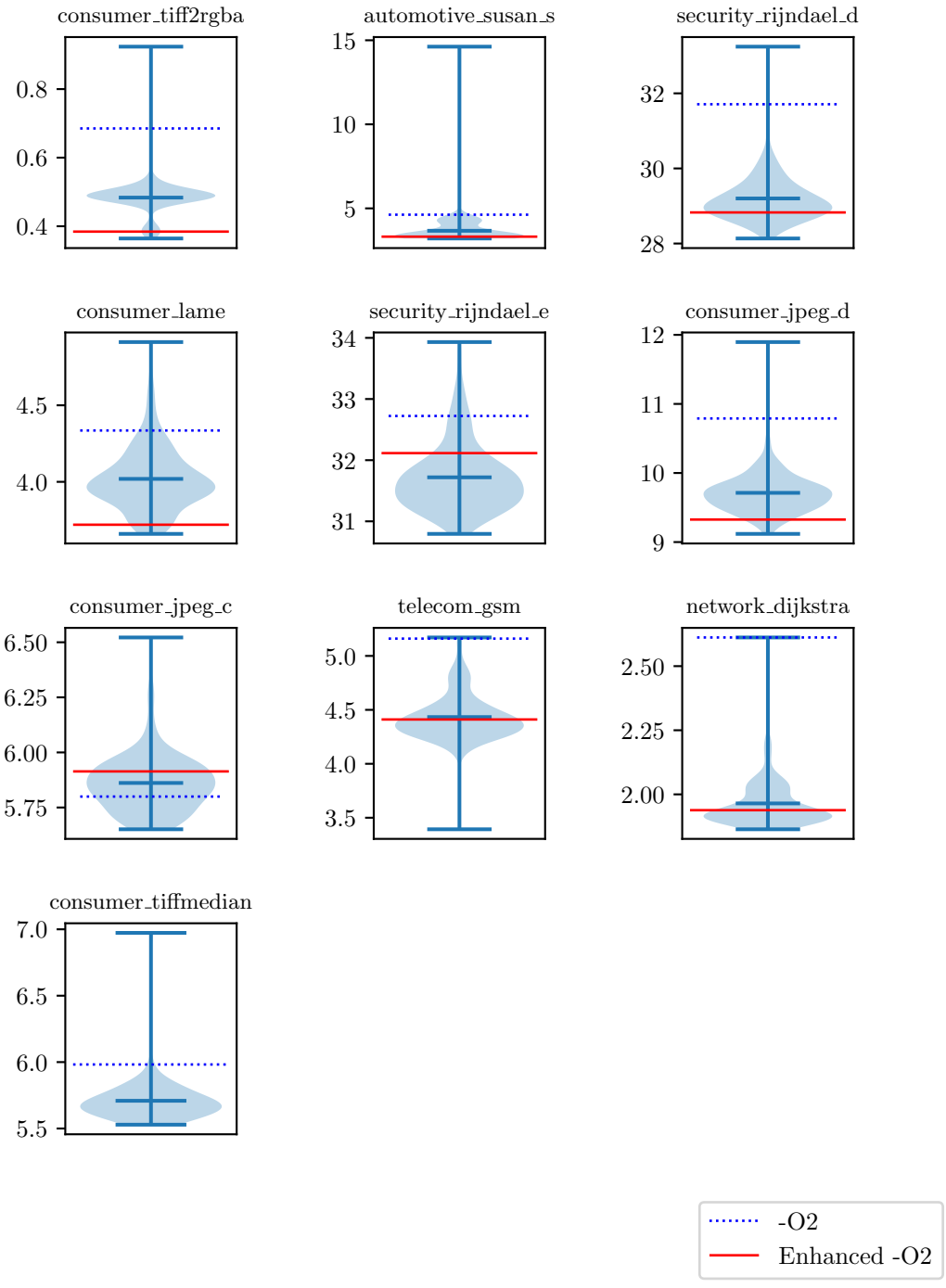

Figure 4.3: Run times of our optimizations with $-O2$ and enhanced $-O2$ baseline for some of the programs in the cBench benchmark suite.

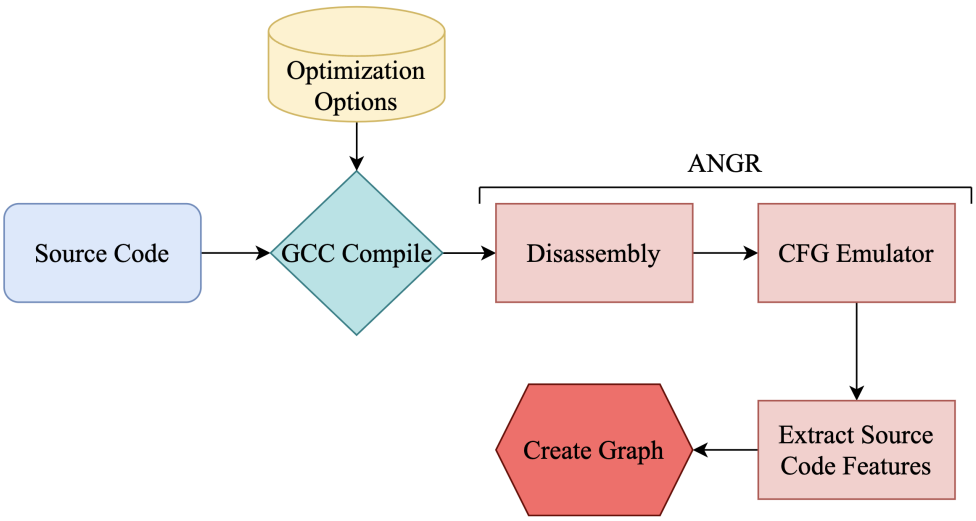

Figure 4.4: Our source code analysis framework to create the graph to be used in our GNN model.

4.2.4 Analyzing Source Codes

Our approach analyzes both source code features and CFG for spatial information. As mentioned before, to extract data from CFG, we need to examine the IR or assembly code directly. Analyzing IR requires some external frameworks with limited capabilities, such as MinIR [28]. To integrate into different compilation pipelines, we did not modify GCC by writing compiler passes. Instead, we analyze the assembly code by directly working on binary. Therefore, one of the essential features of our framework is that it does not require modifications in the existing compiler or installing third-party analyzer tools.

For analyzing the assembly output of the GCC compiler, we used the ANGR library [1]. ANGR is an open-source binary analysis library created for Python, which allows us to perform static analysis on binary outputs.

Our source code analysis framework is given in Figure 4.4. First, we compile the source code using a GCC compiler with the desired optimization options. Then, the binary executable file is fed into our framework. We first use

the `disassembly` function of ANGR to get the assembly instructions of the executable file. Then, we extract the CFG of the source code using the `CFGEmulated` function, which gives the basic block information (such as addresses, instructions inside these basic blocks, and edges between them) as shown in Figure 3.3.

After we get the CFG information, we extract the features from the basic blocks. Since we have the instructions inside the basic blocks, we generate node features using the static source code analysis. The node features help the GNN to learn the spatial information (the edges between nodes) and the structure of the node itself. The node features that we used are as follows:

- Number of arithmetic operations: This feature is the number of total arithmetic operations inside the basic block.
- Number of floating point arithmetic operations: Total number of floating point operations. We added this separately since floating point operations consume more time than the other arithmetic operations.
- Number of memory operations: Operations that require memory access like `MOV`, `POP`, `PUSH`, etc.
- Number of floating point memory operations: Floating point memory operations such as `MOVSS`, etc.
- Number of compare operations: Number of comparison operations that may lead to jumps and branches.
- Number of jump operations: Number of total jump operations that may lead to branches.
- Number of function operations: Total number of function calls inside the basic block.
- Number of return operations: Indicator of whether the basic block returns anything.
- Number of register accesses: Number of total register accesses like `RAX`, `RES`, `R11`, etc.

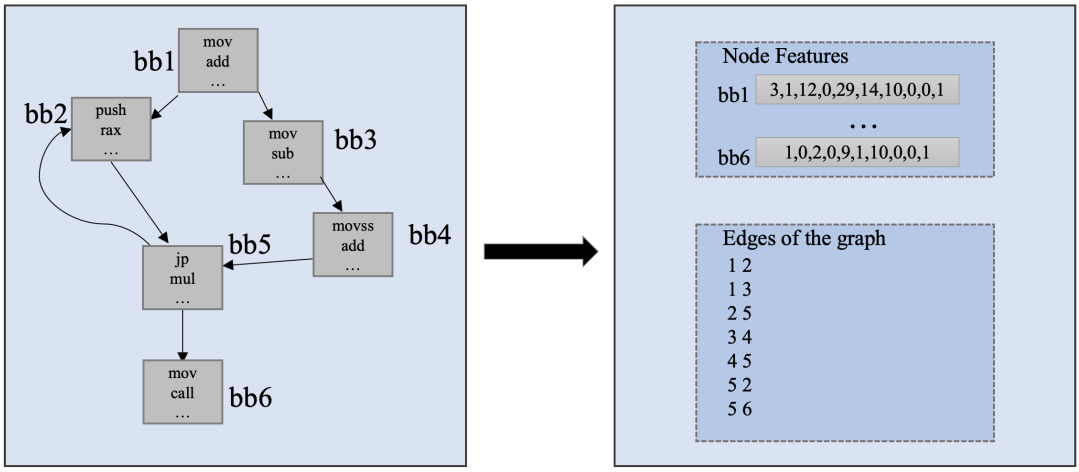

Figure 4.5: Example graph generated from the CFG and the feature vectors representing the nodes.

- Number of total instructions: Number of all instructions inside the basic block.
- Number of predecessors of the current basic block: This parameter shows the number of branches before that basic block. We use this information as an indicator of loops.
- Number of successors of the current basic block: This parameter shows the number of branches after that basic block if the basic block is a branching one. We use this information as an indicator of loops.
- Loop depth: How many loops cover the current basic block?

Using basic blocks as nodes, basic block features as node features, and connections between those nodes as edges, we create a directed graph for each output that we generate using the combinations of our binary flags. An example graph generated from the CFG and the feature vectors for the graphs is given in Figure 4.5.

Table 4.2 shows all the node features we generated with a graph ID and node ID. Features of each node in each graph are recorded here with their corresponding

Table 4.2: Node features for each graph is recorded to be used in the GNN model.

graph_id	node_id	arithmetic	arithmetic_FP	memory	memory_FP	compare	jump	call	ret	reg	number_of_ins	number_of_predecessors	number_of_successors	loop_depth
0	0	1	0	10	0	0	0	1	0	11	12	0	1	0
0	1	2	6	19	1	0	0	1	0	5	5	1	1	0
0	2	5	0	16	0	5	0	1	0	3	4	1	1	1
1	3	11	0	12	0	0	0	1	0	7	8	0	1	0
1	4	10	0	14	0	7	0	1	0	11	12	2	1	3
1	5	12	0	16	0	0	0	1	0	11	12	1	1	0

chart and node IDs so that the network can associate these features with the edge information during training.

The edge information is stored with a graph and node IDs as well. Edges between each node in each graph are recorded in a table similar to Table 4.3 using the same IDs in the node features table. These two tables are used as input to the GNN model.

Table 4.3: Edge information for the graph to be used in the GNN model.

graph_id	source	destination
0	1	2
0	2	3
0	2	5
0	3	4
1	1	3

4.3 Labeling Graphs

To train the GNN model, we need to label all the graphs we generated. Since we create a GNN model for each optimization flag, we need to label the dataset according to the effects of a single optimization flag on each program. For instance, for creating a model for `-funsafe-math-optimizations` flag, we get all the graphs that are not compiled with `-funsafe-math-optimizations`. By using the other seven flags, excluding `-funsafe-math-optimizations` flag, there are $2^7 = 128$ possible optimization sequences and therefore graphs that we can generate as a subset.

We then choose a graph, which we can call G_1. We modify the optimization sequence, add the `-funsafe-math-optimizations` flag, and get the new graph, G_2. After that, we compare the run times of those two graphs. If using `-funsafe-math-optimizations` reduced the run time, i.e., if the run time of G_2 is less than G_1, this means that G_1 favored the `-funsafe-math-optimizations` flag so we label G_1 as 1. If the run time of G_2 is more significant than G_1, we label G_1 as 0. The flow of the labeling strategy is given in Figure 4.6.

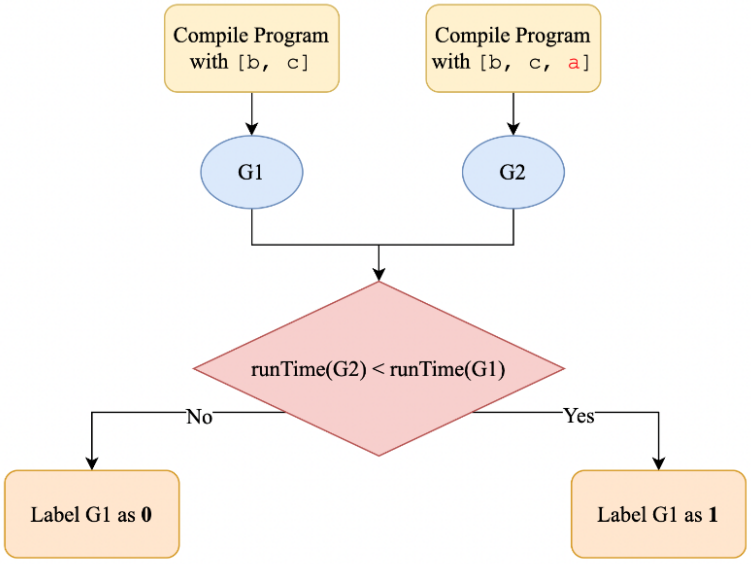

Figure 4.6: Flow of the graph labeling operation.

G_1 is the graph that is generated from the $G1$ application that is compiled as follows:

```
gcc -O2 -fivopts  -o G1
```

G_2 is the graph that is generated from the $G2$ application that is compiled as follows:

```
gcc -O2 -fivopts -funsafe-math-optimizations -o G2
```

We have 128×[Number of Applications] labeled graphs for a single optimization flag. Also, to prevent mislabeling due to slight differences in run time, we check whether the added flag improves the performance more than 10% of the initial run time (see Algorithm 1 for details). This way, we prevented the noises generated by the machine's state at run time.

Algorithm 1 Graph Labeling Algorithm

```
for f in all flags do
    combinations_without_f = all_combinations - combinations_with_f
    for combination in combinations_without_f do
        compile G1 with combination
        t1 = run time of G1
        new_combination = combination + f
        compile G2 with new_combination
        t2 = run time of G2
        if t1 − t2 > (t1 ∗ 0.1) then
            label of G1 = 1
        else
            label of G1 = 0
        end if
    end for
end for
```

After we label the graphs, we normalize the features of the nodes and create a data loader for the training. However, since some of the flags have unbalanced labels, before training, we balance the labels 50%-50% by oversampling. To increase the number of graphs in the small set, we randomly choose more samples until the two classes have equal samples.

4.4 GNN Parameter Tuning

The next step after generating the dataset is to tune the GNN model. GNN architecture consists of graph convolution layers, linear layers, and non-linear activation functions. We tested different network configurations with other parameters to measure the accuracy and the loss. The parameters that we tuned in the GNN model are as follows:

- **Number of Layers:** A total number of graph convolution layers in the network. As the complexity of the task increases, the number of layers can also be increased. However, using too many layers may lead to loss of information, known as *vanishing gradient problem* as well [58]. Therefore, this parameter should be tuned carefully according to its complexity. A *layer* contains a graph convolution module, a linear module, and a non-linear activation function.

- **Hidden Layers' Dimension:** The dimension of the output of the hidden units. Optimally, this value should be between the input and output dimensions for passing information inside the network.

- **Graph Pooling Type:** Aggregation type of the information between nodes [59]. It is used for converting the graph to a 1D vector at the end. Since pooling is used to obtain graph representations from node representations, this parameter defines how the network should combine all the node information [59]. We apply graph pooling after all the layers, in between all nodes, before the output layer. We use *(max)*, *(mean)*, and *(sum)* for graph pooling.

- **LR Scheduler - *Step Size*:** Learning rate (LR) scheduler is used for adjusting the learning rate during the training. While using a higher learning rate achieves faster training, keeping it too large may result in divergence. In such a scenario, the network may miss what is essential. On the other hand, using a minimal LR results in convergence but leads to very long training times. Therefore, the LR scheduler creates an adaptive learning

rate. LR starts high and after some time, it becomes smaller. *Step size* specifies the number of times the entire dataset should pass through the network (number of epochs) to decrease the learning rate. For example, step size 50 means in every 50 generations, LR will be reduced.

- **LR Scheduler - *Gamma*:** This parameter specifies the ratio of the decrease in the LR. For instance, gamma 0.1 and step size 50 means LR will be reduced to one-tenth in every 50 epochs. If Gamma is chosen too large, the learning rate may be reduced too quickly, leading to very slow convergence, while using it very small may not decrease the learning rate effectively, so the network may not learn the small details.

- **Weight Initialization Method:** This parameter shows whether we randomly generate the initial weights of the network or use a weight initialization method. The weight initialization is crucial for the network's convergence (completing the learning phase). If the weights are correctly initialized, the learning process will be much shorter and easier [60].

We trained our model for 500 epochs and recorded the accuracy and the loss values. **Accuracy** is calculated as the ratio of the correct labels among the labels predicted by the network. **Loss** is the output of the loss function, which specifies a penalty for an incorrect estimate [61]. We chose the configurations that lead to the highest accuracy and the lowest loss. The following sections describe the fine-tuning steps in detail.

To fine-tune to model, we split our dataset into the training, validation, and test sets. We used the 5-fold cross-validation technique for fine-tuning and created validation sets of 1000 graphs to calculate the accuracy and loss during the fine-tuning.

4.4.1 Number of Layers

To find the optimum number of layers, we tried different layers from 1 to 15. While changing the number of layers, we kept all other parameters fixed so that we could observe the change only caused by the number of layers. The changes in the average accuracy and the average loss with error bars indicating the different values in the different cross-validation sets are given in Figure 4.7. As can be seen, 10 is our dataset's optimum number of layers. Although more layers mean more features to learn and the node feature information can be propagated to other nodes, practical observations show that using more layers may drop the performance [62]. This is caused by *over smoothing* of the node features in deep networks [58].

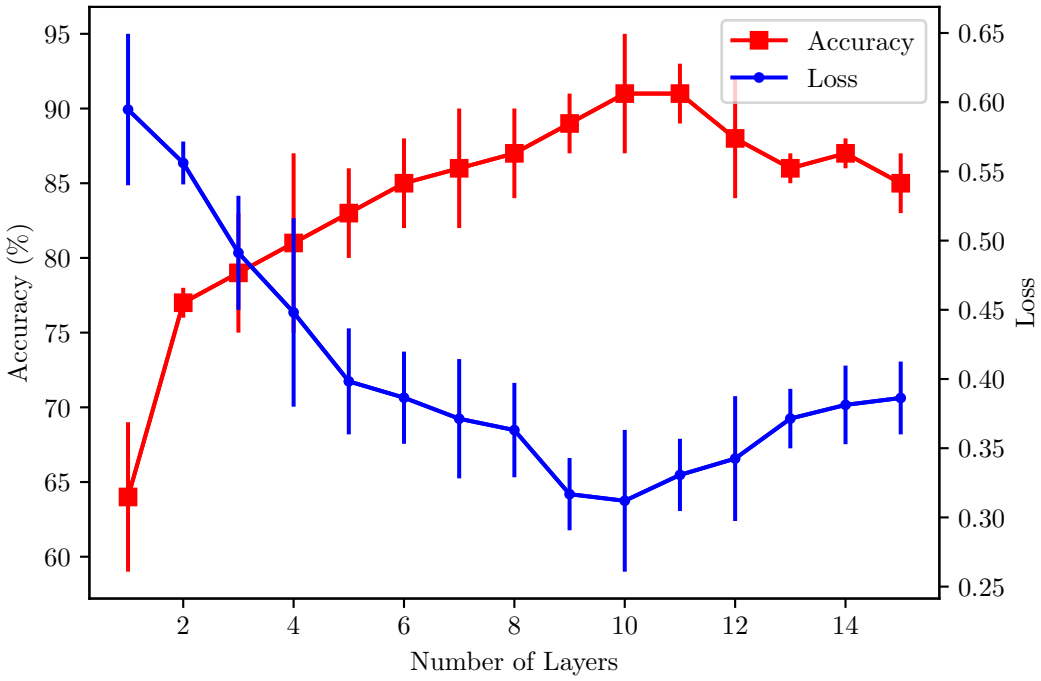

Figure 4.7: Accuracy/Loss vs. number of layers used in the GNN model.

4.4.2 Hidden Layers' Dimension

For this parameter, we tried different values from 1 to 8. We keep all other parameters fixed and change only the dimension of the hidden units. The changes in average accuracy and the loss given in Figure 4.8 indicate that the optimum dimension for our dataset is 5.

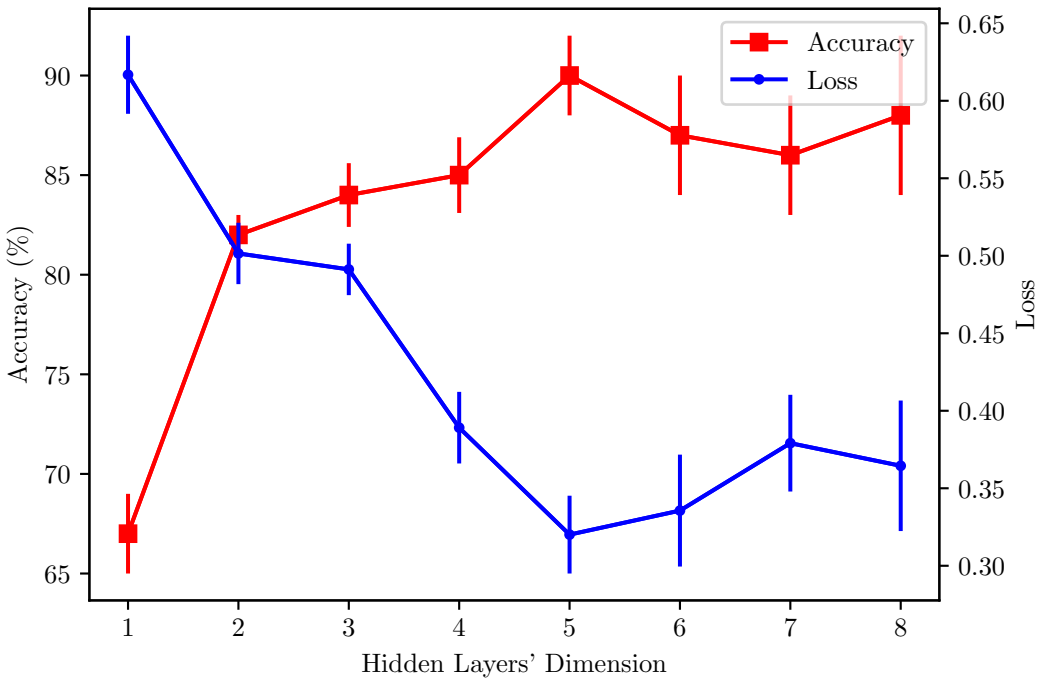

Figure 4.8: Accuracy/Loss vs. hidden layers' dimension used in our GNN model.

4.4.3 Pooling Types

We tried three possible pooling types for the graph pooling operation to find the best pooling type. Results for all pooling types are given in Figure 4.9. As can be seen, we are using *sum* as graph pooling is the optimum choice.

In networks where the number of neighbor nodes varies greatly, using sum as neighbor aggregation can lead to problems such as nodes with high degrees may

have much larger aggregation than nodes with low degrees [59].

In our case, since the number of neighbors is close in all nodes in the graphs that we generated from the source codes, instead of averaging them or taking the maximum of the node features, using *sum* as pooling type resulted in the best accuracy.

This is expected since the program spends most of its time on the basic blocks with the heaviest load. This follows from the fact that our features grow when the block is heavier with a higher number of nodes, instructions, etc. Therefore, as we sum all the information from the convolution operation, the aggregated information shows the computationally heavy parts of the source code, which allows the network to focus on the most time-consuming aspects.

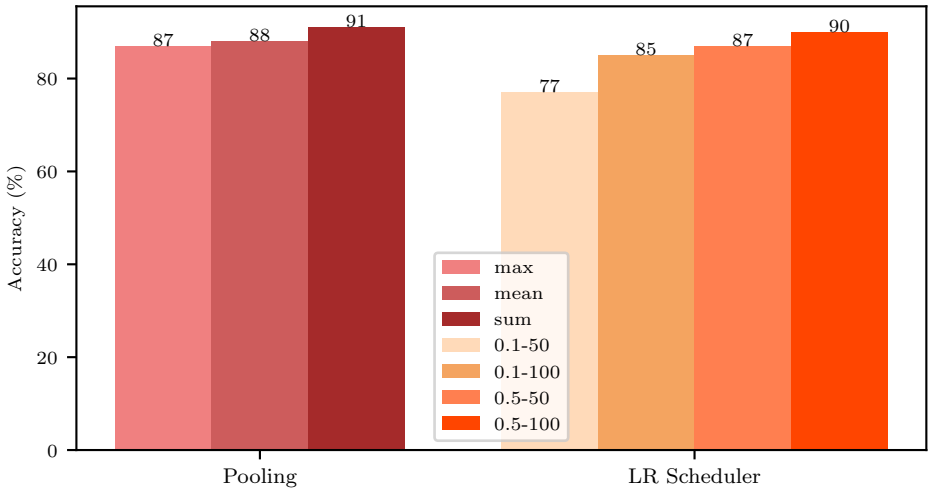

Figure 4.9: Accuracy vs. pooling types and LR scheduler parameters used in our GNN model.

4.4.4 LR Scheduler

To see the effects of the LR scheduler, we tried two different Gamma parameters with two different step sizes. All possible four pairs are given in Figure 4.9. In

this graph, values in the y-axis are the gamma and step size pairs, where the first value is the gamma and the second value is the step size for the LR scheduler.

In the best case, the LR scheduler step size is set to 100, and the gamma is set to 0.5. Since we are reducing LR 10 times, using 0.1 would result in a tiny LR, nearly stopping the learning process.

As an initial learning rate, we choose 10^{-4} since this value resulted in the best results in our fine-tuning runs.

4.4.5 Weight Initialization Method

Properly initialized weights increase the performance of the network dramatically [60]. Therefore, we tested our network with and without the Xavier weight initialization method. Results showed that Xavier weight initialization increased the accuracy from 82% to **91%**. This is because Xavier prevents weights from exploding or vanishing by keeping the variance the same across every layer in the network [63].

For instance, although the difference between 1.1 and 0.9 is just 0.2, in a network with 50 layers, when the weights are multiplied repeatedly, $1.1^{50} = 117930$ while $0.9^{50} = 0.00515$. The difference shows the importance of weight initialization, which we tried avoiding using Xavier [63].

4.4.6 Loss Function

As a loss function, we used *Cross Entropy Loss* since we have a binary classification task. Cross entropy measures the difference between the probability distribution of a machine learning classification model and the predicted distribution as given in Equation 4.1, where p is the predicted probability, y is the ground truth, and l is the loss [64].

Table 4.4: Final GNN model parameters.

# of Layers	Hidden Layers' Dimension	Neighbor Pooling Type	Graph Pooling Type	R Scheduler - Gamma	LR Scheduler - Step Size	Weight Initialization Method	**Accuracy**	**Loss**
10	5	max	sum	0.5	100	Xavier	92%	0.296

$$l = -(y \times log(p) + (1 - y) \times log(1 - p)) \tag{4.1}$$

4.4.7 Network Optimizer

We used *Adaptive Moment Estimation (Adam)* as an optimizer in our network. Adam optimizer is used in many ML applications using cross-entropy loss, and it is proven to be a successful algorithm for convergence where cross-entropy loss is used [65].

Our GNN model parameters obtained after experiments for fine-tuning are given in Table 4.4.

Chapter 5

Evaluation

We evaluate our scheme by measuring the compiled application's run time. We compare our approach against two different baselines. The first is $-O2$, and the second is *enhanced* $-O2$. We turn all the tested flags on in the enhanced $-O2$ case. Therefore, we use both extremes, where we have all flags enabled, and all flags disabled on top of $-O2$.

We created an extensive evaluation methodology to test the ideas and concepts we present thoroughly. The steps of our framework are as follows:

- Generate subsets
- Compare the run times
- Label graphs
- Train our GNN for that subset
- Test the model on the rest of the set

5.1 Setup

The details of the test setup we created to evaluate the ideas we discussed are given in the following sections.

5.1.1 Benchmark Datasets

We have used two benchmark datasets, Polybench and cBench. Polybench is a collection of benchmarks containing different numerical computations. These computations are extracted from domains like linear algebra, image processing, statistics, physics simulation, etc. [66]. The details of the Polybench benchmarks are given in Table 5.1.

The cBench dataset contains real-world applications with high computational requirements. It contains six categories [18]:

- **Automotive and Industrial:** This category demonstrates the use of embedded processors in control systems. The main focus is performance in basic math abilities and data manipulation.
- **Network:** Represents network devices like switches and routers. Includes shortest path calculations, tree lookups, etc.
- **Security:** Includes algorithms for data encryption, decryption, and cryptographic algorithms.
- **Consumer Devices:** Focuses on multimedia applications including video and audio.
- **Office Automation:** Text manipulation and word processing algorithms.
- **Telecommunications:** Voice encoding and decoding, frequency analysis algorithms.

Table 5.2 shows benchmarks under these categories and their descriptions.

Table 5.1: Polybench benchmarks and their properties.

Benchmark	**Description**	**Number of Basic Blocks**
adi	Alternating Direction Implicit solver	81
atax	Matrix Transpose and Vector Multiplication	71
bicg	BiCG Sub Kernel of BiCGStab Linear Solver	76
cholesky	Cholesky Decomposition	73
correlation	Correlation Computation	84
covariance	Covariance Computation	74
doitgen	Multiresolution analysis kernel (MADNESS)	78
durbin	Toeplitz system solver	74
dynprog	Dynamic programming (2D)	68
fdtd-2d	2-D Finite Different Time Domain Kernel	82
fdtd-apml	FDTD using Anisotropic Perfectly Matched Layer	104
floyd-warshall	Floyd–Warshall for finding shortest paths	62
gemver	Vector Multiplication and Matrix Addition	96
gesummv	Scalar, Vector and Matrix Multiplication	64
gramschmidt	Gram-Schmidt decomposition	101
jacobi-1D	1-D Jacobi stencil computation	60
jacobi-2D	2-D Jacobi stencil computation	67

Continued on next page

Table 5.1: Polybench benchmarks and their properties. (Continued)

Benchmark	**Description**	**Number of Basic Blocks**
lu	LU decomposition	66
ludcmp	LU decomposition	79
mvt	Matrix Vector Product and Transpose	69
reg-detect	2-D Image processing	84
seidel	2-D Seidel stencil computation	62
syr2k	Symmetric rank-2k operations	74
syrk	Symmetric rank-k operations	72
trisolv	Triangular solver	61
trmm	Triangular matrix-multiply	64

Table 5.2: cBench benchmarks and their properties.

Auto./Industrial	**Number of Basic Blocks**	**Consumer**	**Number of Basic Blocks**
susan (edges)	713	jpeg_c	1275
bitcount	62	lame	6174
susan (corners)	713	jpeg_d	1004
susan (smoothing)	713	tiff2rgba	2250
		tiffmedian	3137
Office	**Number of Basic Blocks**	**Network**	**Number of Basic Blocks**
stringsearch	119	dijkstra	129
bzip2d	4005		
bzip2e	4005		

Continued on next page

Table 5.2: cBench benchmarks and their properties. (Continued)

Security	**Number of Basic Blocks**	**Telecomm.**	**Number of Basic Blocks**
blowfish enc.	105	CRC32	44
blowfish dec.	105	ADPCM enc.	44
rijndael enc.	223	ADPCM dec.	38
rijndael dec.	223	GSM	818

5.1.2 Execution Environment

The compilation, source code analysis, and graph generation operations are done using a MacBook Pro (macOS Big Sur v11.0.0.1) with a 3.1 GHz Dual-Core Intel Core i7 CPU and 16 GB of memory. GNN training operations are done on a workstation with an Intel Core i9 CPU, 128 GB of memory, and NVIDIA RTX 3090 GPU.

For compilation, we used `gcc v11.2.0` [67]. Our Python scripts are written using Python 3.9. For binary analysis we used `angr v9.1.109` [1]. For GNN training operations, we have used `dgl v0.6.1` [68] and `torch v1.9.1` [69] libraries.

5.2 Accuracy Results

5.2.1 Effectiveness of GNN Model

We compared three different classifier models with our method to see how using a GNN model affects performance. We tested a 1-nearest neighbor classifier, similar to the one used in Milepost, and also trained a Deep Neural Network using our train dataset.

To fit a 1-nearest neighbor classifier, we flattened all of our node features and summed them as if we had a single feature vector for each program. For instance, instead of extracting the number of arithmetic operations for each basic block, we now have a total number of operations in the whole program as a feature. The accuracy results in the validation set are given in Figure 5.1.

Also, to observe the effect of spatial features, we trained a deep neural network model that also has ten layers. We again flattened the graphs and created a single feature vector for each program. As shown in Figure 5.1, using a deep neural network resulted in 81% accuracy. The results show that the GNN model outperformed both nearest neighbor and neural network models with the help of the spatial information that we extracted.

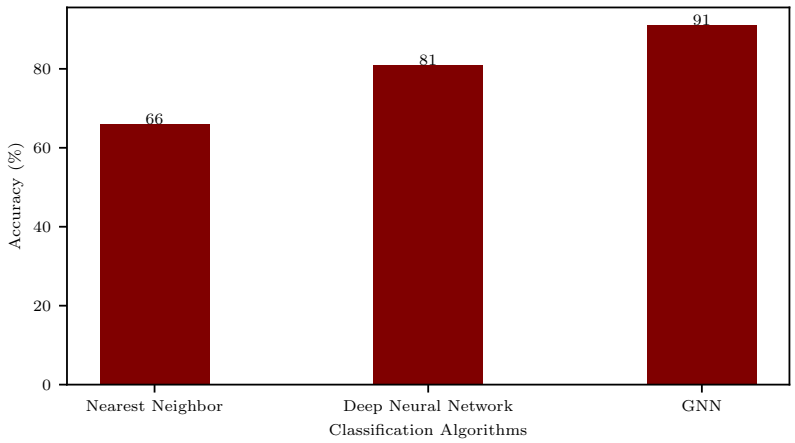

Figure 5.1: Comparison between different classification algorithms.

5.2.2 Results for Single-Flag Experiments

We trained our network with a single flag for this set of experiments. Labels used in this experiment are as follows:

- 0: The flag did not reduce the run time of the graph.
- 1: The flag reduced the run time of the graph.

We trained our network using the Polybench dataset alone and the cBench dataset alone and using both of them. The accuracy results for these tests are given in Figures 5.2, 5.3, and 5.4.

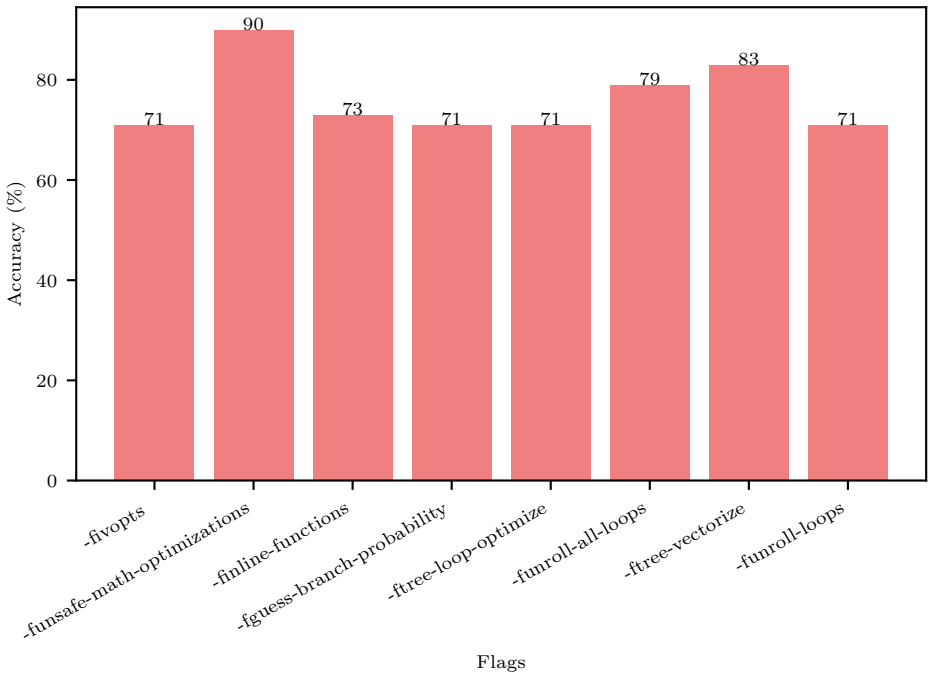

Figure 5.2: Single flag accuracy results for Polybench.

When we compare the accuracy results of the Polybench set, we observe that the best performance is achieved by the `-funsafe-math-optimizations` flag. As discussed below, the Polybench set consists of applications in domains like linear algebra and statistics. Therefore, the result shows that our model learned the patterns of math applications from the Polybench dataset, resulting in the best performance for `-funsafe-math-optimizations`. The second best flag, `-ftree-vectorize`, also supports our discussion. Moreover, since the Polybench set consists of matrix multiplication algorithms, we can say that our model learned when to unroll loops since we have 79% accuracy for the `-funroll-all-loops` as well.

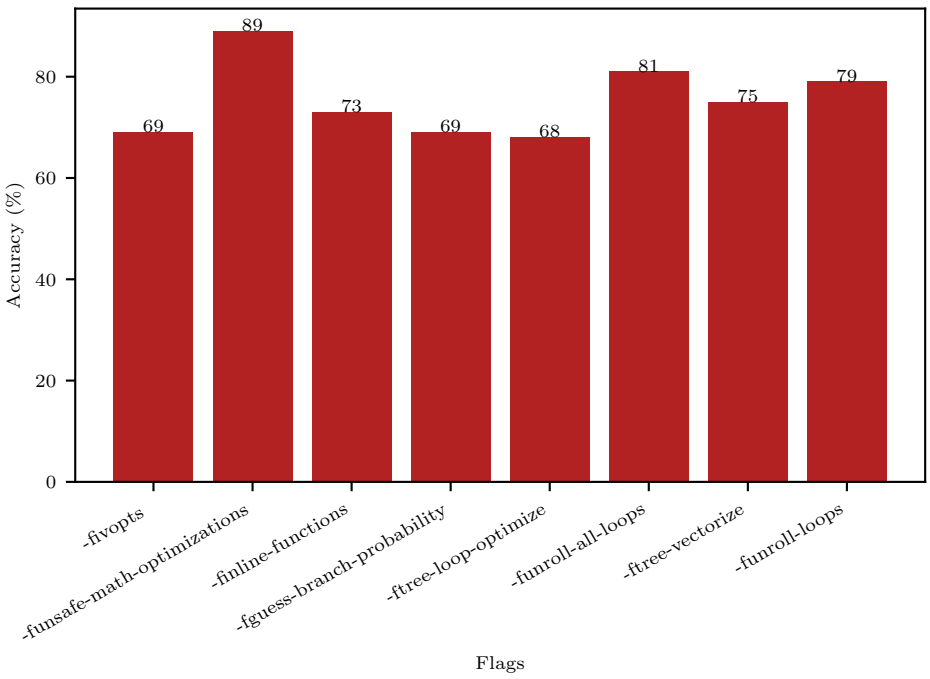

Figure 5.3: Single flag accuracy results for cBench.

The cBench results in Figure 5.3, show that the best performance is achieved again with the `-funsafe-math-optimizations`. Since both benchmark sets target CPUs, they contain computationally intensive algorithms. Therefore, our model mostly focused on those math applications, which is why it performs best with `-funsafe-math-optimizations`. However, for cBench set, `-funroll-all-loops` performs better than `-ftree-vectorize`. This shows that

applications with data-intensive operations like media applications in cBench helped our model learn patterns for `-funroll-all-loops` flag.

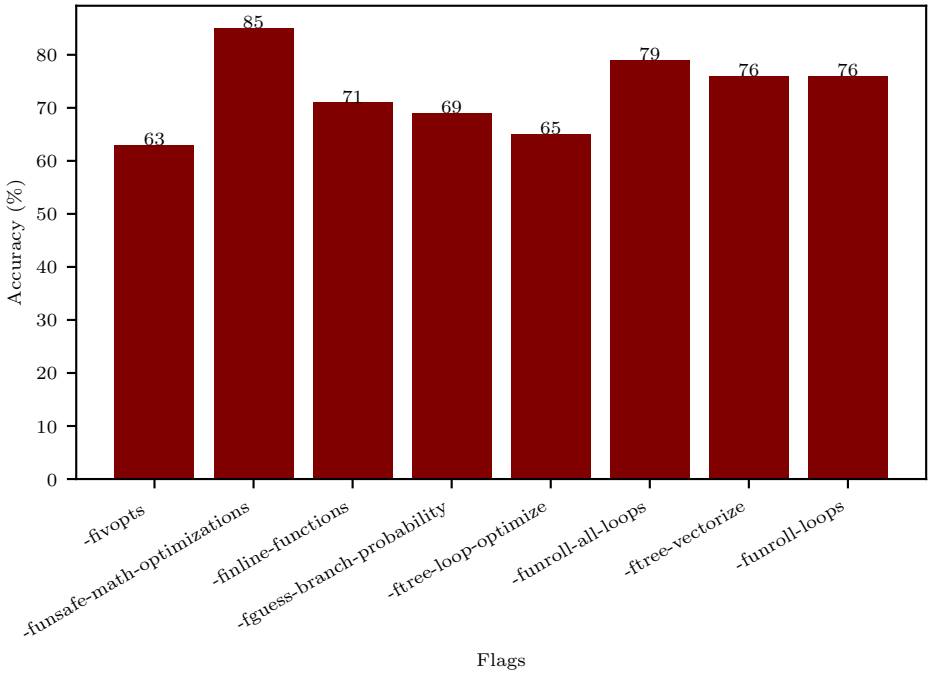

Figure 5.4: Single flag accuracy results for both benchmark sets.

When we train our model using both benchmark sets, we observe a 4% accuracy drop on average for all the flags, as seen in Figure 5.4. This is also expected because the network now tries to learn from small and large programs. Also, training in cross-domains is a challenging task as well [70]. While the model can learn similar patterns from the programs in the same domain (for instance, math applications), patterns may vary in mixed domains, thereby increasing the complexity of the problem. Therefore, the accuracy of the model on different domains decreases.

However, the amount of decrease in the accuracy is acceptable. We believe that a 4% drop is tolerable since the difference between the datasets is enormous.

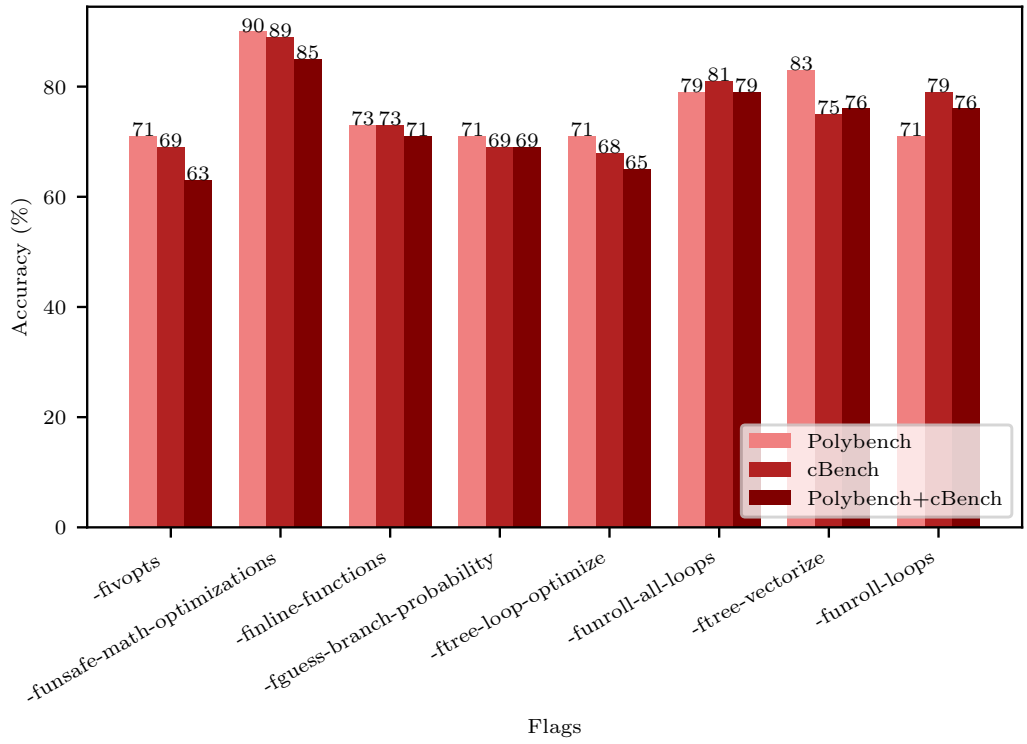

Figure 5.5: Single flag accuracy when trained with different benchmark sets.

We give the accuracy results of all the flags when trained with different benchmark sets in Figure 5.5. Results show that our network performed best on the Polybench set. This is relatively expected because Polybench includes simpler programs when compared to cBench. For instance, while a typical Polybench program contains around 100 nodes (basic blocks), a typical cBench program has more than 1000 nodes (even sometimes 6000 nodes). Therefore, some of the cBench programs are complex, so it is hard to predict from thousands of nodes, which results in a nearly 2% decrease in accuracy.

However, for some flags, the results are better for cBench. The reason for this behavior may be related to the domain of the datasets. While the Polybench contains mainly math applications (2D matrix multiplications, image processing tasks, etc.), the cBench contains sorting, zipping, and encryption algorithms. Therefore, while `-ftree-vectorize`, the flag that enables auto-vectorization

(vectorizes the convolution loops for faster execution) [71], performed significantly better on Polybench, `-funroll-all-loops`, a flag that speeds up the run time by unrolling all loops [2], performed better on the cBench.

Moreover, the models' average precision-recall (PR) curves are given in Figure 5.6. The PR curves show that for Polybench and cBench models, the average precision value is very close, and we obtain an f1 score approximately equal to 0.8 for both sets. The slight difference between the performance of Polybench and cBench is also observed here, but the f1 score and precision-recall values are very close.

Figure 5.6 shows the PR curve for both models. The 6% drop in average precision shows that the models performed relatively poorer than the single dataset models.

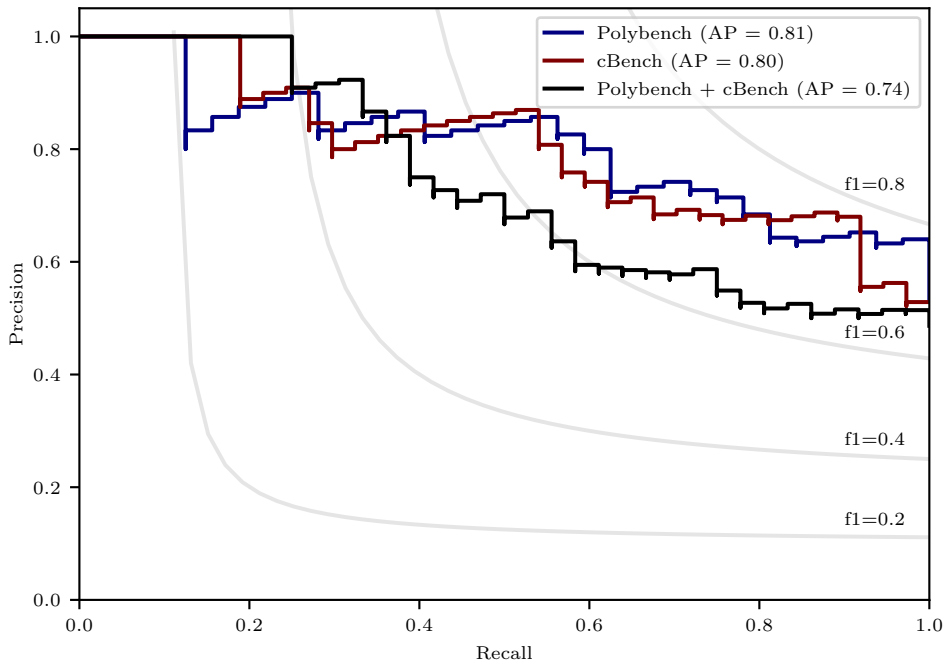

Figure 5.6: Precision-recall curves for single flag tests.

5.2.3 Results for Two-Flag Experiments

To perform a deeper analysis, we evaluate the model's inference performance with more than one flag. For this test, we use two flags in the model, and we try to find the best choice among four possible scenarios:

- 0: Using no flags results in the best run time.
- 1: Using only the first flag results in the best run time.
- 2: Using only the second flag results in the best run time.
- 3: Using both flags results in the best run time.

We labeled our graphs according to these choices with four labels. There are 28 possible two-flag combinations for eight flags, and we chose only 7 of them at the beginning for a proof of concept. After labeling, we train the model for Polybench, cBench, and both, as we did in the single flag tests. The accuracy results on test sets are given in Figures 5.7, 5.8, and 5.9.

Instead of deciding whether a particular flag will reduce the run time, we are now deciding if we should use the first, second, or both. Therefore, the model's primary purpose here is to find the best possible solution instead of just reducing the run time.

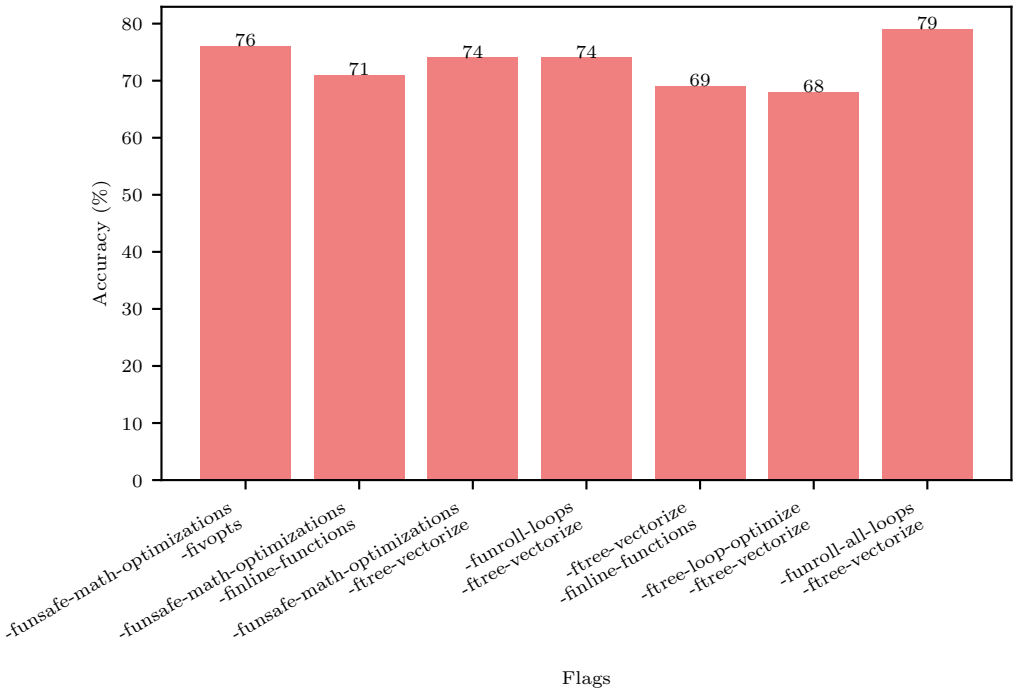

Figure 5.7: Two flag accuracy results for Polybench.

Figure 5.7 shows the accuracy results for two flags on the Polybench set. The plot shows that we have 73% accuracy for the two-flag model on average. Although this is not as successful as the single flag test, it is acceptable since the model has four labels.

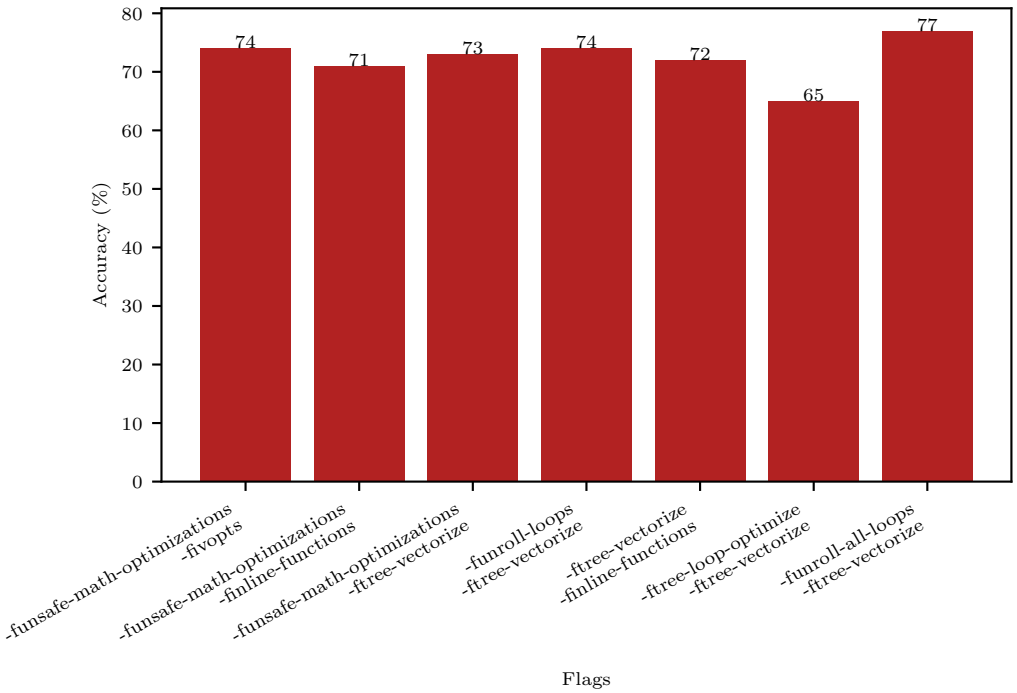

Figure 5.8: Two flag accuracy results for cBench.

We observe similar behavior for the cBench set. Figure 5.8 shows that we have 72% accuracy on average. Moreover, as shown in Figure 5.9, the accuracy of the combination of those two sets is very close to cBench and Polybench.

Results show that our model cannot only predict whether a flag can reduce the run time of a program but also choose between flags for the best performance. The model predicted the correct choice of 71.9% on average when two flags were given.

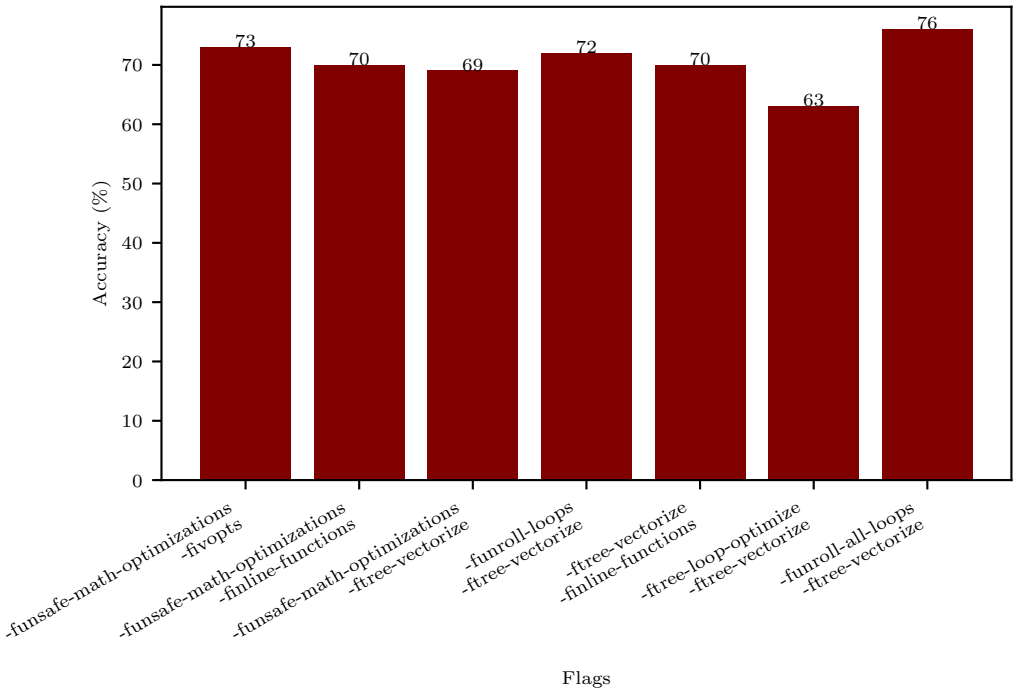

Figure 5.9: Two flag accuracy results for both sets.

We performed these experiments to observe the model's capabilities and to see if we can train the model so that it predicts not only if a flag can reduce the run time but also if the model predicts the best choice when we present more than two options. Note that among the four options, there may be more than one choice that results in a decrease in run time, but we tried to find the best option so that we can reduce the run time more aggressively.

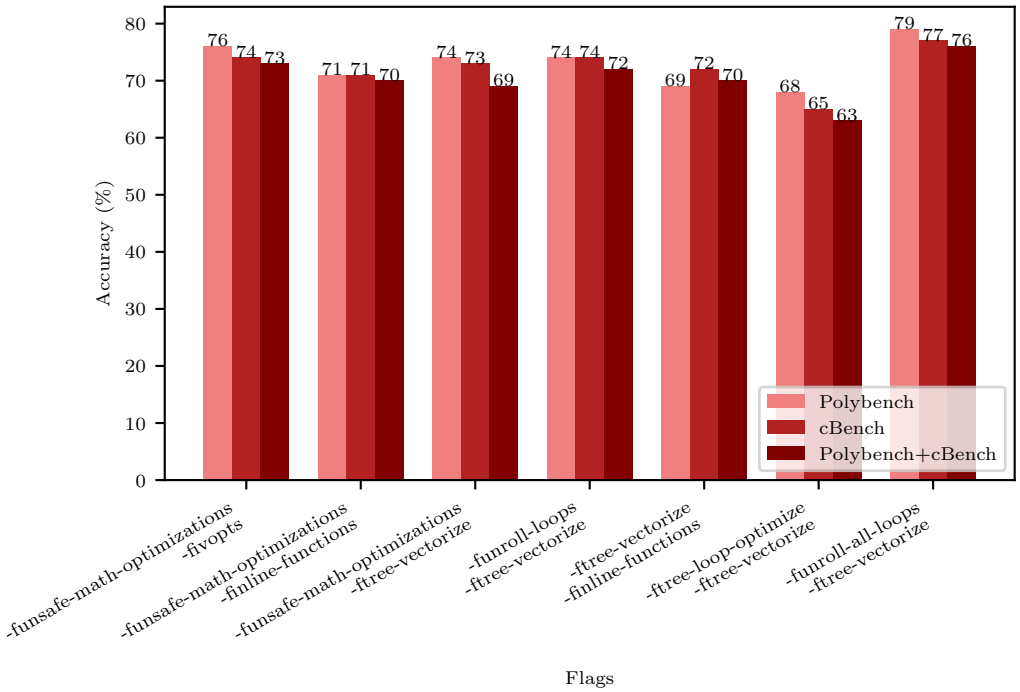

Figure 5.10: Two flag accuracy comparison with Polybench, cBench, and both benchmarks.

The flag correlation between two flag results' accuracy and single flag results has an interesting pattern. More specifically, as can be seen in Figure 5.10, while two flag accuracy for `-ftree-loop-optimize` and `-ftree-vectorize` is around 65%, the single flag accuracy for `-ftree-loop-optimize` is around 68%, and `-ftree-vectorize` is around 75%. The two-flag accuracy is close to the minimum single-flag accuracies among the pairs. We observe similar patterns for all the pairs. Therefore, we expect to see the same kind of results even with more than two flags. Although we did not perform experiments toward this goal, we believe getting results close to the prediction rates may be possible.

5.3 Performance Results

5.3.1 Single-Domain Results

We conducted tests using our benchmark sets to see how our model performs in real use cases. Our methodology is composed of the following steps for single-domain tests:

- Exclude one program from the dataset
- Create a training set with the rest of the programs in the dataset
- Train the model using the dataset
- Test the model using the excluded program

We first tested this approach on the Polybench dataset. The results of this test are given in the violin graphs in Figures A.1, A.2, A.3, and A.4. In these graphs, we show the possible run times of each program with $-O2$, enhanced $-O2$ (where we enabled all tested flags with the $-O2$) baselines, and our results. The percentage next to program names shows the speed-up our results achieved compared to the baseline (a negative value means that our result runs slower than the baseline).

Our approach generates the prediction result of the network for each program. After we compile the program, we feed it to the network and the network outputs a binary vector such as $[1, 0, 1, 1, 0, 0, 0, 1]$. This vector shows which flags should be used and which flags should not, where 1 means use the flag and 0 means not use it.

Using these vectors, we recompiled the programs and recorded the run times. To demonstrate the run time distribution of each program in the Polybench set, we include violin graphs for each program in the Appendix.

Figure 5.11 shows the speed-up percentages for each benchmark in the Polybench dataset. Our approach results in faster run times for 15 out of 26 programs

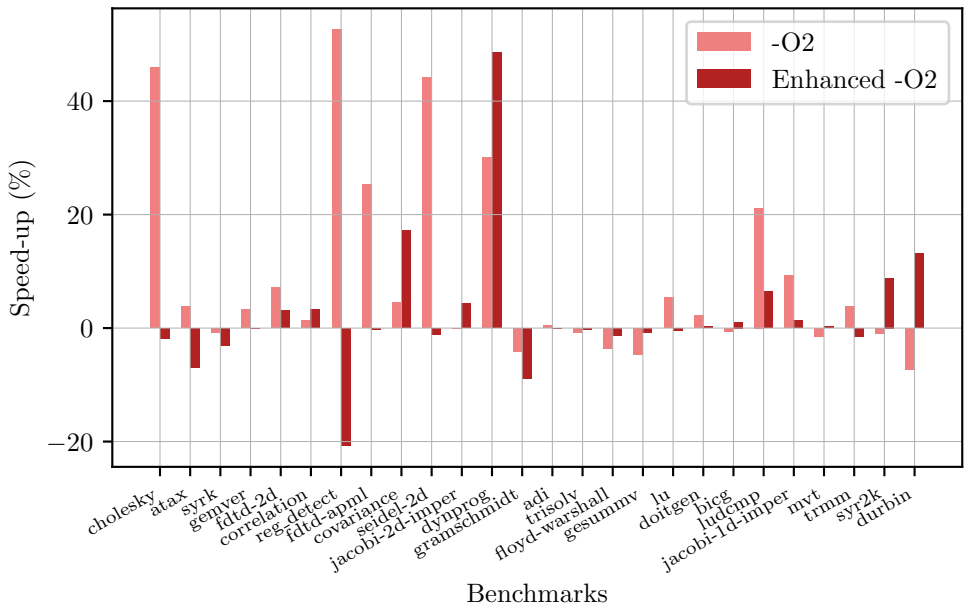

Figure 5.11: Speed-up for each benchmark in Polybench benchmark suite compared to $-O2$ and enhanced $-O2$ baselines.

when compared with $-O2$. The average speed-up compared to $-O2$ for the Polybench dataset is **9.06%** with a maximum of **52%**. In the worst case, we generate a 7% slow down compared to the $-O2$ run times.

When we compare our results with the enhanced $-O2$ where all the tested flags are enabled, it turns out that we get faster run time in 10 out of 26 programs. Our model's average speed-up for the Polybench dataset is **2.33%** with a maximum of **48.6%**. In the worst case, we generate a 20% slowdown compared to the case where all the tested flags are enabled. We observe the worst results for *reg_detect* benchmark because it has several loops with undefined loop counts. Furthermore, since the loop characteristics highly depend on the input data, the model could not predict the best flags for this benchmark.

For the 58% of the benchmarks in Polybench, our predictions are faster than $-O2$ with an average **16.93%** speed-up. Moreover, our predictions are faster than enhanced $-O2$ for 39% of all the benchmarks with an average of **10.1%**.

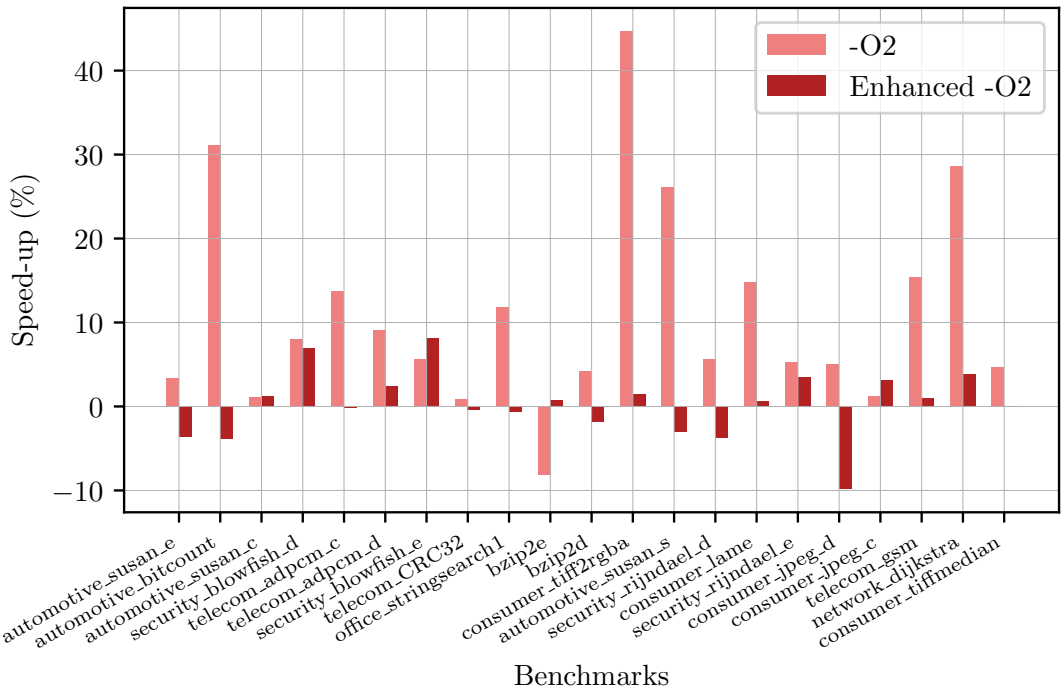

Figure 5.12: Speed-up for each benchmark in cBench benchmark suite compared to $-O2$ and enhanced $-O2$ baselines.

The speed-up results for the cBench set are given in Figure 5.12. In this benchmark set, we get better run time results than $-O2$ for 19 out of 21 programs. The average speed-up for the cBench dataset is **11.07%** with a maximum of **44%**. In the worst case, our approach results in an 8% slow down compared to the $-O2$ run time. We observe this result for the *consumer_jpeg_d* benchmark since this benchmark has several loops in which their loop counts highly depend on the input data, as we observed in *reg_detect* in our Polybench set.

When we compare our results with the enhanced $-O2$, we get better run times for 9 of 21 benchmarks in the cBench dataset. The average speed-up that we get is **0.28%** with a maximum of **8.01%**, and in the worst case, our approach results with a **9.87%** slow down compared to the all tested flags enabled case.

For 90% of the benchmarks in cBench, our results perform faster than $-O2$ with an average speed-up of **12.21%**. Moreover, our predictions are faster than

the enhanced $-O2$ for 43% of all the benchmarks with an average speed-up of **3.1%**.

The single domain results show that, on average, our model generates better run times when compared to $-O2$ and enhanced $-O2$. In the cases where our approach generated faster run time than the baselines, our average speed-up is better for the Polybench set. This is expected since benchmarks in Polybench are simpler programs than the cBench. Although our results generated faster results for more programs in cBench, the average speed-up is higher in the Polybench suite.

5.3.2 Cross-Domain Results

As a final test, we observe how our model performs in cross-domains. To measure the performance of our model, we train it using one dataset and measure the performance in the other. For instance, we trained the model using all programs in the Polybench dataset and tested the model using the cBench dataset, and vice-versa.

We first trained the model using the cBench dataset and tested it using the Polybench dataset. As seen from Figure 5.13, the flags predicted by the cBench model performed faster than $O2$ in 17 out of 26 programs. The average speed-up on the Polybench dataset achieved by the cBench model is **8.44%** with a maximum of **46%**. In the worst case, we get a 10% slow down in *syrk2*.

We get faster results for 8 out of 26 programs than the enhanced $-O2$ baseline. In this case, the average speed-up is **1.25%**, and the maximum speed-up is **48%**. In the worst case, our approach generates 35% slower results than the baseline in *reg_detect* benchmark as in the single domain test.

Figure 5.13 shows the speed-ups generated in the different domain tests using the cBench model on the Polybench suite. Our results are faster than the $-O2$ for the 65% of the benchmarks in the Polybench suite with an average of **13.46%**

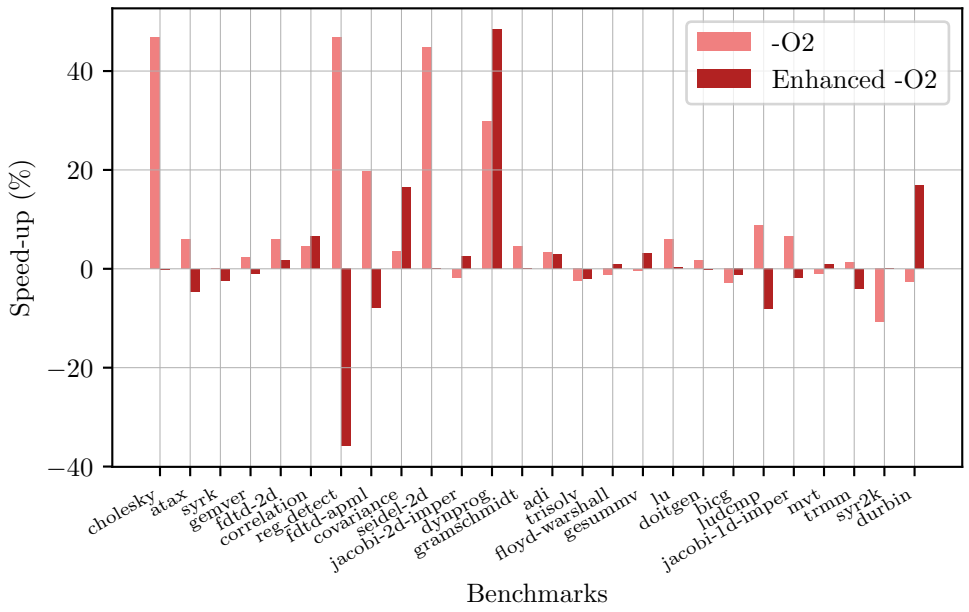

Figure 5.13: Speed-up using cBench model for each benchmark in Polybench suite compared to $-O2$ and enhanced $-O2$ baselines.

speed-up. When we compare it with the enhanced $-O2$, our approach generates faster results for **30%** of the benchmarks with an average of **9.87%** speed-up.

To see how the Polybench model performs on the cBench set, we trained our model using the Polybench set and tested it on the cBench set. As can be seen from Figure 5.14, 18 out of 26 programs run faster than the $-O2$ case with our predictions using the GNN model. On average, we get **10.36%** speed-up, and the maximum speed-up that we achieve is **44%**.

The Polybench model generated better run times than the enhanced $-O2$ for 8 out of 26 benchmarks. However, when we calculated the average speed-up compared to enhanced $-O2$, we got a **0.47%** slow down.

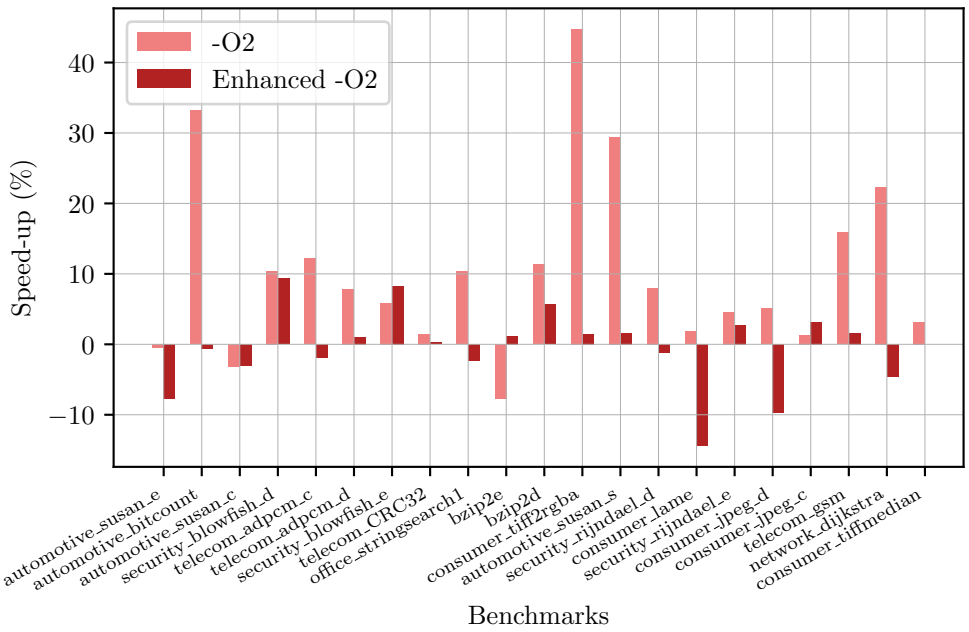

Figure 5.14: Speed-up using Polybench model for each benchmark in cBench suite compared to $-O2$ and enhanced $-O2$ baselines.

This was anticipated since the model generated using the Polybench suite is trained using much smaller and more basic programs than the cBench suite. Therefore, the Polybench model could not perform as well as the cBench model.

As seen from Figure 5.15, our model generated faster run times for both sets in single domain tests. The maximum achieved speed-up values show that our model predicted better when we trained and tested the model in the same domain (as we did in single-domain tests). More specifically, we get up to 52% faster run times with an average of approximately 10% when compared to $-O2$.

Moreover, our model beats the enhanced $-O2$ baseline, where we enabled all the tested flags with an average of 1.3% speed-up. This result shows that our GNN model can learn the patterns of different applications in a domain and generate an optimization flag set which results better than using all the flags in the set.

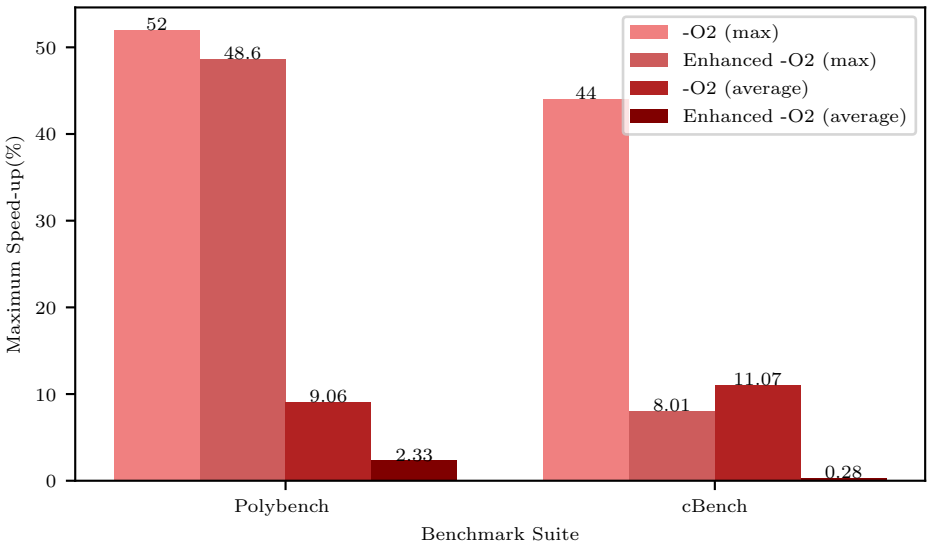

Figure 5.15: Speed-up in single domain tests compared to $-O2$ and enhanced $-O2$ baselines.

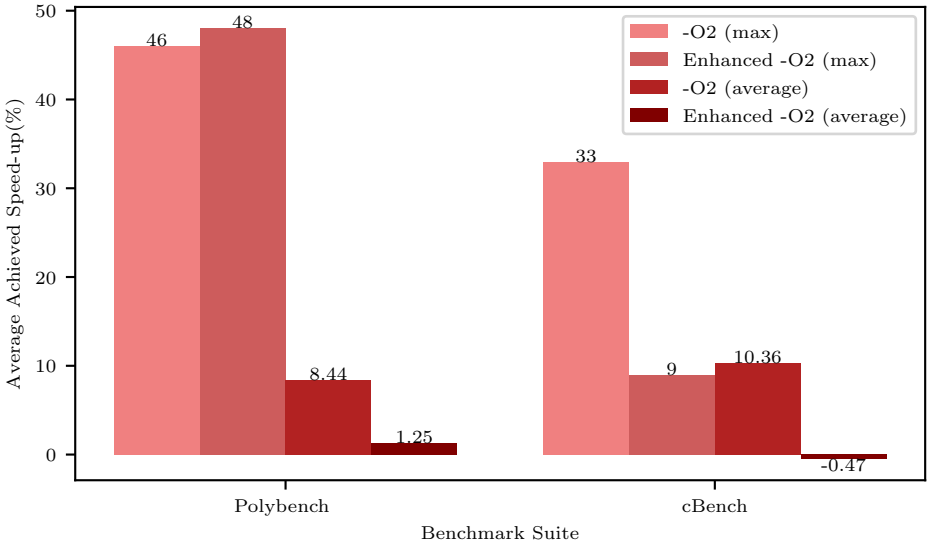

Figure 5.16: Speed-up in cross domain tests compared to $-O2$ and enhanced $-O2$ baselines.

Table 5.3: Comparison of our approach with the conventional approaches.

Approach	Maximum Speed-up	Average Speed-up
Genetic Algorithms [6]	34%	21.65%
Iterative Search [7]	40%	1.03%
GNN Approach (ours)	**52%**	**11%**

As seen from Figure 5.16, when we train a model in one domain and test it using another, we still beat both $-O2$ and enhanced $-O2$ with the maximum speed-up. When we compare the average speed-ups between the same domain set and different domain tests, it can be seen that although the models are trained using different domains, the reduction in average performances is nearly 1%. Using the cBench model, we achieved an 8.44% speed-up against the $-O2$ time and 1.25% speed-up against the enhanced $-O2$ time. This shows that even if the model is trained using a different domain, it can still generate better optimization set than enabling all the flags.

When we trained the model using the Polybench benchmark and tested it on cBench benchmark, although we managed to get better results than $-O2$, our average was below the enhanced $-O2$ time. This result was anticipated because the Polybench benchmark is rather small and contains relatively more straightforward algorithms when compared to the cBench benchmark. Therefore, the model cannot learn complex patterns that the cBench benchmark has, and we perform poorly compared to the case with all flags enabled.

Finally, we compare our results with the conventional methods used for automatic compiler optimization. The comparisons of maximum and average speed-ups achieved using different methods with respect to $-O2$ are given in Table 5.3.

Plotnikov et al. tuned and tested the genetic algorithm approach using popular open-source applications such as C-Ray, Crafty Chess, libevas, SciMark, x264, and zlib [6]. When they compared their results with the $-O2$ run times, they observed a maximum speed-up of 34%, which can be obtained in several hours to a few days, depending on the test application [6]. Therefore, when we compare the maximum speed-up, our GNN approach generates faster run times than the genetic algorithm approach in a short amount of time.

On the other hand, the iterative search algorithm approach is tuned and tested on a small benchmark set which consists of 6 optimization programs such as the ant colony optimization algorithm, search implementation for solving the quadratic assignment problem, and greedy algorithms for solving different problems [7]. The authors report that the iterative search approach can generate up to 40% faster run times on computationally intensive algorithms compared to the predefined optimization levels [7]. Therefore, when we compare the maximum speed-up and average speed-up, our GNN approach generates faster run times than the iterative search approach without needing to run and compile programs several times.

The results show that our approach achieves the best maximum speed-up compared to genetic algorithms and iterative search-based methods. Therefore, we can conclude that our approach can result in faster run times in a relatively short time compared to genetic algorithms and iterative search-based methods since these methods require thousands of iterations of compiling and running the program [21].

Chapter 6

Conclusion

Compilers offer a great variety of optimization options for reducing run time by consuming fewer resources. The compilers have hundreds of optimization options available to users to optimize the typical cases in the programs.

Although many options provide excellent benefits, choosing a good optimization sequence for a program is challenging. While it is hard to predict the effect of a single optimization flag, it is difficult to anticipate how an optimization sequence will affect a particular program before compiling it. To make it easier, some compilers provide predefined optimization levels for enabling generic optimization options. However, these options usually do not generate the best possible result. On the other hand, manual code optimization is a challenging and time-consuming task as well.

There are many efforts focused on finding an optimization sequence automatically by analyzing source code. Some of these prior works proposed using genetic algorithms, iterative search algorithms, etc. Although they provide good optimization sequences, they are relatively slow and require expert input. Due to these problems and improvements in machine learning research, using machine learning models for automatic code optimization became an active research area. Machine learning models are widely used in the literature by representing code

as fixed-length feature vectors. While some of these models focused on static features, others proposed using spatial features. In this thesis, we reviewed and classified the literature on automatic code optimization using static, spatial, and deep analysis methods and tried to combine their best features.

To extract all of the information from the source code, we analyzed the source code at basic block level. We extracted static code features from basic blocks and created a feature vector for each basic block in the source code using the CFG, which contains essential spatial features like loops, jumps, calls, etc. Since CFG shows the connections between the basic blocks, we got the spatial information from the CFG and created directed graphs using the basic blocks and edge information between them. Thanks to these directed graphs, we can represent the source code as a scalar vector.

We implemented and trained our model using eight optimization flags and two benchmark sets. We compiled 47 programs using each combination of the 8 flags and generated 12000 graphs using two benchmark suites, Polybench and cBench suites.

We created a GNN model and trained the model using the graphs we created. Unlike the SVM kernels, GNN learned much more complex patterns from the graphs thanks to its layered structure. As a result, we implement a GNN model that predicts an optimization sequence for a program without running it.

We first tested our models using a single flag and observed if the GNN model predicts whether a source code can favor an optimization flag. We were able to obtain an average accuracy of 75%. Furthermore, we tested the model using a sequence of 2 flags, and the results showed that our model could find the best possible sequence among four choices.

We performed single-domain tests on each benchmark. To do this, we excluded a program from the set, trained using others, and tested our model on that program. Our models generated better results than both $-O2$ and *enhanced* $-O2$ (where we enabled all the tested flags with $-O2$) baselines.

Additionally, we tested our model in different domains. We trained the GNN model in one benchmark and tested it on another. The model trained using a complex benchmark (cBench) performed well on the Polybench suite and achieved an average speed-up of 1.25% compared to the case where we enabled all the flags. This result showed that even though the domain is different, the GNN model can analyze the source code and propose better optimization sequences than enabling all of them.

Similarly, we trained the model using the more straightforward benchmark set (Polybench) and tested it on a more complex benchmark set (cBench). As a result, our model generated a 10% faster run time than the $-O2$ baseline, but on average, the results were 0.47% slower than the case where all flags are enabled. This was an expected result since the domains are different, and the test benchmark set contains much more complex programs.

In future work, to improve the cross-domain results, we plan to use *transfer learning*. When the training data is expensive or hard to collect, a model trained with easily collected data can be transferred to another domain by introducing less with the new data [72]. Therefore, we can prepare a base model using standard benchmarks like cBench and Polybench, train the base model using fewer data from specific domains, and create a particular model for different domains.

Our results show that the GNN model is not only capable of proposing suitable optimization flags in the single domain, but it is also capable of providing suitable optimization flags on different domains as well. Therefore we can conclude that static and spatial features successfully represent code, and GNN models can learn complex patterns from source code.

Bibliography

[1] F. Wang and Y. Shoshitaishvili, "Angr-the next generation of binary analysis," in *2017 IEEE Cybersecurity Development (SecDev)*, pp. 8–9, IEEE, 2017.

[2] G. Team, "Options that control optimization." https://gcc.gnu.org/onlinedocs/gcc-4.5.2/gcc/Optimize-Options.html.

[3] "Llvm's analysis and transform passes." https://llvm.org/docs/Passes.html.

[4] D. Branco and P. R. Henriques, "Impact of gcc optimization levels in energy consumption during c/c++ program execution," in *2015 IEEE 13th International Scientific Conference on Informatics*, pp. 52–56, IEEE, 2015.

[5] G. Fursin, Y. Kashnikov, A. W. Memon, Z. Chamski, O. Temam, M. Namolaru, E. Yom-Tov, B. Mendelson, A. Zaks, E. Courtois, *et al.*, "Milepost gcc: Machine learning enabled self-tuning compiler," *International journal of parallel programming*, vol. 39, no. 3, pp. 296–327, 2011.

[6] D. Plotnikov, D. Melnik, M. Vardanyan, R. Buchatskiy, R. Zhuykov, and J.-H. Lee, "Automatic tuning of compiler optimizations and analysis of their impact," *Procedia Computer Science*, vol. 18, pp. 1312–1321, 2013.

[7] L. Pérez Cáceres, F. Pagnozzi, A. Franzin, and T. Stützle, "Automatic configuration of gcc using irace," in *International Conference on Artificial Evolution (Evolution Artificielle)*, pp. 202–216, Springer, 2017.

[8] G. Sher, K. Martin, and D. Dechev, "Preliminary results for neuroevolutionary optimization phase order generation for static compilation," in *Proceedings of the 11th Workshop on Optimizations for DSP and Embedded Systems*, pp. 33–40, 2014.

[9] W. F. Ogilvie, P. Petoumenos, Z. Wang, and H. Leather, "Minimizing the cost of iterative compilation with active learning," in *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 245–256, IEEE, 2017.

[10] C. Blackmore, O. Ray, and K. Eder, "Automatically tuning the gcc compiler to optimize the performance of applications running on embedded systems," *arXiv preprint arXiv:1703.08228*, 2017.

[11] T. Young, D. Hazarika, S. Poria, and E. Cambria, "Recent trends in deep learning based natural language processing," *ieee Computational intelligenCe magazine*, vol. 13, no. 3, pp. 55–75, 2018.

[12] A. M. Malik, "Spatial based feature generation for machine learning based optimization compilation," in *2010 Ninth International Conference on Machine Learning and Applications*, pp. 925–930, IEEE, 2010.

[13] G. Pekhimenko and A. D. Brown, "Efficient program compilation through machine learning techniques," in *Software Automatic Tuning*, pp. 335–351, Springer, 2011.

[14] E. Park, J. Cavazos, and M. A. Alvarez, "Using graph-based program characterization for predictive modeling," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pp. 196–206, 2012.

[15] H. Peng, L. Mou, G. Li, Y. Liu, L. Zhang, and Z. Jin, "Building program vector representations for deep learning," in *International conference on knowledge science, engineering and management*, pp. 547–553, Springer, 2015.

[16] C. Mendis, A. Renda, S. Amarasinghe, and M. Carbin, "Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks," in *International Conference on machine learning*, pp. 4505–4515, PMLR, 2019.

[17] L.-N. Pouchet., "Polybench/c the polyhedral benchmark suite," 2012.

[18] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proceedings of the fourth annual IEEE international workshop on workload characterization. WWC-4 (Cat. No. 01EX538)*, pp. 3–14, IEEE, 2001.

[19] M. Werner, L. Servadei, R. Wille, and W. Ecker, "Automatic compiler optimization on embedded software through k-means clustering," in *2020 ACM/IEEE 2nd Workshop on Machine Learning for CAD (MLCAD)*, pp. 157–162, IEEE, 2020.

[20] K. Georgiou, C. Blackmore, S. Xavier-de Souza, and K. Eder, "Less is more: Exploiting the standard compiler optimization levels for better performance and energy consumption," in *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems*, pp. 35–42, 2018.

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

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

[23] S. Kulkarni and J. Cavazos, "Mitigating the compiler optimization phase-ordering problem using machine learning," in *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pp. 147–162, 2012.

[24] M. Namolaru, A. Cohen, G. Fursin, A. Zaks, and A. Freund, "Practical aggregation of semantical program properties for machine learning based optimization," in *Proceedings of the 2010 international conference on Compilers, architectures and synthesis for embedded systems*, pp. 197–206, 2010.

[25] G. Team, "Rtl - intermediate representation called register transfer language." https://gcc.gnu.org/onlinedocs/gcc-3.4.6/gccint/RTL.html.

[26] C. Blackmore, O. Ray, and K. Eder, "A logic programming approach to predict effective compiler settings for embedded software," *Theory and Practice of Logic Programming*, vol. 15, no. 4-5, pp. 481–494, 2015.

[27] A. Koseki, H. Komastu, and Y. Fukazawa, "A method for estimating optimal unrolling times for nested loops," in *Proceedings of the 1997 International Symposium on Parallel Architectures, Algorithms and Networks (I-SPAN'97)*, pp. 376–382, IEEE, 1997.

[28] J. Guen, C. Guillon, and F. Rastello, "Minir: a minimalistic intermediate representation," in *Workshop on Intermediate Representations (WIR'11), Chamonix, France*, 2011.

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

[30] O. Levi, "Pin - a dynamic binary instrumentation tool." https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html.

[31] K. Hoste and L. Eeckhout, "Microarchitecture-independent workload characterization," *IEEE micro*, vol. 27, no. 3, pp. 63–72, 2007.

[32] G. Fursin and O. Temam, "Collective optimization: A practical collaborative approach," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 7, no. 4, pp. 1–29, 2010.

[33] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather, "End-to-end deep learning of optimization heuristics," in *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 219–232, IEEE, 2017.

[34] Z. Zhang, S. Liu, Q. Yang, and S. Guo, "Semantic understanding of source and binary code based on natural language processing," in *2021 IEEE 4th Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC)*, vol. 4, pp. 2010–2016, 2021.

[35] K. W. Church, "Word2vec," *Natural Language Engineering*, vol. 23, no. 1, pp. 155–162, 2017.

[36] S. S. Du, K. Hou, R. R. Salakhutdinov, B. Poczos, R. Wang, and K. Xu, "Graph neural tangent kernel: Fusing graph neural networks with graph kernels," *Advances in neural information processing systems*, vol. 32, 2019.

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

[38] G. Fursin, M. F. O'Boyle, O. Temam, and G. Watts, "A fast and accurate method for determining a lower bound on execution time," *Concurrency and Computation: Practice and Experience*, vol. 16, no. 2-3, pp. 271–292, 2004.

[39] R. Baghdadi, M. Merouani, M.-H. Leghettas, K. Abdous, T. Arbaoui, K. Benatchba, *et al.*, "A deep learning based cost model for automatic code optimization," *Proceedings of Machine Learning and Systems*, vol. 3, pp. 181–193, 2021.

[40] F. Bodin, T. Kisuki, P. Knijnenburg, M. O'Boyle, and E. Rohou, "Iterative compilation in a non-linear optimisation space," in *Workshop on profile and feedback-directed compilation*, 1998.

[41] L. G. Martins, R. Nobre, J. M. Cardoso, A. C. Delbem, and E. Marques, "Clustering-based selection for the exploration of compiler optimization sequences," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 13, no. 1, pp. 1–28, 2016.

[42] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Massachusetts: Addison-Wesley, 1986.

[43] G. Team, "Gimple." https://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html.

[44] T. Mowry, "Optimizing compilers: Introduction," February 2002.

[45] L. Team, "Llvm's analysis and transform passes." https://llvm.org/docs/Passes.html.

[46] K. D. Cooper and L. Torczon, "Chapter 5 - intermediate representations," in *Engineering a Compiler (Second Edition)* (K. D. Cooper and L. Torczon, eds.), pp. 221–268, Boston: Morgan Kaufmann, second edition ed., 2012.

[47] G. Team, "Basic blocks." https://gcc.gnu.org/onlinedocs/gccint/Basic-Blocks.html.

[48] G. Team, "Control flow graph (cfg)." https://gcc.gnu.org/onlinedocs/gccint/Control-Flow.html.

[49] G. Montavon, W. Samek, and K.-R. Müller, "Methods for interpreting and understanding deep neural networks," *Digital signal processing*, vol. 73, pp. 1–15, 2018.

[50] A. I. Georgevici and M. Terblanche, "Neural networks and deep learning: a brief introduction," 2019.

[51] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and S. Y. Philip, "A comprehensive survey on graph neural networks," *IEEE transactions on neural networks and learning systems*, vol. 32, no. 1, pp. 4–24, 2020.

[52] S. Xiao, S. Wang, Y. Dai, and W. Guo, "Graph neural networks in node classification: survey and evaluation," *Machine Vision and Applications*, vol. 33, no. 1, pp. 1–19, 2022.

[53] M. Zhang and Y. Chen, "Link prediction based on graph neural networks," *Advances in neural information processing systems*, vol. 31, 2018.

[54] X. Wang and A. Vinel, "Benchmarking graph neural networks on link prediction," *arXiv preprint arXiv:2102.12557*, 2021.

[55] M. Defferrard, X. Bresson, and P. Vandergheynst, "Convolutional neural networks on graphs with fast localized spectral filtering," *Advances in neural information processing systems*, vol. 29, 2016.

[56] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[57] I. H. Sarker, "Deep learning: a comprehensive overview on techniques, taxonomy, applications and research directions," *SN Computer Science*, vol. 2, no. 6, pp. 1–20, 2021.

[58] S. Hochreiter, "The vanishing gradient problem during learning recurrent neural nets and problem solutions," *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, vol. 6, no. 02, pp. 107–116, 1998.

[59] Z. Wang and S. Ji, "Second-order pooling for graph neural networks," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2020.

[60] S. K. Kumar, "On weight initialization in deep neural networks," *arXiv preprint arXiv:1704.08863*, 2017.

[61] statistics.com, "Loss function." https://www.statistics.com/glossary/loss-function/, 2022. Accessed: 2022-12-01.

[62] K. Zhou, Y. Dong, K. Wang, W. S. Lee, B. Hooi, H. Xu, and J. Feng, "Understanding and resolving performance degradation in deep graph convolutional networks," in *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*, pp. 2728–2737, 2021.

[63] K. . Kunin, "Initializing neural networks," 2019. Accessed: 2022-12-08.

[64] S. Maheskar, "What is cross entropy loss?," 2019. Accessed: 2022-12-08.

[65] S. Bock and M. Weiß, "A proof of local convergence for the adam optimizer," in *2019 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8, IEEE, 2019.

[66] M. Á. Abella-González, P. Carollo-Fernández, L.-N. Pouchet, F. Rastello, and G. Rodríguez, "Polybench/python: benchmarking python environments with polyhedral optimizations," in *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*, pp. 59–70, 2021.

[67] G. Team, "Gcc 11 release series." https://gcc.gnu.org/gcc-11/changes.html, Oct 2022.

[68] M. Wang, D. Zheng, Z. Ye, Q. Gan, M. Li, X. Song, J. Zhou, C. Ma, L. Yu, Y. Gai, T. Xiao, T. He, G. Karypis, J. Li, and Z. Zhang, "Deep graph library: A graph-centric, highly-performant package for graph neural networks," *arXiv preprint arXiv:1909.01315*, 2019.

[69] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32* (H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, eds.), pp. 8024–8035, Curran Associates, Inc., 2019.

[70] C. Wang, Y. Liang, Z. Liu, T. Zhang, and S. Y. Philip, "Pre-training graph neural network for cross domain recommendation," in *2021 IEEE Third International Conference on Cognitive Machine Intelligence (CogMI)*, pp. 140–145, IEEE, 2021.

[71] G. Team, "Auto-vectorization in gcc." https://gcc.gnu.org/projects/tree-ssa/vectorization.html, Oct 2011.

[72] K. Weiss, T. M. Khoshgoftaar, and D. Wang, "A survey of transfer learning," *Journal of Big data*, vol. 3, no. 1, pp. 1–40, 2016.

Appendix A

Run Time Distribution of Tested Programs

The following violin graphs show the run time distribution for each program in each benchmark suite. In these graphs, we show the possible run times of each program with $-O2$, enhanced $-O2$ (where we enabled all tested flags with the $-O2$), and our approach. The percentage next to program names shows the speed-up our results achieved compared to the baseline (a negative value means that our result runs slower than the baseline).

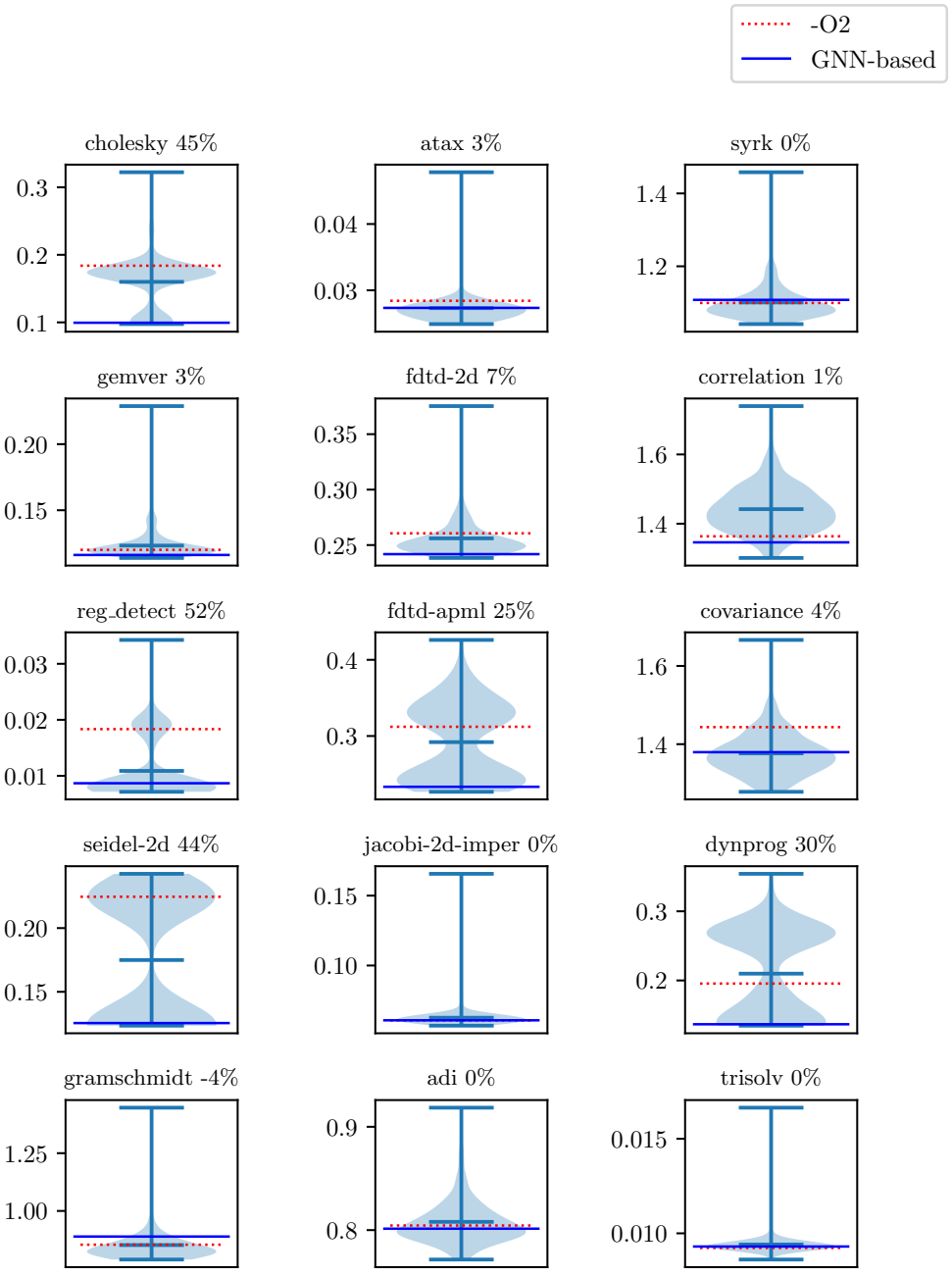

Figure A.1: Single domain test results for the Polybench benchmark suite for GNN-based and $-O2$.

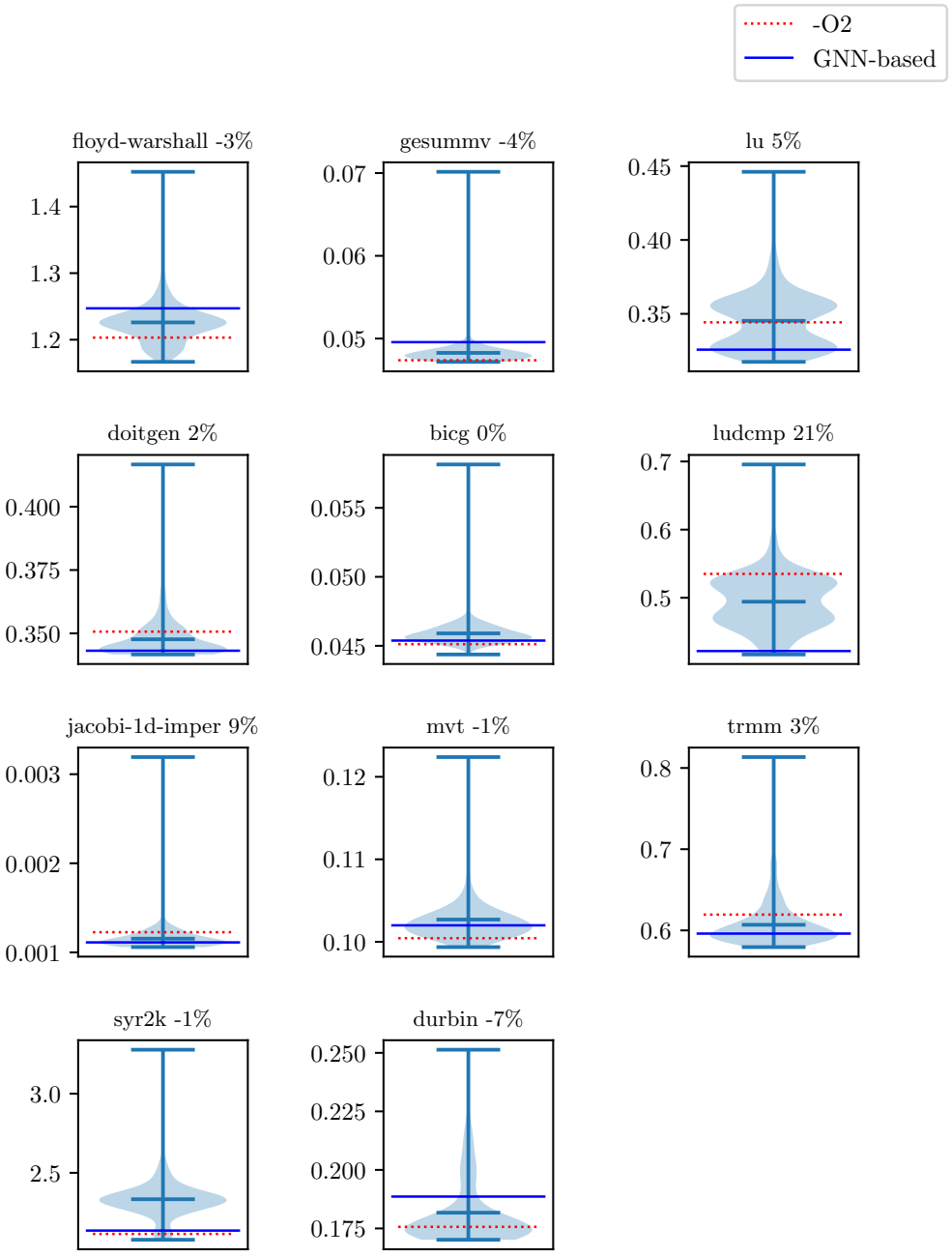

Figure A.2: Single domain test results for the Polybench benchmark suite for GNN-based and $-O2$ (continued).

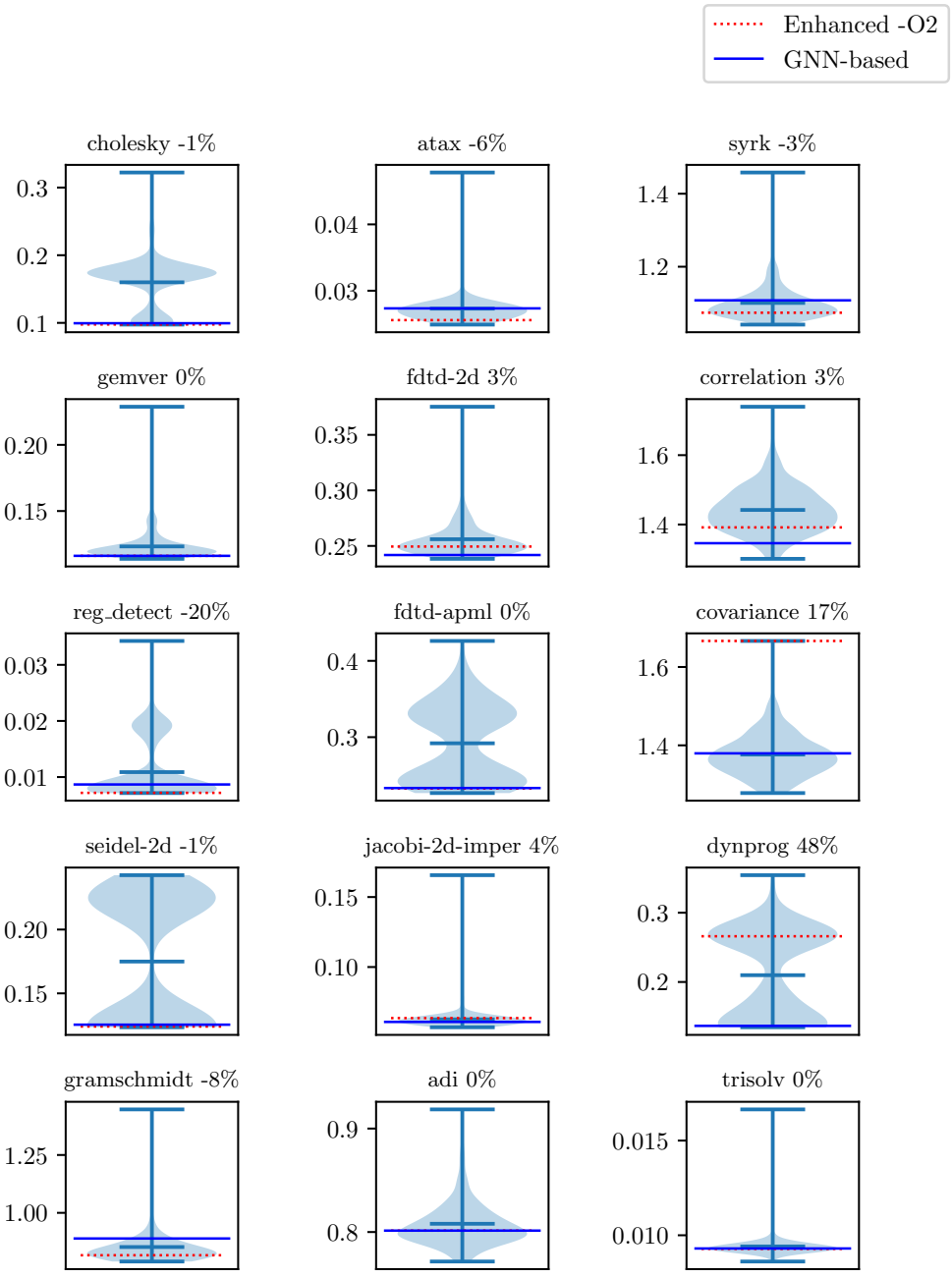

Figure A.3: Single domain test results for the Polybench benchmark suite for GNN-based and enhanced $-O2$..

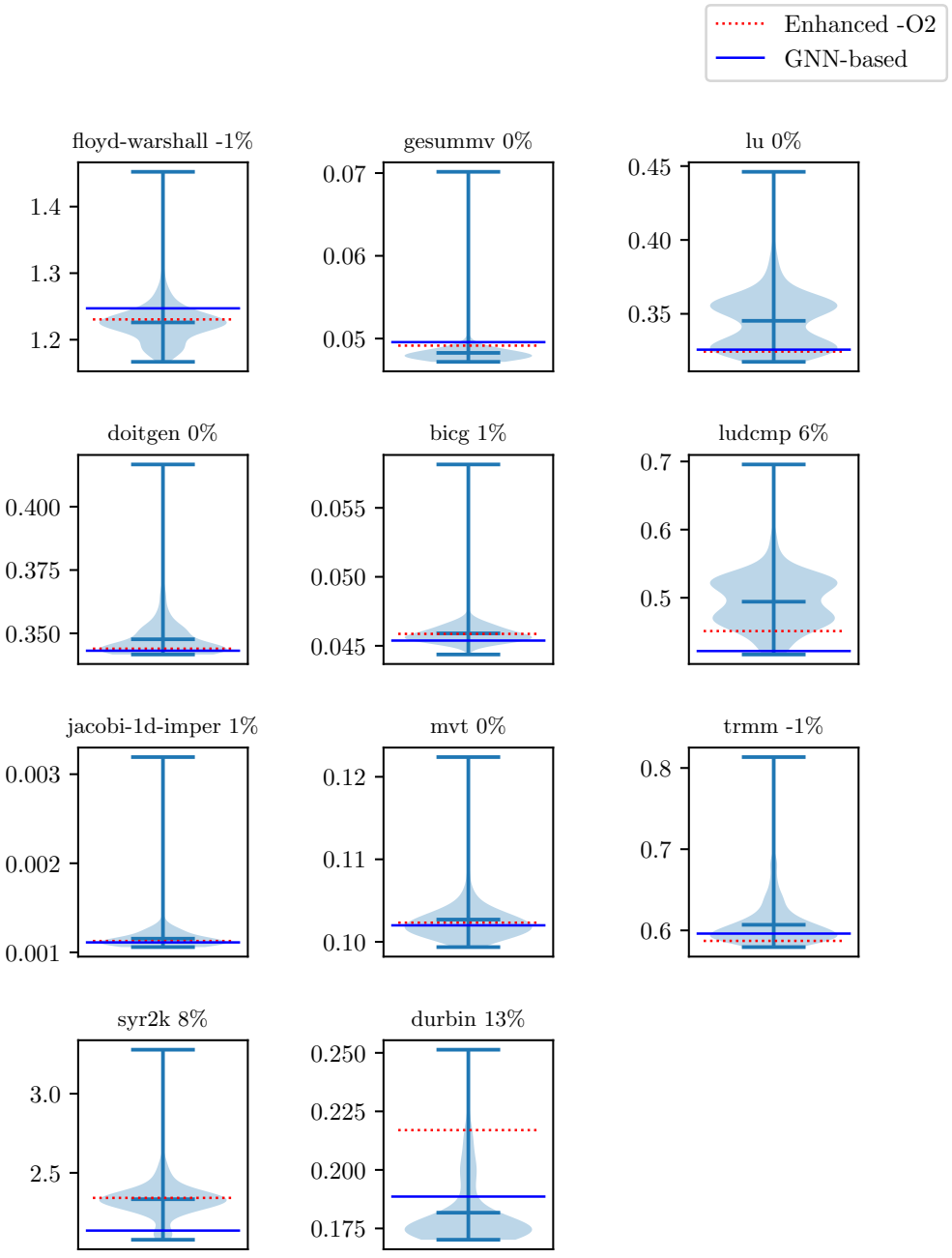

Figure A.4: Single domain test results for the Polybench benchmark suite for GNN-based and enhanced $-O2$.(continued).

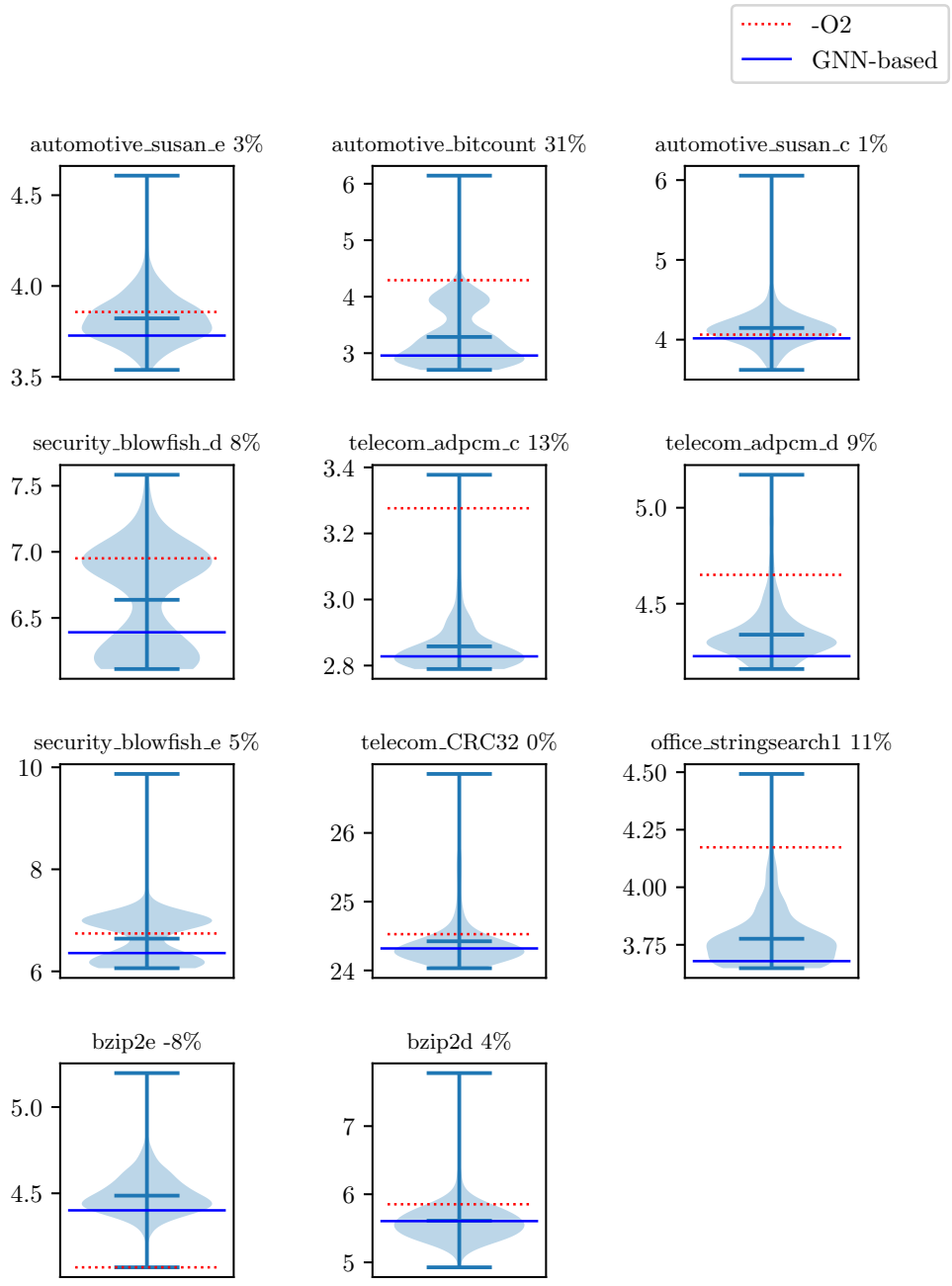

Figure A.5: Single domain test results for the cBench benchmark suite for GNN-based and $-O2$.

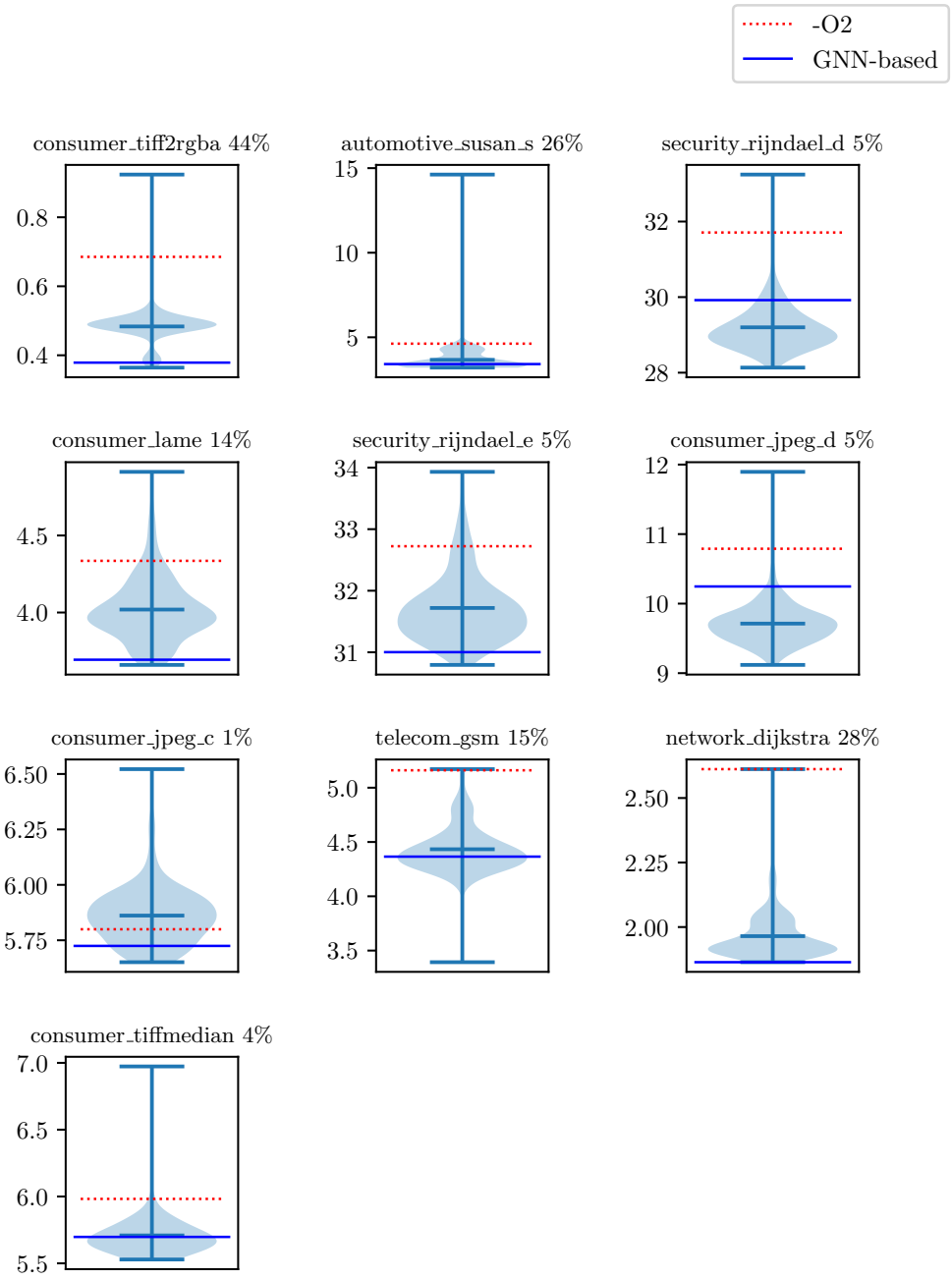

Figure A.6: Single domain test results for the cBench benchmark suite for GNN-based and $-O2$ (continued).

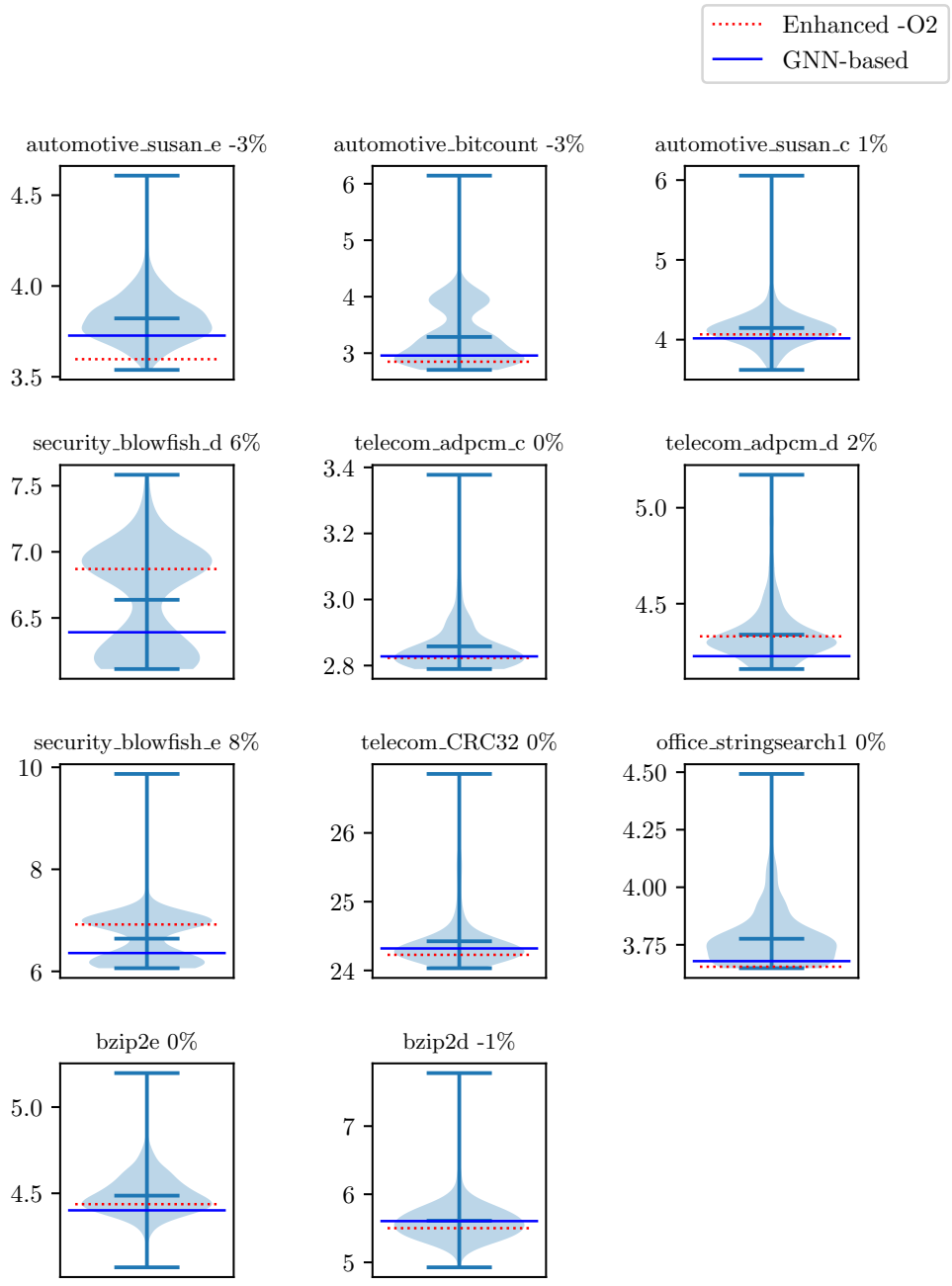

Figure A.7: Single domain test results for the cBench benchmark suite for GNN-based and enhanced $-O2$.

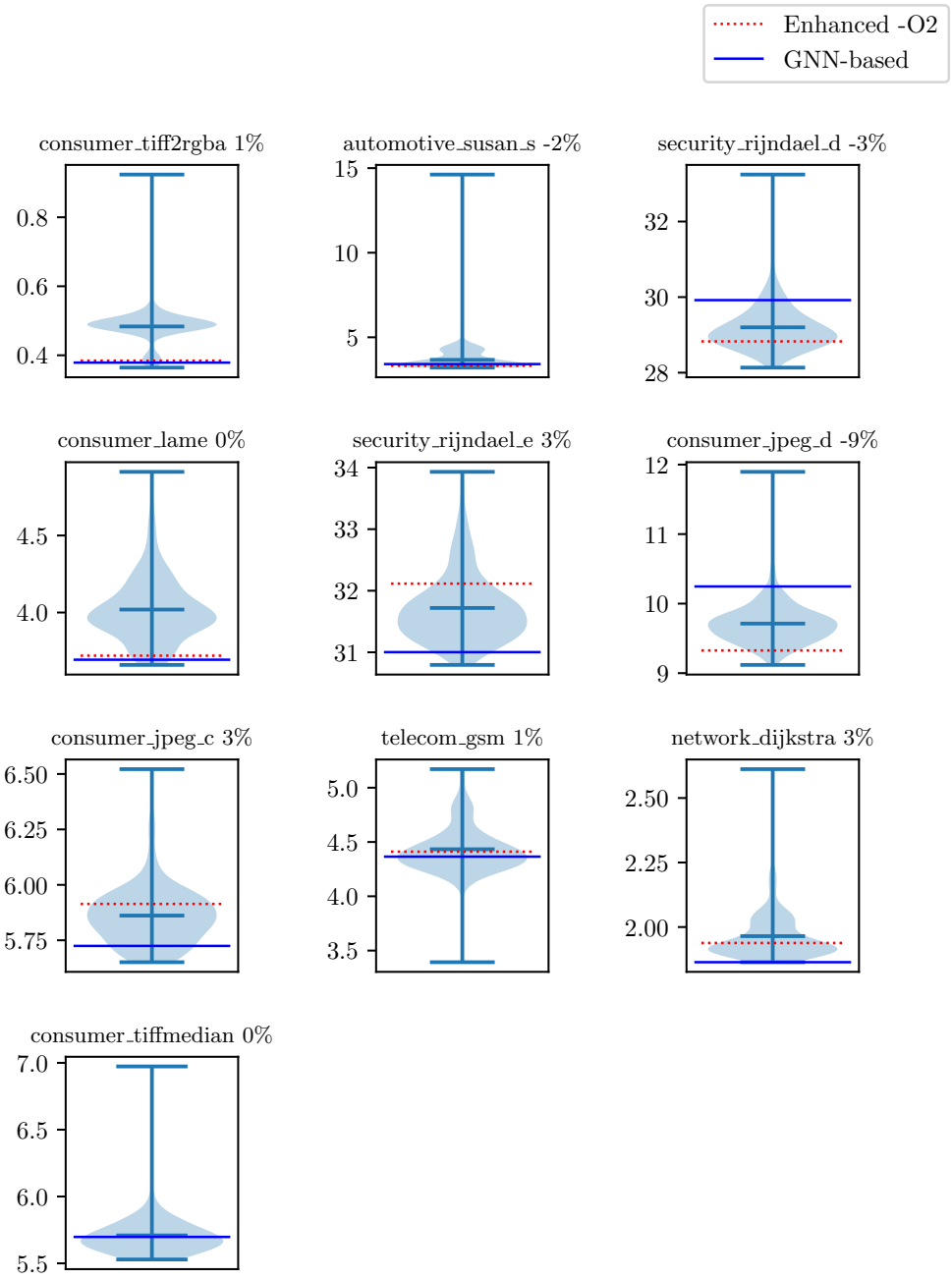

Figure A.8: Single domain test results for the cBench benchmark suite for GNN-based and enhanced $-O2$ (continued).

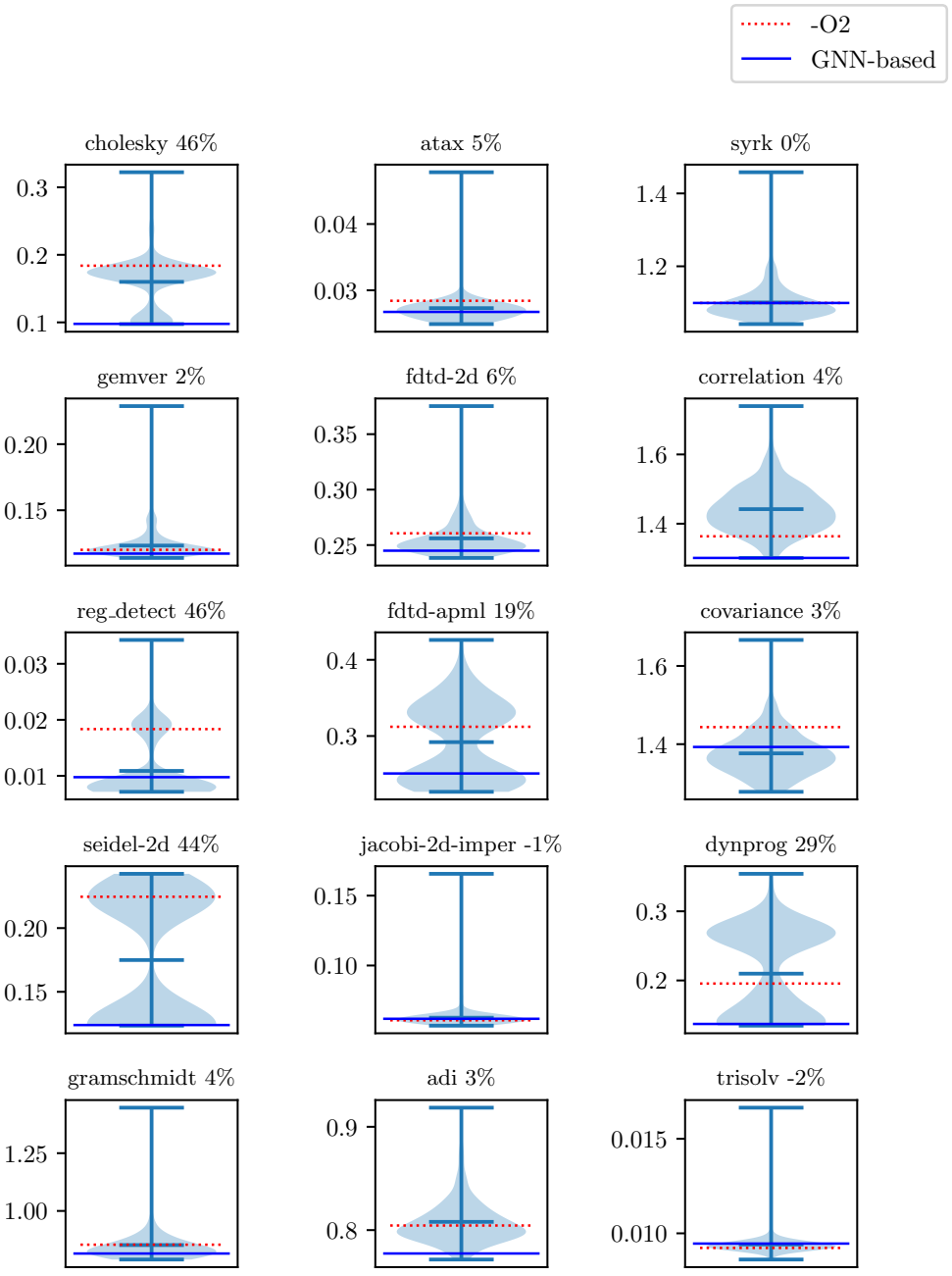

Figure A.9: Cross domain test results of cBench model on Polybench benchmark suite for GNN-based and $-O2$.

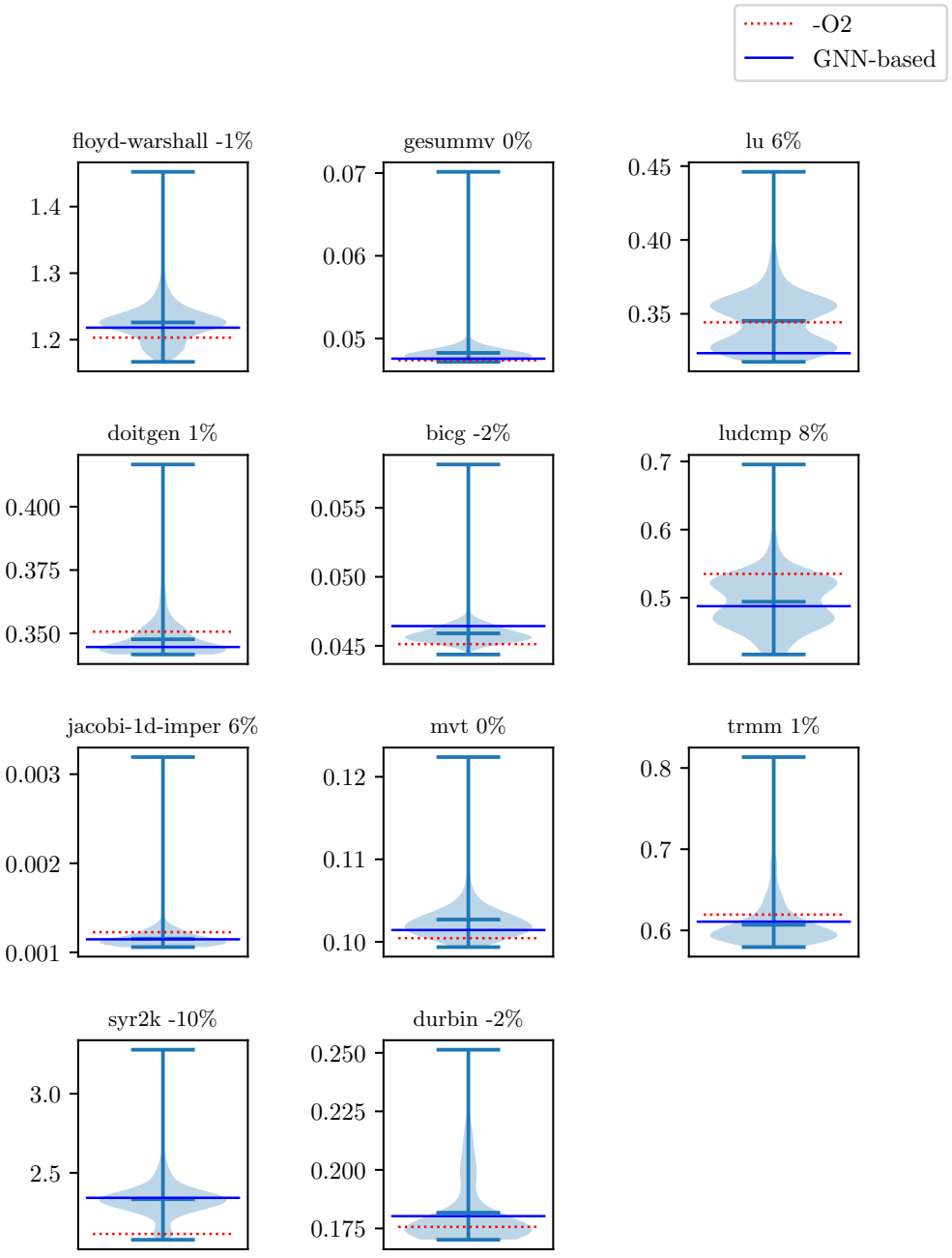

Figure A.10: Cross domain test results of cBench model on Polybench benchmark suite for GNN-based and $-O2$ (continued).

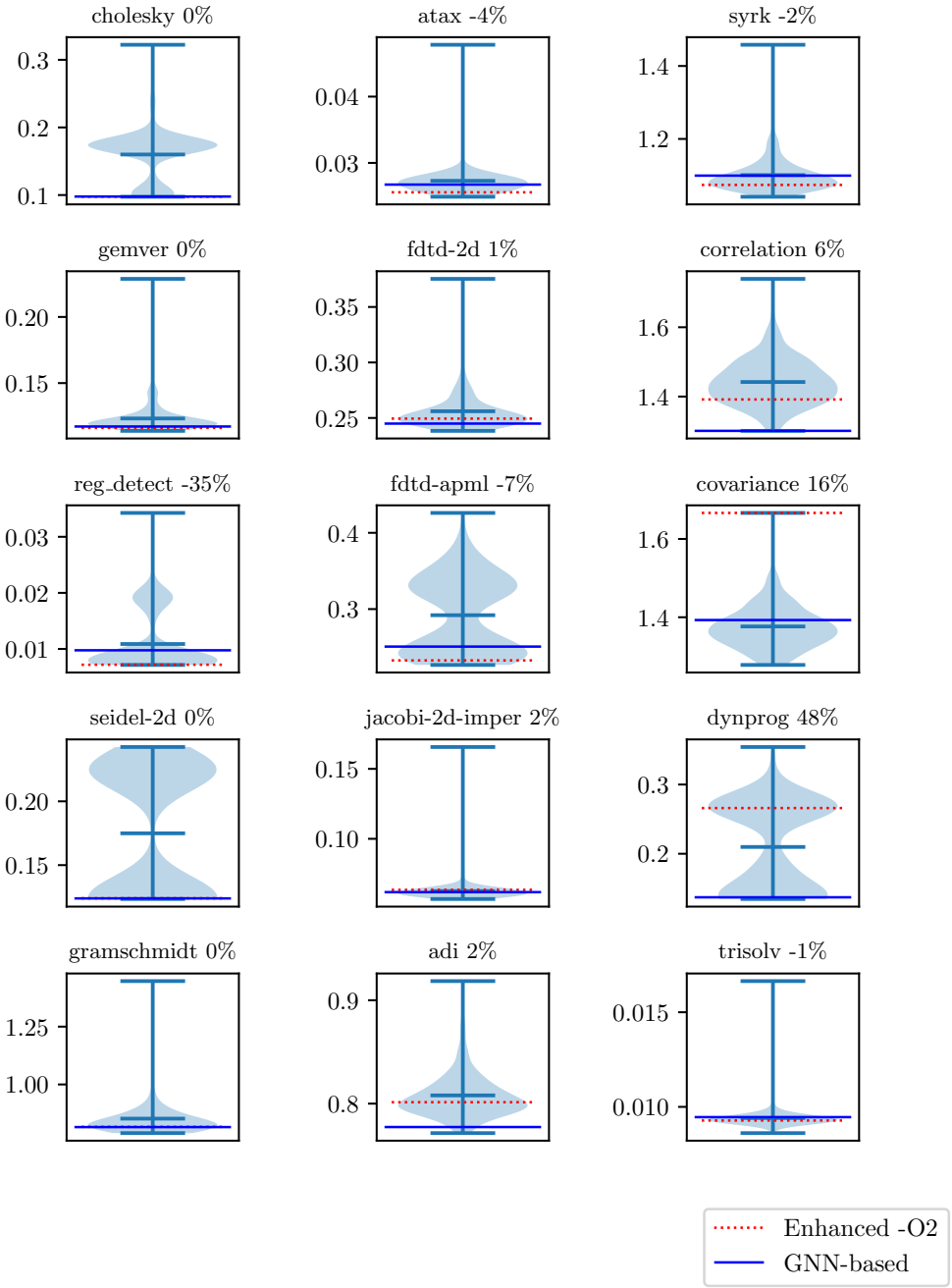

Figure A.11: Cross domain test results of cBench model on Polybench benchmark suite for GNN-based and enhanced $-O2$.

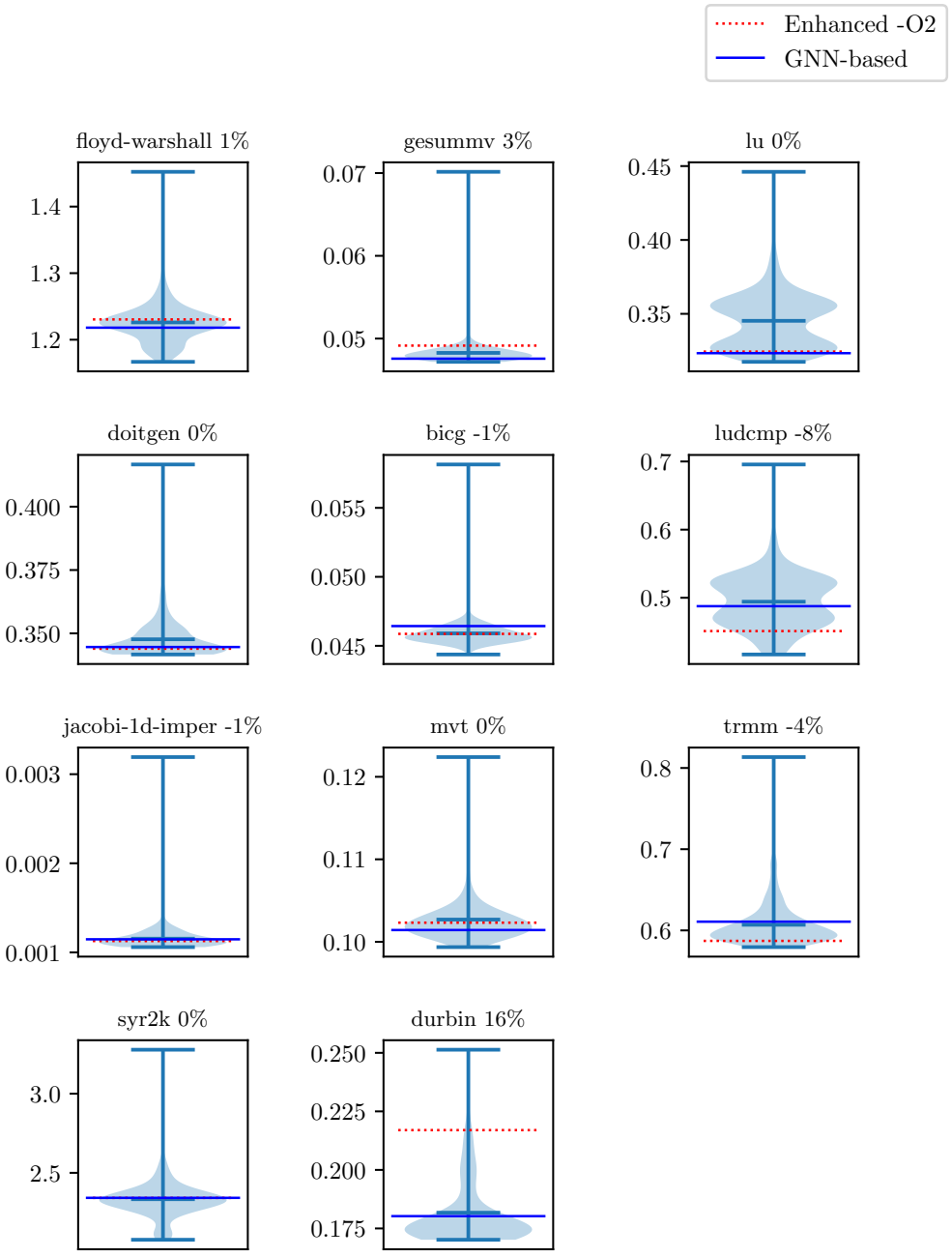

Figure A.12: Cross domain test results of cBench model on Polybench benchmark suite for GNN-based and enhanced $-O2$ (continued).

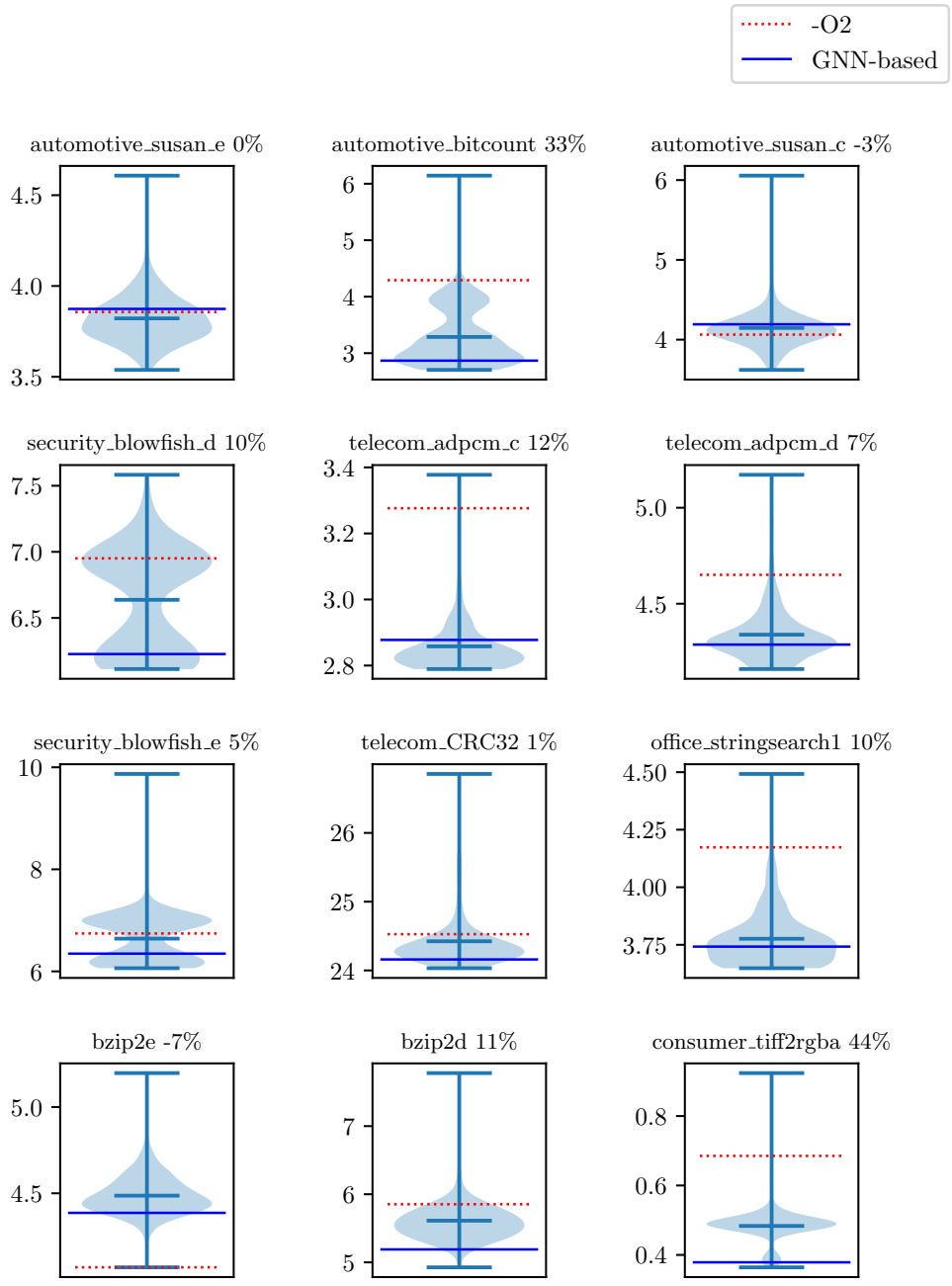

Figure A.13: Cross domain test results of Polybench model on cBench benchmark suite for GNN-based and $-O2$.

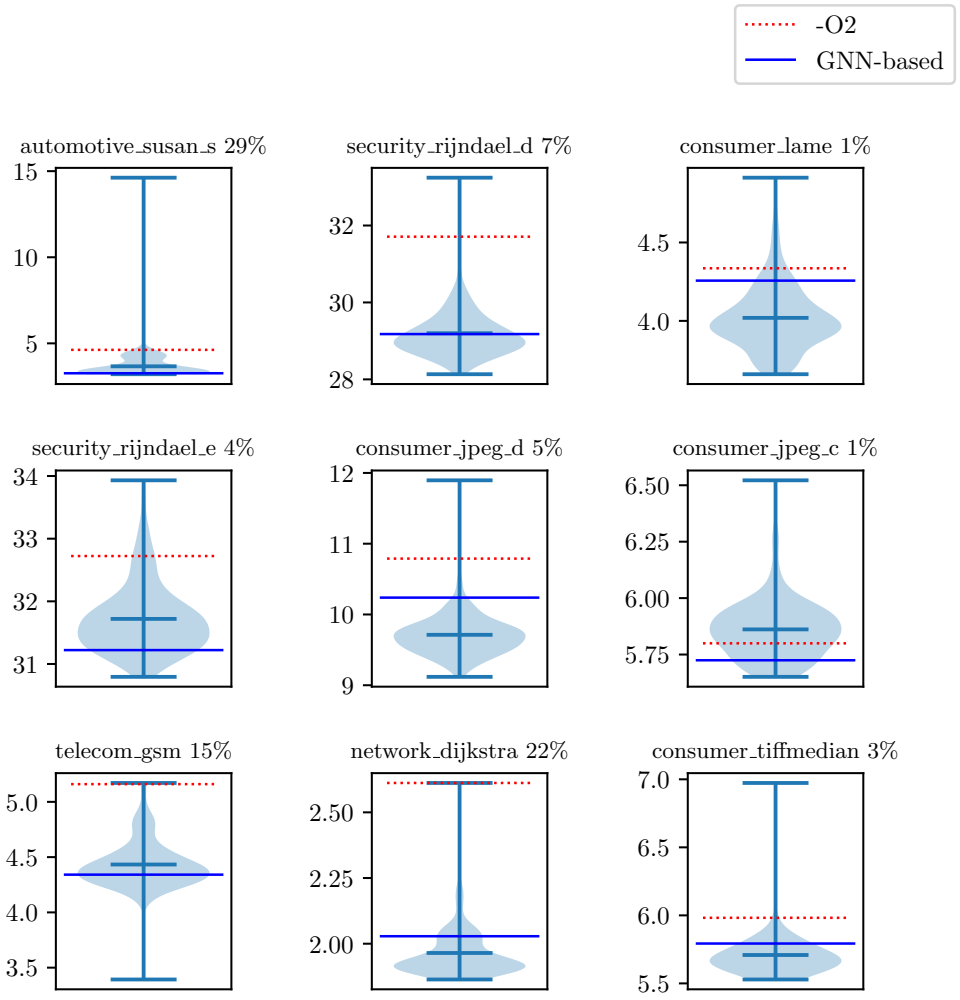

Figure A.14: Cross domain test results of Polybench model on cBench benchmark suite for GNN-based and $-O2$ (continued).

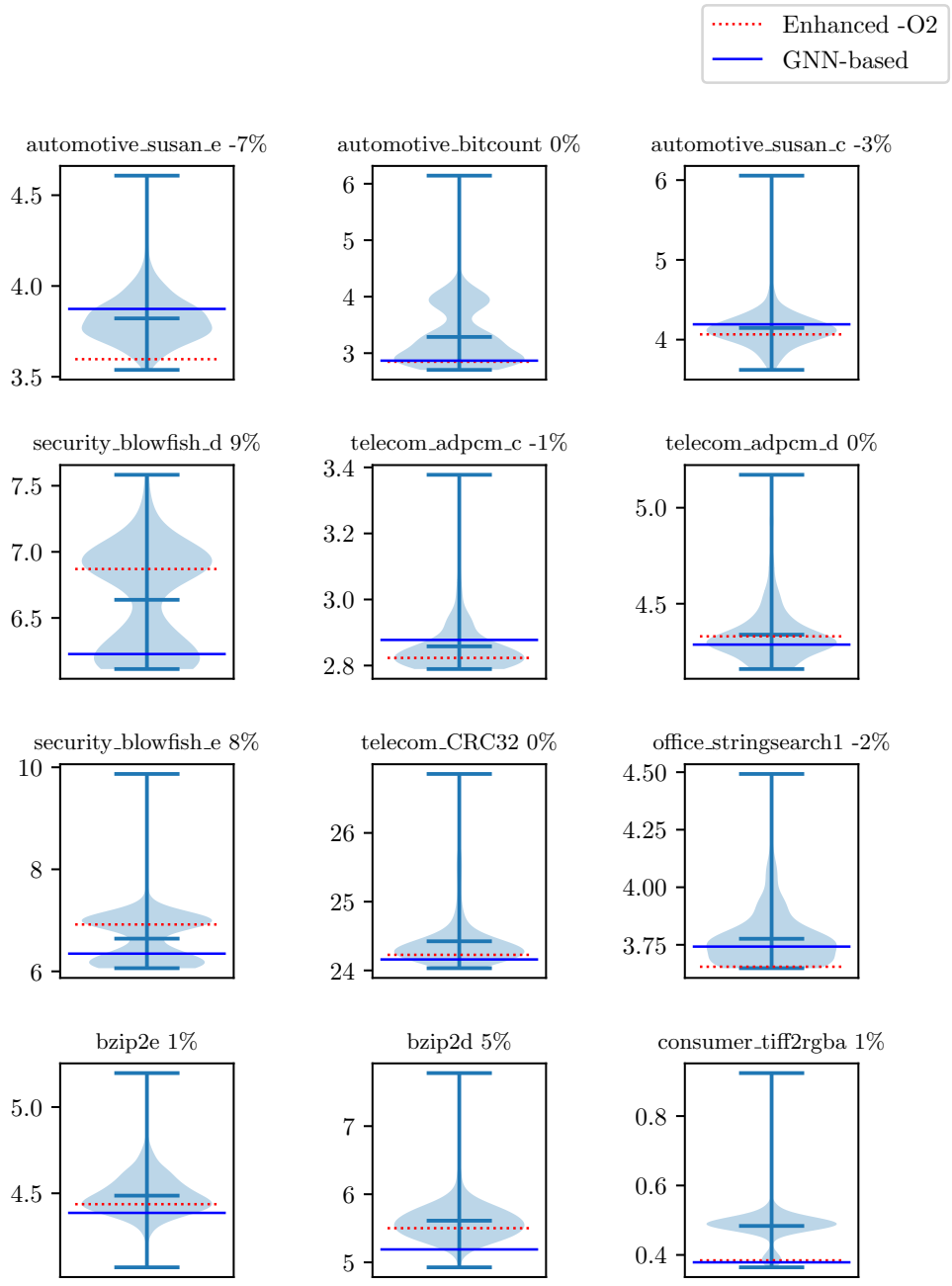

Figure A.15: Cross domain test results of Polybench model on cBench benchmark suite for GNN-based and enhanced $-O2$.

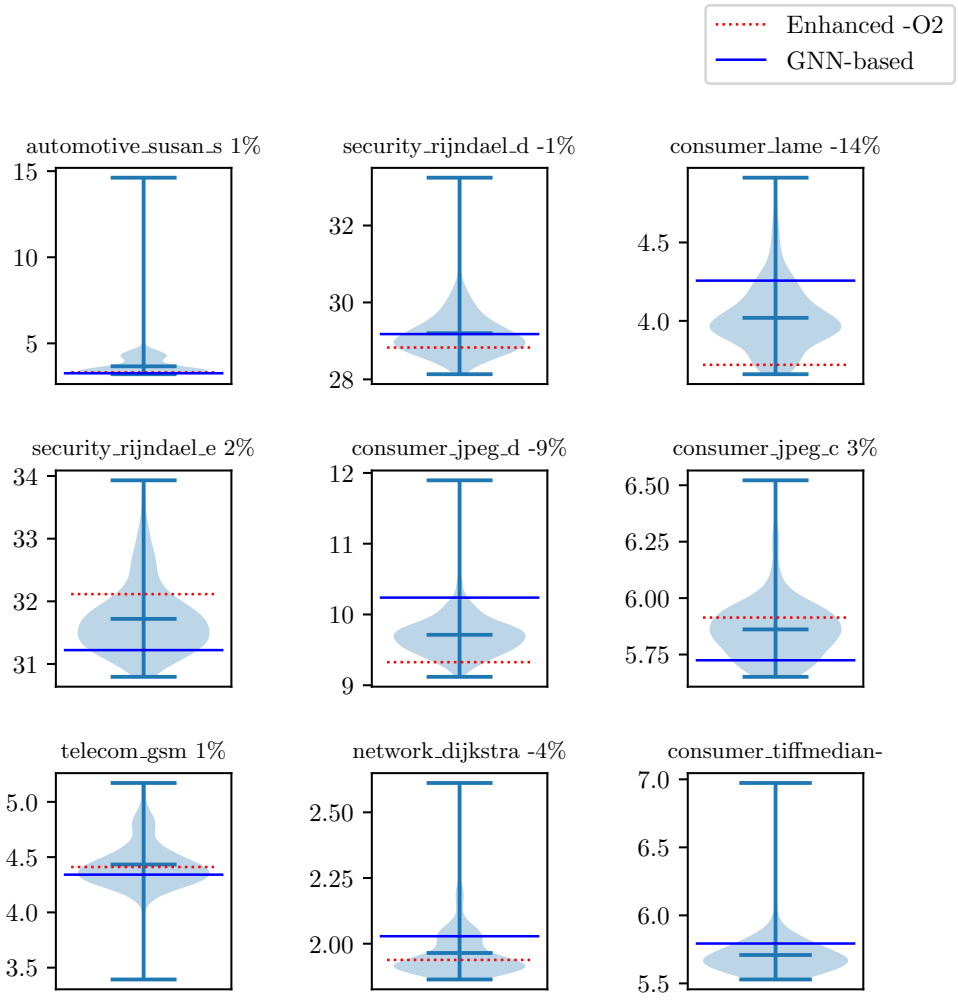

Figure A.16: Cross domain test results of Polybench model on cBench benchmark suite for GNN-based and enhanced $-O2$ (continued).