mLOOP: Optimize Loop Unrolling in Compilation with a ML-based Approach

Zhongchun Zheng
Sun Yat-Sen University
Guangzhou, China
zhengzhch3@mail2.sysu.edu.cn

Yuan Wu
Phytium Technology Co. Ltd
Changsha, China
wuyuan1363@phytium.com.cn

Xianwei Zhang
Sun Yat-Sen University
Guangzhou, China
zhangxw79@mail.sysu.edu.cn

2024 International Conference on Networking, Architecture and Storage (NAS) | 979-8-3315-2050-2/24/$31.00 ©2024 IEEE | DOI: 10.1109/NAS63802.2024.10781373

***Abstract*—Loops are a fundamental component of programs, providing an structured and efficient way to execute repetitive tasks. Given their prevalence and significance, the performance of loops has a direct impact on the overall execution of a program. Predicting loop unroll factor holds remarkable importance in the domain of loop optimization and vectorization parallelism. With the rapid advancements in this field, leveraging machine learning (ML) methods for compilation optimization has emerged as a new research focus. Whereas traditional heuristic algorithms lack precision and Profile-Guided Optimization (PGO) techniques incur considerable compilation overhead, ML method serve as a more balanced approach with respect to accuracy and compilation time. Nonetheless, existing ML approaches are commonly confined to individual optimizations and fail to consider the interplay between multiple optimizations. Additionally, there is inadequate utilization of compilation optimization parameters, resulting in redundant calculations across different optimization processes. This paper proposes `mLOOP`, a method that employs the XGBoost model to predict loop unroll factors which are integrated into the metadata for use throughout the compilation pipeline. To facilitate deployment and testing in practices, `mLOOP` is encapsulated into a LLVM optimization pass. By testing on multiple loop-intensive benchmarks, `mLOOP` achieves 7% speedup on X86 platform and 12% on ARM.**

***Index Terms*—Loop unrolling, Compiler optimization, Machine learning.**

I. Introduction

The demand for performant and efficient program execution is increasingly growing, driving the rapid evolution of both programming languages and hardware processors. As the bridge between language and hardware, compiler transforms source code into machine code by employing a variety of optimization techniques to enhance program performance. With the swift advancements in computer architecture, the reliance on compiler optimization has greatly intensified to fully utilize hardware resources. Exploring compiler optimization techniques remain a focal point of research, emphasizing the critical task of identifying potential optimization opportunities within programs. A key aspect of this process is determining loop unroll factors. Traditional compilers typically employ intricate heuristic algorithms to develop cost functions that evaluate optimization quality based on expertise and fixed rules, considering factors such as loop code size, iteration count, and dependencies. Additionally, traditional compilers construct an abstract architecture model, encapsulating the target computer architecture's features, including the instruction set, memory model, and cache structure. This model guides optimization decisions tailored to the target platform's characteristics and the desired optimization goals, such as selecting suitable instruction sequences, memory access patterns, and register allocation strategies.

However, neither the heuristic algorithm nor the abstract architecture model succeeds to fully unleash the performance potential of loop unrolling. Determining various parameters at compile time, such as loop unroll factors, presents significant challenges. Firstly, the evaluation criteria for the quality of the generated program differ widely. The performance of a program compiled with identical code under different parameters can vary significantly, where the program may run quickly but have a large program size, or may be compact but run slowly. In cases with ample memory, the focus is solely on execution speed, whereas limited memory necessitates partially sacrificing execution performance to ensure proper program function. Consequently, the evaluation of program quality is environment-dependent, complicating the assessment of the generated program. Secondly, accurately measuring a program's execution performance is challenging. During execution, factors such as kernel scheduling, cache behavior, and system noise can cause significant fluctuations in runtime, making it difficult to measure performance under consistent conditions. Thirdly, there is often no direct causal relationship between code performance and compilation parameters. Compilation parameters indirectly affect runtime outcomes, complicating the selection and adjustment of corresponding parameters. A complete compilation optimization process may involve tens of thousands of parameters, making manual adjustment or rule-based design both difficult and time-consuming.

To tackle the aforementioned challenges, PGO (Profile-Guided Optimization) [1] technology has been developed to adjust compilation optimization parameters using a profile file generated during program execution. The profile file contains measurements of various indicators reflecting the actual running efficiency of the program. By fine-tuning compilation parameters and recompiling based on the profile file, the program can achieve better performance. However, implementing PGO is complex, requiring the program to be compiled and run once before optimization can occur, thereby significantly lengthening compilation time.

979-8-3315-2050-2/24/$31.00 ©2024 IEEE
Authorized licensed use limited to: SUN YAT-SEN UNIVERSITY. Downloaded on October 10,2025 at 14:44:54 UTC from IEEE Xplore. Restrictions apply.

Currently, compiler optimizers are structured into pipeline stages, with each being called as an optimization pass. These passes act on the intermediate representation (IR) of the code to optimize. Optimization passes adopt various heuristic algorithms to make decisions on compilation parameters. For example, when determining the loop unroll factor or assessing branch probabilities, the compiler uses these heuristic algorithms to balance program size, runtime, and compilation time. In LLVM [2], optimization is decoupled into separate passes, which can be turned on or off based on compilation parameters and program characteristics. Practitioners can easily add new passes to the workflow to deploy optimization methods for specific steps in LLVM.

To balance performance and efficiency, machine learning (ML) methods are employed to adjust the loop unroll factor in compilation optimization. By predicting optimization parameters through offline training models, it is possible to reduce compilation time while achieving better operational results. The trained ML model can be encapsulated into an optimization pass, and further integrated into LLVM for facilitating easy deployment and application. Following the idea, this paper proposes `mLOOP` to explore the use of ML methods to predict loop unroll factors, which are written into metadata for use throughout the compilation process.

To collect features and labels required for ML training, we integrate this functionality into the optimization pass in LLVM. This optimization pass gathers loop-related features and labels from each loop and basic blocks within the loop, including loop unroll factor generated during LLVM optimization and the estimated number of loop runs. Eighteen loop and basic block features relevant to loop unrolling performance are selected as training and prediction features. These include loop depth, statistics on the number of various instructions in the loop, and other features. These labels and features are integrated through interfaces implemented in LLVM. After deploying this optimization pass, the collected information from over 16,000 loops and estimates of loop unroll factors for training ML models is automatically saved to a preset file.

The XGBoost [3] library function is then used to train a decision tree model for predicting loop unroll factors. These features serve as intermediate nodes of the decision tree, with the labels being leaf nodes. After training, the model is exported and encapsulated in an optimization pass. This optimization pass, applied to all loops, collects and integrates loop features, using the decision tree to predict the loop unroll factor based on these features. The predicted factor is then added to the loop metadata. In subsequent loop unrolling optimizations, the predicted factor is used, instead of the heuristic algorithm's estimated factor, to determine the number of unrolling steps.

The contributions of this paper are summarized as follow:

- Aiming to facilitate compiler optimizations with ML, we adopt XGBoost to predict loop unroll factors, supplementing the conventional heuristic-based and PGO methods.

```
for(int i = 0;i<n;i+=1){
    ...
    func(i);
    ...
}
```

(a) loop without unrolling

```
for(int i = 0;i<n;i+=2){
    ...
    func(i);
    ...
    ...
    func(i+1);
    ...
}
```

(b) loop with unrolling by a unroll factor of 2

Fig. 1. The example of loop unrolling

- An LLVM optimization pass is implemented for collecting features and labels, specifically selecting eighteen features pertinent to loop unrolling. This pass can be seamlessly deployed within the LLVM framework to gather training data for machine learning models.
- A substantial dataset of loop unroll factors is compiled, which has proven effective in enhancing program performance. This dataset provides a valuable resource for further research in future studies.

II. Background and Motivation

A. Background

Loops are fundamental control flow mechanisms in programming, enabling the repeated execution of a code block until a specified condition is satisfied. This repetitive execution facilitates efficient processing of large datasets, the consistent performance of specific actions, and the generation of repetitive patterns within a program. Loop unrolling is a prominent loop optimization technique that enhances program performance by replicating the loop body multiple times within each iteration (Figure 1). This replication is guided by a loop unroll factor, which effectively reduces the total number of loop iterations while modifying the loop control logic. The benefits of loop unrolling are multifaceted: it minimizes loop control overhead, mitigates branch prediction failures, promotes memory locality, and facilitates instruction parallelism. However, loop unrolling can also negatively impact performance by increasing code size, potentially reducing instruction cache hit rates and exacerbating register pressure. Consequently, determining an optimal loop unroll factor is crucial to maximize the performance gains of loop unrolling while minimizing the detrimental effects of code size expansion.

B. Motivation

The loop unroll factors derived from heuristic methods are often suboptimal. In some scenarios, adjusting the loop unroll factor can enhance performance, while in others, reducing the unroll factor can decrease program size without compromising performance. This distinction arises because, during execution, only a subset of loops, known as hot loops, is frequently executed. By assigning an appropriate unroll factor to hot loops and minimizing the factor for cold loops, one can achieve both a reduction in program size and an improvement in performance. Randomly adjusting the loop unrolling factor

Authorized licensed use limited to: SUN YAT-SEN UNIVERSITY. Downloaded on October 10,2025 at 14:44:54 UTC from IEEE Xplore. Restrictions apply.

can result in a performance enhancement of 3% to 30% in certain test cases. Heuristic methods generally rely solely on static program information, whereas Profile-Guided Optimization (PGO) provides dynamic execution data. Utilizing PGO, the execution frequency of loops can be more accurately estimated, allowing for the prediction of more suitable unroll factors. However, PGO's effectiveness is contingent on the program's input, and its implementation is time-consuming, requiring the program to be compiled twice and executed once. Moreover, there is a deficiency in the effective use of PGO data, rendering dynamic information challenging to leverage. In contrast, machine learning techniques can swiftly determine superior loop unroll factors compared to heuristic methods, offering a more efficient alternative.

III. Design

A. Overview

The whole process of `mLOOP` is divided into offline phase and online phase as shown in Figure 2.

Offline Phase: In this phase, the random forest model named XGBoost is selected for predicting loop unroll factors. First, the features of all loops in the training set are collected and the loop unroll factors of each loop are collected as labels. In this phase, a new optimization pass is added to the optimizer and enabled through compilation parameters. First, all programs in the training set are compiled with this optimization pass enabled. This optimization pass collects the relevant features of loop unrolling, denoted as X. At the same time, because the PGO technology used in LLVM does not generate loop unroll factor technology, the heuristic algorithm of LLVM is used to calculate the loop unroll factor as a label. This optimization pass records the loop unroll factor provided by the LLVM optimizer, denoted as Y. The features used for subsequent predictions are the same as those collected above. The information collected above will be saved to a large file on disk. A Python script will then be used to call the XGBoost library to train this data. Generate a random forest that can predict loop unroll factors based on the features of the loop. And this model will be generated as C code and then integrated into an optimization pass.

Online Phase: In the online phase, another optimization pass is introduced. When enabled, this pass automatically predicts the loop unroll factor for each loop during compilation and writes it into the loop's metadata. This prediction is loop-specific, providing a finer granularity compared to setting a universal loop unroll factor in the compilation parameters. The new optimization pass also collects loop features and uses the model trained in the offline phase to predict the loop expansion factor.

B. Feature Collection

The accuracy of a model's predictions is largely determined by the selected feature set, which necessitates the collection of extensive data for training. In this study, features and loop expansion factors for over 16,000 loops across various programs were gathered for training purposes. Eighteen distinct features were selected, including the number of instructions within each loop, the variety of instruction types (such as return and exception handling instructions), loop depth, and the number of basic blocks. These features were stored in arrays as floating-point numbers. The primary data source for the training set was the llvm-test-suite, with features collected by compiling and executing these programs. The feature collection process was encapsulated as an optimization pass, enabling easy deployment within LLVM for feature collection. By executing a script to add the optimization pass to the LLVM optimizer and specifying the necessary compilation parameters, the optimization pass operates automatically, storing the collected features in a predefined file.

TABLE I
Selected Features

num_instr	num_float_ops
num_phis	num_branches
num_calls	num_operands
num_preds	num_memory_ops
num_succ	num_unique_predicates
ends_with_unreachable	trip_count
ends_with_return	num_uses
ends_with_cond_branch	num_blocks_in_lp
ends_with_branch	loop_depth

In addition to the features directly related to the loop, contextual features can be incorporated to differentiate identical library functions called from different files. In such scenarios, the intrinsic features of the library function remain constant and cannot distinguish between calls from various files. Including contextual information effectively addresses this issue. Additionally, hardware features can be integrated to enhance the model's scalability across different architectures. Runtime-related features, such as the number of loop executions, can also be included. Incorporating dynamic features allows this method to be combined with Profile-Guided Optimization (PGO) technology, potentially yielding better performance. The complete list of all eighteen features is presented in Table I.

C. Label Collection

The XGBoost algorithm employed in this study is a supervised learning algorithm, necessitating labeled data for training, specifically the loop unroll factor as a training label. Several methods exist to obtain this factor. One approach is to extract compilation information from the Profile file using Profile-Guided Optimization (PGO) technology. Alternatively, different loop unroll factors can be set, and the program's runtime under these varying factors can be tested to determine the optimal unroll factor. For instance, setting loop unroll factors from 1 to 10 for the same program and measuring their respective runtimes can help identify the best performing factor as the label. However, this method is coarse-grained, as LLVM typically sets a uniform loop expansion factor for the entire compilation stage, which is suboptimal because only hot loops require unrolling, while cold loops do not. Unrolling hot loops can enhance execution speed, whereas unrolling cold

Authorized licensed use limited to: SUN YAT-SEN UNIVERSITY. Downloaded on October 10,2025 at 14:44:54 UTC from IEEE Xplore. Restrictions apply.

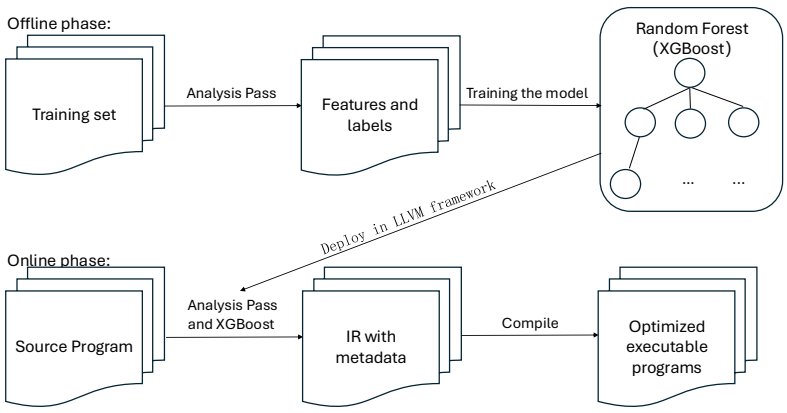


Fig. 2. Overall workflow of mLOOP.

loops may increase program size and reduce performance. Consequently, predicting the loop unroll factor for each loop individually is expected to yield better results than using a unified factor.

In this study, LLVM's built-in heuristic algorithm is employed to predict the loop unroll factor, which serves as the initial training label. This label is subsequently refined by exploring values within its vicinity. During the feature and label collection optimization pass, the heuristic function predicts the loop unroll factor for each loop. We then adjust this unroll factor to 0.6x, 0.8x, 1.2x, 1.4x, and 1.8x its original value, and also round it up and down to the nearest power of 2. By evaluating the performance of these adjusted unroll factors, we select the best performing ones as the labels for model training.

This pseudocode outlines a method for gathering loop features and unroll factors, executed as a function pass. This approach is convenient to implement and effectively captures the contextual information of loops. The process involves iterating through all loops within the function to collect predefined features and labels. Subsequently, the acquired data is stored in a file.

Algorithm 1 Feature and label collection Pass

Input: Function Pointer F
Output: All loop features and labels within the function

```
for Loop in F do
    Collect loop unroll factors generated by the O3 heuristic
    Collecting Loop features
    for BasicBlock in Loop do
        Collect BasicBlock features
    end for
    Integrate Loop features and BasicBlock features
end for
return Features ans labels
```

D. Model Training

In the initial phase, over 16,000 loop features and corresponding loop unroll factors were collected and stored in a file, with each line representing the features and unroll factor of a specific loop. The XGBoost library in Python was used to train a model on this dataset.

A training set with n examples and m features can be represented as:

$$\mathbf{D} = \{< x_i, y_i > \mid \ |D| = n, x_i \in \mathbb{R}^m, y_i \in \mathbb{R}\}$$

The XGBoost model, which is a decision forest, integrates K activation functions to predict labels based on features. The prediction method is illustrated by the following formula:

$$y_i = \sum_{k=1}^{K} f_k(x_i)$$

Here, each $f_k$represents a decision tree that predicts the loop unroll factor based on the m-dimensional features, effectively mapping these features to a one-dimensional label. During training, the dataset was divided into a 1:9 ratio, with 10% used for testing and 90% for training. The model achieved a prediction accuracy of 75% on the test set, and the entire training process was completed in less than one minute.

E. Code Export

Post-training, the model, initially stored in Python, must be converted to C/C++ for integration into the LLVM optimization pass, which is implemented in C++. To facilitate this, the model is exported as a C/C++ function, enabling its direct use in the optimization pass.

XGBoost, comprising multiple decision trees, provides the model in JSON format. A script can convert these JSON model parameters into C/C++ code, creating corresponding decision trees and synthesizing their results. This script, sourced from the repository https://github.com/nadavrot/pgo_ml, automates the conversion process. Once converted, the model can be

Authorized licensed use limited to: SUN YAT-SEN UNIVERSITY. Downloaded on October 10,2025 at 14:44:54 UTC from IEEE Xplore. Restrictions apply.

added to the LLVM project and compiled, allowing direct reference and usage in subsequent optimization passes.

F. Prediction Process

The prediction of loop unroll factors is conducted via an optimization pass, applied to all loops within a function each time it is called. The LLVM optimizer applies this pass to all functions in the program. During the optimization pass, each loop is traversed to collect features. These features, including those of basic blocks and loop depth, are compiled into a floating-point array, serving as input to the decision tree model to predict the loop unroll factor. This factor is then added to the loop's metadata.

Since LLVM does not allow direct modification of loop metadata, new metadata is created to overwrite the existing one. This involves copying existing metadata, adding or modifying the loop unroll factor, and using the relevant interface to apply the new metadata. In subsequent compilation phases, if no specific compilation parameters are set, LLVM prioritizes using this loop metadata for loop unrolling calculations.

To ensure the predicted loop metadata is utilized during optimization, the optimization pass must precede the loop unrolling and vectorization passes. Additionally, it relies on information generated by other optimization passes, necessitating careful manual placement within the LLVM optimization pass management file. In this study, the optimization pass for predicting the loop unroll factor was placed adjacent to the loop unrolling optimization pass.

This pseudocode delineates the procedure for predicting the loop unroll factor, implemented as a loop optimization pass. It iterates through each loop within the function, collects the loop's features, and inputs the feature vector into the XGBoost model. The model predicts and returns the loop unroll factor, which is then assigned as loop metadata for utilization in the subsequent unrolling phase.

Algorithm 2 Predicting loop unroll factors Pass

Input: Function Pointer F and Trained XGBoost model
Output: Function with loop unroll factor metadata

```
for Loop in F do
    Collecting Loop features
    for BasicBlock in Loop do
        Collect BasicBlock features
    end for
    Integrate Loop features and BasicBlock features
    Call the XGBoost model and use the collected features
    to predict the loop unroll factor
    Write the predicted loop unroll factor into the metadata
end for
return Function with loop unroll factor metadata
```

IV. Evaluation

A. Methodology

1) System Configurations: The experimental evaluation was conducted on two distinct computational architectures: the X86 and the Arm platforms. A detailed enumeration of the device parameters utilized in the study is delineated in Table II. The experimental procedures were encapsulated within a Docker environment to ensure consistency and reproducibility. The implementation of the experiment necessitated proficiency in C++ and Python programming languages. For the Python-based components, the installation of libraries including NumPy, scikit-learn, and XGBoost was mandatory. Furthermore, the configuration of the LLVM project was an essential prerequisite for the experiment.

TABLE II
Hardware and Software Environment.

Architecture	Field	Value
X86	CPU	Intel(R) Xeon(R) Gold 6150 CPU @2.7GHz
	OS	Debian GNU/Linux 11(bullseye)
Arm	CPU	Kunpeng-920
	OS	Ubuntu 22.04.4 LTS

2) Workloads: To comprehensively assess the impact of the model, the LLVM-TEST-SUITE benchmark suite was employed, encompassing two benchmarks characterized by their loop-intensive nature: PolyBench and TSVC. These benchmarks were chosen to scrutinize the model's performance under conditions of high computational demand within loop structures. Furthermore, to examine the model's efficacy in the context of broader large-scale applications, the SPEC2017 benchmark suite was also incorporated into the testing regimen. A succinct overview of these benchmarks is presented in Table III.

TABLE III
Benchmark.

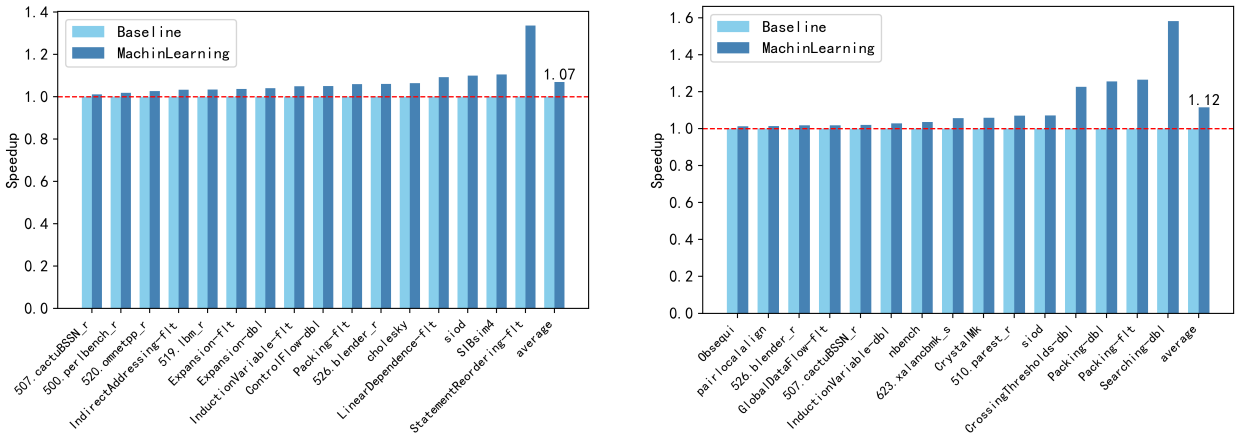
Benchmark	Note
LLVM-TEST-SUITE	This benchmark is used to test the functional correctness of LLVM. It contains several small loop-intensive benchmarks.
SPEC 2017	The SPEC CPU® 2017 benchmark package contains industry-standardized, CPU intensive suites for measuring and comparing compute intensive performance, stressing a system's processor, memory subsystem and compiler.

3) Metrics: In this study, three primary metrics are utilized to assess the efficacy of the optimization: code compilation time, runtime performance of the executable file, and the resultant file size. The emphasis is placed on the runtime and file size of the executable, as these parameters are indicative of the program's operational efficacy within a practical environment. Concurrently, it is imperative to ensure that the compilation time remains within an acceptable threshold.

B. Execution Time

Figure 3 illustrates the performance enhancement achieved by employing machine learning models to predict loop unroll factors across two distinct architectures: X86 and Arm. On the X86 platform, the models yielded an average speedup

Authorized licensed use limited to: SUN YAT-SEN UNIVERSITY. Downloaded on October 10,2025 at 14:44:54 UTC from IEEE Xplore. Restrictions apply.



(a) X86 (b) ARM

Fig. 3. Speedup of machine learning methods relative to LLVM O3

of 7% relative to the LLVM O3 optimization level. In contrast, the Arm architecture demonstrated a more pronounced improvement, with an average speedup of 12%. Notably, for the StatementReordering test case on the X86 architecture, a peak speedup of 33.6% was recorded. Similarly, on the Arm architecture, the Searching test case exhibited an exceptional peak speedup of 58.2%. These findings underscore the potential of machine learning-assisted optimization in enhancing computational efficiency, particularly in the context of loop unrolling optimizations.

The Arm architecture demonstrates a higher average speedup (12%) compared to the x86 architecture (7%), suggesting that the ML models may be more effective on the Arm platform for loop unrolling optimizations.

C. Code Size

TABLE IV
ON X86, THE EXPANSION OF PROGRAM SIZE

test case	baseline	optimization	expansion
500.perlbench_r	2486920	2605704	4%
507.cactuBSSN_r	5458928	6499312	19%
519.lbm_r	30472	30472	0%
520.omnetpp_r	2909768	2983496	3%
526.blender_r	23062920	23562632	2%
IndirectAddressing-flt	68453	46245	-32%
Expansion-flt	70917	48933	-31%
Expansion-dbl	48309	49349	2%
InductionVariable-dbl	47045	47989	2%
ControlFlow-dbl	52837	54085	2%
Packing-flt	67013	44853	-33%
siod	127061	130149	2%
cholesky	3733	4197	12%
LinearDependence-flt	72005	50421	-30%
SIBsim4	46405	52581	13%
StatementReordering-flt	66869	44677	-33%
total	34619655	36255095	5%

The data in Tables IV and V delineate the variations in code size for the x86 and Arm architectures, respectively, following optimization. For the x86 platform, the overall code size increased by approximately 4.7%, while the Arm architecture exhibited a more substantial augmentation of 7.1%.

TABLE V
ON ARM, THE EXPANSION OF PROGRAM SIZE

test case	baseline	optimization	expansion
507.cactuBSSN_r	5458928	6499312	19%
510.parest_r	13948408	15513080	11%
526.blender_r	23062920	23562632	2%
623.xalancbmk_s	7334960	7736368	5%
Obsequi	32860	34436	5%
pairlocalalign	355388	512244	44%
GlobalDataFlow-flt	39724	38076	-4%
InductionVariable-dbl	51564	37340	-28%
nbench	40268	44132	10%
CrystalMk	4332	4772	10%
siod	135692	141220	4%
CrossingThresholds-dbl	50780	36716	-28%
Packing-dbl	48772	34500	-29%
Packing-flt	36148	34572	-4%
Searching-dbl	48268	33964	-30%
total	50649012	54263364	7%

Table IV provides detailed insights into the code size changes for various test cases on the x86 architecture. The total code size expanded from 34,619,655 bytes to 36,255,095 bytes post-optimization. Notably, most test cases experienced an increase in size, with 507.cactuBSSN_r showing a significant rise from 5,458,928 bytes to 6,499,312 bytes. However, a few test cases, such as IndirectAddressing-flt and Packing-flt, demonstrated a reduction in code size.

Similarly, Table V illustrates the code size changes for different test cases on the Arm architecture. The total code size grew from 50,649,012 bytes to 54,263,364 bytes after optimization. Significant increases were observed in test cases such as 510.parest_r and pairlocalalign, with the latter increasing from 355,388 bytes to 512,244 bytes. Conversely, some test cases, like GlobalDataFlow-flt and InductionVariable-dbl, experienced a decrease in size.

The comparative analysis reveals that the Arm architecture experienced a greater overall increase in code size compared to the x86 architecture. This disparity could be attributed

Authorized licensed use limited to: SUN YAT-SEN UNIVERSITY. Downloaded on October 10,2025 at 14:44:54 UTC from IEEE Xplore. Restrictions apply.

to differences in optimization techniques and architectural characteristics. While both architectures exhibited an overall increase in code size, the magnitude and distribution of these changes varied across different test cases.

D. Compilation Time

Regarding compilation time, we consider two primary components. The first is the time taken by the machine learning model to predict the loop unroll factor, and the second is the increased compilation time resulting from the expanded code volume after adjusting the loop unroll factor. The time consumed by the machine learning model is negligible compared to the overall compilation process, as the depth of each decision tree in the random forest is shallow, resulting in minimal time consumption. The increase in compilation time due to code volume expansion is generally acceptable in most test cases. However, in a few instances, the compilation time may increase significantly, sometimes by an order of magnitude, due to the substantial code volume. Therefore, in these specific test cases, it is necessary to limit the loop unroll factor.

E. Generate IR for Comparison

This section demonstrates how the loop unroll factor is stored in metadata after being predicted. As illustrated in the following code, this is an intermediate representation generated in an experiment. The identifier comprising an exclamation mark and a number is the metadata created during the compilation process. Specifically, llvm.loop.unroll.count is the metadata used for loop unrolling, added by the optimization pass implemented in this study. In the given code, only llvm.loop.unrolltest.count is present, and its value matches llvm.loop.unroll.count. This is because the original metadata is utilized and subsequently deleted during the loop unrolling process. For clarity, We added llvm.loop.unrolltest.count metadata to preserve relevant information. It can be observed that in this function, the loop unroll factor predicted by the machine learning model is 5, and the metadata llvm.loop.unroll.disable indicates that the loop unrolling optimization has been completed, implying that future loop unrolling is unnecessary. In the actual intermediate code representation, the loop-related basic blocks are copied multiple times.

```
define i32 @foo() {
  ...
}
!7 = distinct !{!7, !8, !9, !10}
!8 = !{!"llvm.loop.mustprogress"}
!9 = !{!"llvm.loop.unrolltest.count", i32 5}
!10 = !{!"llvm.loop.unroll.disable"}
```

V. Related Work

The application of machine learning to compiler optimization has been extensively studied, demonstrating its potential to surpass human-designed heuristics, as evidenced by the works of Ashouri et al. [4] and Leather et al. [5].

Supervised learning is the most commonly employed technique in this domain. It has been successfully applied to various problems, such as loop unrolling [6], [7], instruction scheduling [8], program segmentation [9], heterogeneous device mapping [10], [11], function inlining [12], and diverse optimization heuristics in GCC [13].

Supervised learning relies on labeled data for training, which is challenging to obtain from compilers. In specific areas like code generation for linear algebra primitives, a fixed compilation pipeline and program for measurement can be used [14]. However, measurement noise complicates the evaluation of program performance, particularly for small metrics such as the runtime of a basic block [15]. To address this, Steiner et al. [16] used noise in measurements as a prediction target. Other projects utilize static indicators, such as static analysis of generated binary code or code size [17], [18].

Feature design is a critical challenge in supervised learning for compiler optimization. Identifying which indicators impact compilation results is difficult, and the quality of a supervised model is closely tied to its feature set. Previous studies often use manually selected numerical feature vectors for model training [19]–[21]. However, manual feature selection is time-consuming and prone to errors. Leather et al. [22] proposed an automatic feature selection method, though it requires complex syntax to describe the feature space. Recent research has introduced deep learning models inspired by natural language processing [7], [10], [23] and graph learning [11], [24] to simplify feature selection by automatically inferring high-level features from low-level input representations. Despite their advantages, these techniques reduce interpretability, making it difficult to understand the relationships between features. Our approach balances interpretability and efficiency by utilizing numerous values readily available from LLVM's static analysis and machine learning algorithms.

Reinforcement learning is another approach to compiler optimization. Some research has framed compiler optimization problems as reinforcement learning environments [18], while others have applied reinforcement learning to vectorization decisions in the LLVM vectorizer [25], [26]. The MLGO project integrates neural networks into LLVM for function inlining heuristics [17]. ESP is an earlier work applying machine learning to branch prediction [27]. These methods share similarities with our work; however, our approach uniquely writes predicted results into metadata, allowing a single prediction to influence the entire compilation process.

This study employs XGBoost to predict loop unroll factors, whereas related work has used the XGBoost algorithm for predicting branch weights [28].

VI. Conclusion and Future Work

This paper employs a ML model to predict the loop unroll factor for each loop at a fine-grained level, achieving a performance comparable to the O3 level. The ML model is integrated into the LLVM framework, facilitating deployment in practices. Furthermore, this study collects various loop unroll factors as labels for the model, which effectively enhance program performance for future research. The feasibility of

Authorized licensed use limited to: SUN YAT-SEN UNIVERSITY. Downloaded on October 10,2025 at 14:44:54 UTC from IEEE Xplore. Restrictions apply.

using ML models to predict loop unroll factors is validated, demonstrating simultaneous improvements in performance and reductions in program size.

However, there are some limitations in this work. The feature selection for the ML model is neither sufficient nor entirely appropriate. Certain test cases cannot be differentiated with the current features, leading to sub-optimal performance in some instances. Additionally, the current features lack context and hardware-related information. Incorporating context information in the future could help distinguish library function calls in different programs, as the same library function may be invoked at varying frequencies across different programs, leading to its classification as a hot function in one context and a cold function in another. Consequently, the optimal loop unroll factor for the same function may differ based on context. Including hardware information in the features can enhance the model's scalability and robustness, ensuring efficient performance across different platforms.

The current model does not account for the program's dynamic information, relying solely on static information to predict the loop unroll factor. Additionally, the quality of the collected loop unroll factors, which serve as labels, is inadequate. The performance of the ML model is contingent on the quality of its training data, and the collected labels currently do not fully explore the optimization space of loop unrolling, resulting in sub-optimal model performance. Presently, the loop unroll factors obtained through O3 optimization using heuristic methods are not optimal. Future research should focus on collecting better unroll factors and improving feature engineering to fully explore this optimization method's potential.

ACKNOWLEDGEMENT

We thank the anonymous reviewers for their constructive comments and suggestions. This research was supported by the CCF-Phytium Fund #202204. Xianwei Zhang is the corresponding author.

REFERENCES

[1] Gupta, R., Mehofer, E. & Zhang, Y. Profile-Guided Compiler Optimizations. *The Compiler Design Handbook: Optimizations And Machine Code Generation*. pp. 143-174 (2002), https://doi.org/10.1201/9781420040579.ch4

[2] Lattner, C. & Adve, V. LLVM: A compilation framework for lifelong program analysis & transformation. *International Symposium On Code Generation And Optimization, 2004. CGO 2004.*. pp. 75-86 (2004)

[3] Chen, T. & Guestrin, C. XGBoost: A Scalable Tree Boosting System. *Proceedings Of The 22nd ACM SIGKDD International Conference On Knowledge Discovery And Data Mining*. pp. 785-794 (2016), https://doi.org/10.1145/2939672.2939785

[4] Ashouri, A., Killian, W., Cavazos, J., Palermo, G. & Silvano, C. A survey on compiler autotuning using machine learning. *ACM Computing Surveys (CSUR)*. **51**, 1-42 (2018)

[5] Leather, H. & Cummins, C. Machine learning in compilers: Past, present and future. *2020 Forum For Specification And Design Languages (FDL)*. pp. 1-8 (2020)

[6] Stephenson, M. & Amarasinghe, S. Predicting unroll factors using supervised classification. *International Symposium On Code Generation And Optimization*. pp. 123-134 (2005)

[7] Cummins, C., Petoumenos, P., Wang, Z. & Leather, H. End-to-end deep learning of optimization heuristics. *2017 26th International Conference On Parallel Architectures And Compilation Techniques (PACT)*. pp. 219-232 (2017)

[8] Moss, J., Utgoff, P., Cavazos, J., Precup, D., Stefanovic, D., Brodley, C. & Scheeff, D. Learning to schedule straight-line code. *Advances In Neural Information Processing Systems*. **10** (1997)

[9] Wang, Z. & O'Boyle, M. Partitioning streaming parallelism for multi-cores: a machine learning based approach. *Proceedings Of The 19th International Conference On Parallel Architectures And Compilation Techniques*. pp. 307-318 (2010)

[10] Ben-Nun, T., Jakobovits, A. & Hoefler, T. Neural code comprehension: A learnable representation of code semantics. *Advances In Neural Information Processing Systems*. **31** (2018)

[11] Cummins, C., Fisches, Z., Ben-Nun, T., Hoefler, T., O'Boyle, M. & Leather, H. Programl: A graph-based program representation for data flow analysis and compiler optimizations. *International Conference On Machine Learning*. pp. 2244-2253 (2021)

[12] Cavazos, J. & O'boyle, M. Method-specific dynamic compilation using logistic regression. *ACM SIGPLAN Notices*. **41**, 229-240 (2006)

[13] Fursin, G., Miranda, C., Temam, O., Namolaru, M., Zaks, A., Mendelson, B., Bonilla, E., Thomson, J., Leather, H., Williams, C. & Others MILEPOST GCC: machine learning based research compiler. *GCC Summit*. (2008)

[14] Zheng, L., Jia, C., Sun, M., Wu, Z., Yu, C., Haj-Ali, A., Wang, Y., Yang, J., Zhuo, D., Sen, K. & Others Ansor: Generating high-performance tensor programs for deep learning. *Proceedings Of The 14th USENIX Conference On Operating Systems Design And Implementation*. pp. 863-879 (2020)

[15] Mendis, C., Renda, A., Amarasinghe, S. & Carbin, M. Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks. *International Conference On Machine Learning*. pp. 4505-4515 (2019)

[16] Steiner, B., Cummins, C., He, H. & Leather, H. Value learning for throughput optimization of deep learning workloads. *Proceedings Of Machine Learning And Systems*. **3** pp. 323-334 (2021)

[17] Trofin, M., Qian, Y., Brevdo, E., Lin, Z., Choromanski, K. & Li, D. Mlgo: a machine learning guided compiler optimizations framework. *ArXiv Preprint ArXiv:2101.04808*. (2021)

[18] Cummins, C., Wasti, B., Guo, J., Cui, B., Ansel, J., Gomez, S., Jain, S., Liu, J., Teytaud, O., Steiner, B. & Others Compilergym: Robust, performant compiler optimization environments for ai research. *2022 IEEE/ACM International Symposium On Code Generation And Optimization (CGO)*. pp. 92-105 (2022)

[19] Kanev, S., Darago, J., Hazelwood, K., Ranganathan, P., Moseley, T., Wei, G. & Brooks, D. Profiling a warehouse-scale computer. *Proceedings Of The 42nd Annual International Symposium On Computer Architecture*. pp. 158-169 (2015)

[20] Trofin, M., Qian, Y., Brevdo, E., Lin, Z., Choromanski, K. & MLGO, D. a Machine Learning Guided Compiler Optimizations Framework. *ArXiv*. **2101** (2021)

[21] Jordan, M., Kearns, M. & Solla, S. Proceedings of the 1997 conference on Advances in neural information processing systems 10. (MIT Press,1998)

[22] Leather, H., Bonilla, E. & O'boyle, M. Automatic feature generation for machine learning–based optimising compilation. *ACM Transactions On Architecture And Code Optimization (TACO)*. **11**, 1-32 (2014)

[23] Kanade, A., Maniatis, P., Balakrishnan, G. & Shi, K. Learning and evaluating contextual embedding of source code. *International Conference On Machine Learning*. pp. 5110-5121 (2020)

[24] Brauckmann, A., Goens, A., Ertel, S. & Castrillon, J. Compiler-based graph representations for deep learning models of code. *Proceedings Of The 29th International Conference On Compiler Construction*. pp. 201-211 (2020)

[25] Haj-Ali, A., Ahmed, N., Willke, T., Shao, Y., Asanovic, K. & Stoica, I. Neurovectorizer: End-to-end vectorization with deep reinforcement learning. *Proceedings Of The 18th ACM/IEEE International Symposium On Code Generation And Optimization*. pp. 242-255 (2020)

[26] Rotem, N. & Schwaighofer, A. Vectorization in LLVM https://llvm.org/devmtg/2013-11/slides/Rotem-Vectorization. pdf. *LLVM Developer's Meeting*. (2013)

[27] Calder, B., Grunwald, D., Jones, M., Lindsay, D., Martin, J., Mozer, M. & Zorn, B. Evidence-based static branch prediction using machine learning. *ACM Transactions On Programming Languages And Systems (TOPLAS)*. **19**, 188-222 (1997)

[28] Rotem, N. & Cummins, C. Profile Guided Optimization without Profiles: A Machine Learning Approach. (2022)

Authorized licensed use limited to: SUN YAT-SEN UNIVERSITY. Downloaded on October 10,2025 at 14:44:54 UTC from IEEE Xplore. Restrictions apply.