Compiler Sequence Optimization Using Machine Learning Prediction Method

Diyar Mohammed[1,*], Esraa Hadi Alwan[1], Ahmed Fanfakh[1]

[1]Department of Computer Science, College of Science for Women, University of Babylon, Hillah, Iraq

Email: diyar.alkaraawy.gsci139@student.uobabylon.edu.iq; esraa.hadi@uobabylon.edu.iq; ahmed.fanfakh@uobabylon.edu.iq

Abstract

Compiler optimization is crucial in improving program performance by improving execution speed, reducing memory usage, and minimizing energy consumption. Nevertheless, modern compilers, such as LLVM, with their numerous optimizations passes, present a significant challenge in identifying the most effective sequence for optimizing a program. This study addresses the complex problem of determining optimal compiler optimization sequences within the LLVM framework, which encompasses 64 optimization passes, causing in an immense search space of 2642^{64}. Identifying the ideal sequence for even simple code can be an arduous task, as the interactions between passes are intricate and unpredictable. The primary objective of this research is to utilize machine-learning techniques to predict effective optimization sequences that outperform the default -O2 and -O3 optimization flags. The methodology involves generating 2,000 sequences per program and picking the one that achieves the shortest execution time. Three machine learning models—K-Nearest Neighbor (KNN), Decision Tree (DT), and Feedforward Neural Network (FFNN)—were employed to predict the optimization sequences based on features extracted from programs during execution. The study used benchmarks from Polybench, Shootout, and Stanford suites, each with varying problem sizes, to validate the proposed technique. The results demonstrate that the KNN model produced optimization sequences with superior performance compared to DT and FFNN. On average, KNN achieved execution times that were 2.5 times faster than those achieved using the O3 optimization flag. This research contributes to the field by programming the process of selecting optimal compiler sequences, which significantly reduces execution time and eliminates the need for manual tuning. It highlights the potential of machine learning in compiler optimization, offering a robust and scalable approach to improving program performance and setting the foundation for future advancements in the domain.

Keywords: Compiler optimization; Machine learning; Optimization sequence; Program performance; LLVM framework

1. Introduction

Contemporary compilers are vital instruments that convert high-level programming languages into machine code, facilitating the efficient execution of programs on hardware. They are engineered to enhance code execution, minimize resource usage, and elevate overall performance. Modern compilers are often categorised into three primary components: the front end, the middle end (optimizer), and the back end. Each stage possesses certain tasks in the compilation process [1]. A primary function of contemporary compilers is to generate an optimized version of the original program, improving its performance regarding execution speed, memory utilisation, and energy consumption. This procedure encompasses several stages of analysis and transformation executed on the program during compilation. [2]. Optimization is executed via optimization passes, which are methodical procedures to the intermediate representation (IR) of the program produced during the front end of compilation. Each pass aims to enhance specific elements of the program while maintaining its functional correctness, arranged in a specific order [3]. Contemporary compilers, including GCC and LLVM, provide standardized optimization levels (e.g., -O1, -O2, -O3) that regulate the degree and intensity of optimizations implemented. Each optimization level implements a distinct array of passes, varying from 100 passes in LLVM to over 200 passes in GCC, hence rendering the optimization process extremely customisable according to the application's requirements. [4,5] . An optimization sequence is the term given to each collection of optimization passes. The optimization sequence space is the set of all optimization sequences, and it is infinitely large because the probability of generating optimization sequences

DOI: https://doi.org/10.54216/FPA.180121
Received: July 20, 2024 Revised: October 24, 2024 Accepted: January 05, 2025

has increased as the number of optimization passes has increased too. Consequently, the number of optimization sequences is equal to (k)L, where k denotes the number of optimizations passes and L denotes the length of the optimization sequence. Nevertheless, the identification of a suitable optimization sequence is a challenge due to the abundance of optimization sequences that involve numerous rounds that interact in intricate ways [6] and [7]. These optimization sequences are implemented in a predetermined order. Manually tuning this order for a particular group of programs is impractical. It is also necessary to retune it whenever the compiler is updated to a new version or when a new optimization pass is introduced. Consequently, the term "optimization" is an inaccurate appellation, as these compilers may exhibit favourable performance for specific programs, but their performance is subpar for other programs (see [8]) and [9]. Additionally, there is an issue with comprehending the impact of the optimization filters' behaviour on the code segment that is to be optimized. These iterations interact with each other in an unpredictable manner during the optimization process. Consequently, there are numerous challenges associated with selecting the approaches that yield the best performance for a specific program and deciding the order in which to apply them [10] and [11]. Figure 1 illustrates the workflow for predicting optimal compiler optimization sequences using machine learning. It begins with an educating set, which undergoes optimization design of experiments to generate different compiler versions. The compiler, resulting in double star whose implementation metrics are collected, processes these versions. The data from these metrics' feeds into a machine-learning algorithm to create a predictive model. The test set is separately processed, involving feature extraction and dimension saving, to create feature vectors representing program characteristics. These feature directions are input into the predictive model, which outputs predicted outcomes that guide the selection of best compiler systems. This roadmap integrates program characterization, machine learning, and compiler optimization to enhance program performance efficiently.

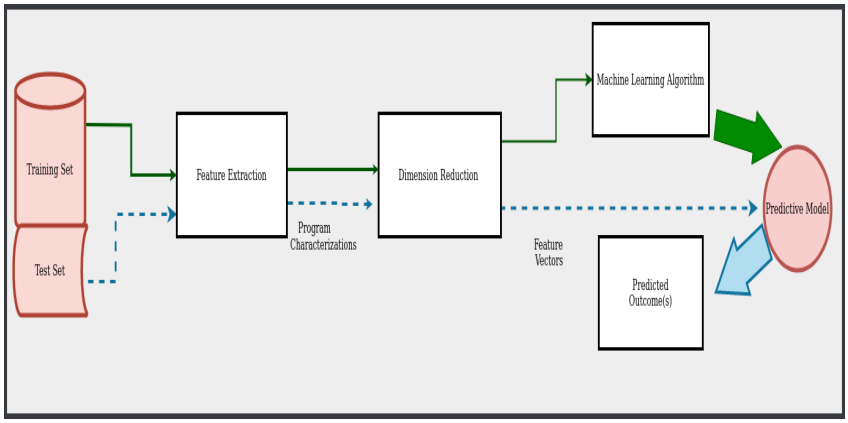

Figure 1. Workflow for Machine Learning-Based Compiler Optimization Prediction

This paper introduces a method for auto-tuning optimization sequences by constructing a prediction scheme-using machine learning to address these issues. This approach predicts the sequences of optimization passes for each program and arranges them according to the program's features by employing three classification algorithms: DT, FFNN, and KNN. The remainder of this paper is structured as follows: The related works to the proposed approach are described in Sect. (2). the introduction to machine learning is presented in Section 3.

The proposed approach and the stages of constructing a prediction scheme are illustrated in Sect. (4). the experimental results of the proposed approach are illustrated in Section (5). Ultimately, the conclusion is delineated in Section (6).

2. Related Work

A method for autonomously selecting optimal optimization orderings on a per-method basis within a dynamic compiler is presented in the paper proposed by Kulkarni and Cavazos [12]. The phase-ordering problem is modelled as a Markov process in the approach, which employs the present state of the code being optimized to generate solutions that are more effective. The technique constructs an artificial neural network that is capable of predicting advantageous optimization orderings for the code being processed through the application of neuro-evolution. In comparison to a well-designed fixed-order approach, the method was implemented in Jikes RVM and exhibited significant enhancements on a range of standard Java benchmarks. Ashouri et al. [13] introduce a machine-learning approach in their paper that is designed to enhance the efficacy of applications on embedded architectures and decrease the cost of compiler auto-tuning. The framework that has been proposed dynamically characterizes applications by utilizing Bayesian Networks and microarchitecture-independent features. The Bayesian Network approach depicts the solution as a complex probability distribution that must be sampled, rather than specifying a

DOI: https://doi.org/10.54216/FPA.180121
Received: July 20, 2024 Revised: October 24, 2024 Accepted: January 05, 2025

fixed set of compiler transformations. The method's efficacy for the selected benchmarks was demonstrated through experiments conducted on an ARM platform using the GCC transformation space. The optimized sequence of transformations was found to be near the selected set of solutions, which represented less than 10% of the search space. This set of solutions achieved a performance speedup of up to 2.8× (1.5× on average) in comparison to -O2 and -O3 for the cBench suite. Furthermore, the procedure achieved a 3× increase in search time in comparison to iterative compilation, while upholding the same quality of the solution.

The paper developed by Junior et. al. [14] presents an approach that integrates Machine Learning with Genetic Algorithms to address the challenge of finding effective compiler optimization sequences for programs. The results show that the proposed method outperforms Genetic Algorithms by up to 3.472% and Machine Learning by 4.94%. In the paper proposed by Martins et. al. [15] the authors propose a design space exploration (DSE) strategy that utilizes a clustering approach to group similar functions and explore a reduced search space by combining optimizations suggested for each group. The method identifies function similarities using a data mining technique applied to symbolic code representation, combining three algorithms: Normalized Compression Distance, Neighbour Joining, and a novel ambiguity-based clustering algorithm. The proposed approach is evaluated through experiments involving the exploration of optimization sequences within the ReflectC compiler, targeting a Xilinx MicroBlaze processor with 49 compilation passes, aimed at improving the performance of 51 functions and four applications. Results show that this clustering based DSE method significantly reduces search space exploration time (20× faster than a Genetic Algorithm approach) while achieving substantial performance speedups (41% over the baseline) with optimized code. Additional experiments conducted with the LLVM compiler, which targeted a LEON3 processor and considered 124 compilation cycles, revealed geometric mean speedups of 1.49×, 1.32×, and 1.24× for the top 10, 20, and 30 functions, respectively. Additionally, a 7% overall improvement was observed in comparison to compiling with -O2/-O2. The paper by Cavazos et. al. [16] proposes an alternative method that employs performance counters to determine the most effective compiler optimization settings. The approach entails the offline training of a model, which can subsequently be applied to novel programs to ascertain the most effective settings. The results indicate that this method is effective and, on average, two orders of magnitude quicker. Furthermore, the performance counter-based approach surpasses methodologies that depend on static code attributes. Improvements were obtained over the optimization settings of the commercial PathScale EKOPath 2.3.1-optimizing compiler on the SPEC benchmark suite, as evaluated on an AMD Athlon 64 3700+ platform, using this technique.

3. Machine Learning (ML)

Machine Learning (ML) is a subset of artificial intelligence (AI) that concentrates on creating algorithms and models enabling computers to learn from data and make judgements or predictions autonomously, without explicit programming for each task. Machine learning systems discern patterns, derive insights, and enhance their performance over time by learning from instances, rather than depending on fixed instructions. The capacity to adapt and derive insights from data renders machine learning a formidable instrument across various domains, such as computer vision, natural language processing, finance, and healthcare.

A Decision Tree is a machine-learning algorithm employed for classification and regression applications. The method operates by recursively partitioning the data set into smaller subsets according to feature values, forming a tree-like structure in which each internal node signifies a decision based on a feature, and each leaf node denotes a final output or prediction [17]. The objective is to identify the optimal splits that reduce impurity in classification or error in regression. Resolution Trees are straightforward to analyses and visualize, rendering them favoured for comprehending decision-making procedures. They can manage both numerical and categorical data with minimal p reprocessing requirements. Nonetheless, they are susceptible to over fitting, particularly with deep trees, as they may become excessively tailored to the training data and exhibit sub pre performance on novel data. Pruning and establishing depth constraints are prevalent methods to mitigate over fitting [18]. The criterion for assessing the quality of a split is Gini impurity, the maximum depth of the tree is 20, the least number of samples necessary to split an internal node is 2, and the minimum number of samples required at a leaf node is 1.

K-Nearest Neighbours (KNN) is a straightforward and intuitive supervised learning technique employed for classification and regression applications. It categorizes a data point by identifying the nearest data points, or "neighbours," in the feature space and designating the class according to the predominant class of these neighbours [19]. KNN functions under the premise that data points exhibiting analogous characteristics will be classified within the same category. The technique employs Euclidean distance to assess similarity among points. The parameter k is established as 5, indicating that the algorithm will consider the 5 closest data points for classification purposes. The distance weight is consistent, indicating that each neighbour contributes similarly to the classification decision, irrespective of its proximity to the point being classed. KNN is non-parametric, indicating that it does not presume any underlying distribution of the data.

DOI: https://doi.org/10.54216/FPA.180121
Received: July 20, 2024 Revised: October 24, 2024 Accepted: January 05, 2025

A Feed forward Neural Network (FFNN) is a category of artificial neural network commonly employed for classification and regression problems. It comprises an input layer, one or more hidden layers, and an output layer, with data progressing unidirectional from the input to the output, devoid of cycles or loops [20] and [21]. The FFNN comprises:

Input Layer: Accepts the input features from the dataset.

Hidden layers: one or more layers of neurons that process the inputs.

Output Layer: In classification tasks, the output layer employs the SoftMax activation function to produce probabilities for each class [22].

4. Proposed method

The proposed method aims to address the challenge of optimizing program execution times by leveraging machine-learning techniques to predict effective compiler optimization sequences. This methodology is designed to navigate the vast search space of possible optimization passes in modern compilers, such as LLVM, by automating the selection process and surpassing the performance of traditional optimization flags (-O2 and -O3). The workflow of the proposed method involves multiple interconnected stages, as described below:

1. Training Set and Experiment Design: The process begins with the selection of a training set comprising representative programs. These programs undergo an **optimization design of experiments**, where different combinations of compiler optimization passes are generated to create multiple compiler versions. This step systematically explores the search space of possible optimization sequences to build a diverse dataset.

2. Feature Extraction and Dimension Reduction: To enable machine-learning algorithms to process the programs, features characterizing the programs are extracted. These features might include static and dynamic attributes such as loop counts, function calls, and execution profiles. Given the high dimensionality of these features, a **dimension reduction** technique is applied to transform the data into a compact and informative representation. This ensures that the machine-learning algorithm operates efficiently while maintaining predictive accuracy.

3. Machine Learning Algorithm and Predictive Model Development: Using the reduced feature set and the collected performance metrics, machine-learning algorithms are trained to predict the effectiveness of different optimization sequences. In this study, three algorithms—K-Nearest Neighbor (KNN), Decision Tree (DT), and Feedforward Neural Network (FFNN)—are evaluated to determine their suitability for this task. The trained model acts as a **predictive model**, capable of mapping program features to optimal compiler sequences.

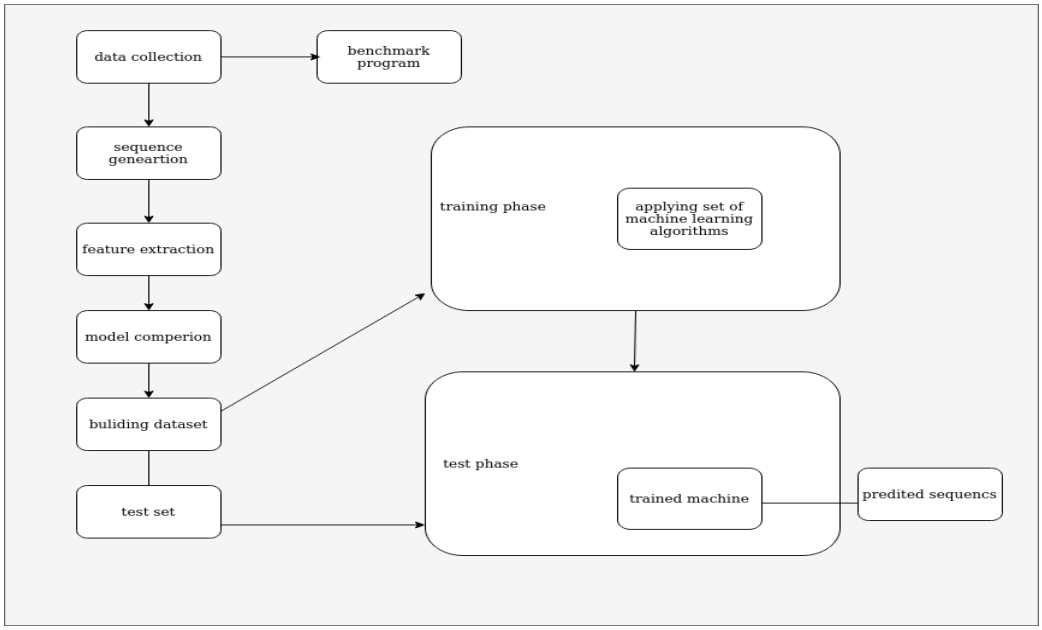

Figure 2. The proposed method

4. Test Set Prediction and Outcome Evaluation: The predictive model is validated using a separate test set of programs. These programs undergo a similar feature extraction and dimension reduction process, transforming them into feature vectors. These vectors are then input into the predictive model, which outputs predicted outcomes in the form of recommended optimization sequences. The performance of these sequences is evaluated against baseline optimization flags (O2 and O3) to measure the effectiveness of the method.

DOI: https://doi.org/10.54216/FPA.180121
Received: July 20, 2024 Revised: October 24, 2024 Accepted: January 05, 2025

This proposed approach offers a robust and scalable solution for compiler optimization, significantly improving execution times and eliminating the need for manual sequence selection. By integrating machine learning into the optimization process, this method advances the state of the art in compiler technology, paving the way for more efficient program execution across diverse domains.

In this research, we propose a method for optimizing program performance by predicting the optimal optimization sequence using machine-learning models. The proposed method involves several key steps, including data collection, sequence generation, and model training, followed by model evaluation and comparison.

4.1. Data Collection

Our experiments are working on building a database of 32 different programs with different problem sizes, selected from three different benchmarks PolyBench, Benchmarks are essential for evaluating the effectiveness of the proposed machine learning-based optimization methodology. In this study, various benchmark suites were utilized, including Polybench, Shootout, and Stanford, to test and validate the model. These benchmarks consist of programs with varying computational complexities and problem sizes, ensuring comprehensive performance evaluation. Benchmarks usually provide different problem sizes—such as small, medium, and large—enabling the study of the scalability and adaptability of the optimization sequences. Smaller difficult sizes test the method's productivity in hampered environments, while bigger sizes evaluate its ability to handle more computationally intensive tasks. By incorporating diverse problem sizes, the methodology's performance is assessed across a broad spectrum of real-world scenarios. Table 1 shows the Benchmarks with Different Problem Sizes

Table 1: Benchmarks with Different Problem Sizes

Benchmark Suite	Program Examples	Problem Sizes	Application Domain
Polybench	Matrix Multiplication, Jacobi-2D	Small, Medium, Large	Scientific computing, numerical simulations
Shootout	Fibonacci, Sorting Algorithms	Small, Medium, Large	General-purpose programming, algorithm testing
Stanford	Graph Traversals, Pathfinding	Small, Medium, Large	Data structures, graph-based applications

The use of benchmarks with different problem sizes ensures that the optimization methodology is robust and applicable to a variety of programs, from lightweight tasks to computationally demanding operations. This diversity also highlights the adaptability of machine learning models in optimizing performance across different scales and domains. Shootout and Stanford suite as shown in table 2.

Table 2: Benchmark with different problem sizes

Kernel	Problem size
Covcol	100-300-500-700-900-1100-1300-1500-1700-1900-2100-2300-2500
Dysrk	100-200-300-400-500-600-700-800-900-1000
Gemver	100-300-500-700-900-1100-1300-1500
heat-1d-	100-300-500-700-900-1100-1300-1500-1600
Mvt	100-300-500-700-900-1100-1300-1500-1700-1900-2100-2300-2500-2700-2900-3100-3300-3500-3700- 3900-4100-4300-4500-4700-4900-5100-5300-5500-5700-5900-6000
Dct	100-300-500-700-900-1100-1300-1500-1700-1900-2100-2300-2500-2700-2900-3000
Matmual	100-300-500-700-900-1100-1300-1500-1700-1900-2100-2300-2500-2700-2900-3000
Strmm	100-300-500-700-900-1100-1300-1500
Seidel	100-300-500-700-900-1000

DOI: https://doi.org/10.54216/FPA.180121
Received: July 20, 2024 Revised: October 24, 2024 Accepted: January 05, 2025

Ssymm	100-200-300-400-500
Tmm	100-300-500-700-900-1100-1300-1500-1700-1900-2100-2300-2500-2700-2900-3000
Lu	100-300-500-700-900-1100-1300-1500-1700-1900-2000
Template	100-300-500-700-900-1000
fdtd-1d	100-300-500-700-900-1100-1300-1500-1700-1800
heat-2d	100-300-500-700-900-1100-1300-1500-1700-1900-2100-2300-2500-2700-2900-3100-3300-3500-3700-3900-4000
jacobi-1d	100-300-500-700-900-1100-1300-1500-1700-1900-2100-2300-2500-2700-2900-3000
jacobi-2d	100-200-300-400-500-600
fdtd-2d	100-200-300-400-500-600-700-800-900-1000
Corcol	100-200-300-400-500-600
dsyr2k	100-200-300-400-500-600
Arr	100-300-500-700-900-1100-1300-1500-1700-1900-2100-2300-2500-2700-2900-3100-3300-3500-3700-3900-4100-4300-4500-4700-4900-5100-5300-5500-5700-5900-6100-6300-6500-6700-6900-7100-7300-7500-7700-7900-8000
Dct	124-224-324-424-524-624-724-824-924-1024
Mvt	400-800-1200-1600-2000-1400-1800-2200-2600-3000-3400-3800-4200-4600-5000-5400-5800-6200-6600-7000-7400-7800-4700-8200-8600-9000-9400-9800-10000
advect3d	100-200-300
dsyr2k	148-348-548-748-948-1148-1348-1548-1748-1948-2048
Gemver	200-400-600-800-1000-1200-1400-1600-1800-2000-2200-2400-2600-2800-3000-3200-3400-3600-3800-4000-4200-4400-4600-4800-5000-5200-5400-5600-5800-6000
fdtd-2d	248-448-648-848-1028-1248-1448-1648-1848-2048
Tile	100-200-300-400-500-600-700-800
Oourafft	124-224-324-424-524-624-724-824-924-1024
Seidel	200-400-600-800-1000-1200-1400-1600-1800-2000
Ssymm	200-400-600-800-1000-1200-1400-1600-1800-2000
gam-of-life	100-200-300

Each program compiles with 2000 randomly generated optimization sequences. These sequences are generated from 60 LLVM passes, which are shown in table 4. The best sequence for each program produces less execution time than -O2 or -O3, are included in our data set. Moreover, all programs are also compiled and executed with optimization flag -O3 -O2, and these times are included in the dataset. We compared the results of these sequences with traditional -O2 and -O3 optimizations flags.

DOI: https://doi.org/10.54216/FPA.180121
Received: July 20, 2024 Revised: October 24, 2024 Accepted: January 05, 2025

Table 3: O3 passes

List of Optimization Passes			
-scalarrepl	-always-inline	-argpromotion	-codegenprepare
-constmerge	-constprop	-correlated-propagation	-dce
-deadargelim	-die	-dse	-early-cse
-globaldce	-globalopt	-gvn	-indvars
-inline	-instcombine	-instsimplify	-memdep
-ipconstprop	-ipsccp	-jump-threading	-licm
-loop-deletion	-loop-idiom	-loop-instsimplify	-loop-reduce
-loop-rotate	-loop-simplify	-loop-unroll	-loop-unswitch
-loops	-lower-expect	-loweratomic	-lowerinvoke
-lowerswitch	-memcpyopt	-mergefunc	-mergereturn
-partial-inliner	-prune-eh	-reassociate	-adce
-sccp	-strip-dead-prototypes	-simplifycfg	-sink
-tailcallelim	-targetlibinfo	-no-aa	-tbaa
-basicaa	-basiccg	-functionattrs	-scalarrepl-ssa
-domtree	-lazy-value-info	-lcssa	-scalar-evolution

This process is repeated for all programs in the dataset, resulting in a comprehensive dataset of optimal execution sequences.

4.2 Features extraction

The dynamic dataset includes features that vary during program execution. These dynamic characteristics are obtained using the Linux 'perf' tool, which provides an empirical picture of the program's dynamic behavior as it interacts with the computing machine while running. For each program, 35 dynamic features are collected. The Perf tool collects hardware performance counter data throughout the execution of benchmark programs. The dynamic features include cache misses, cache hits, and other relevant hardware performance metrics as shown in table 4. These events (features) are also included in the dataset.

Table 4: List of dynamic features

Event	type
Cpu-cycles or cycles, instructions, Cache-references, Cache-misses, Bus_cycles,branch-instruction,branch-misses	Timed Profiling
Cpu_clock(msec), Task_clock(msec), Page-faults OR faults, Context-switches, Cpu-migrations, Alignment-faults, Emulation-faults	Software event
L1-dcache-loads, L1-dcache-loads misses, L1-dcache-stores, L1-dcache-stores-misses,L1-dcashe-prefetches,l1-dcash-prefetches-misses, L1-icache-loads,L1-icash-loads-store,L1-icash-load-stores-misses, L1-icache-loads misses, L1-icache-loads-prefetches, L1-icache- prefetches –misses, LLC-load, LLC-loads misses, LLC-strores, LLC-strores misses, LLC-prefetch-misses, Dtlb-loads, Dtlb-loads misses, Dtlb-store, Dtlb-store-misses, Dtlb-prefetches, Dtlb-prefetches -misses, Itlb -loads, Itlb –loads -misses, Branch-loads, Branch-loads-misses	User Statically-Defined Tracing
Stalled_cycles-frontend , Stalled_cycle-backend, Sched:sched-stat-runtime , Sched:sched-pi-setprio, Syscalls:sys_enter_socket, Kvmmmu:kvm_mmu_pagetable_walk	Dynamic Tracing
Sched:sched_process_exec, Sched:sched_process_frok, Sched:sched_process_wait, Sched:sched_process_wait_task, Sched:sched_process_exit	Kernel Trace point event

4.3 Data Splitting and Model Training

We use three classifier and divided data into 80% for training and 20% for testing to ensure the models can generalize well to unseen data. All model of classifier contains 35 input and 18 output. The following machine learning models are trained on the dataset.

Decision Tree: DT builds a decision tree to classify data into 18 categories using 35 input features. Pre-processing, optional feature selection and careful hyperactive parameter tuning are crucial for achieving good performance and avoiding overfitting, especially given the high dimensionality of the input data.

The classification methodology employs the decision tree algorithm as detailed in Algorithm 1 below.

DOI: https://doi.org/10.54216/FPA.180121
Received: July 20, 2024 Revised: October 24, 2024 Accepted: January 05, 2025

Algorithm 1: DT

Input: Feature matrix X (N x d), Label vector y (N)

Output: Decision tree model

Steps:

1. Start
2. If all samples in y belong to the same class:
    a. Create a leaf node with that class label
    b. Return the node
3. If no features are left to split:
    a. Create a leaf node with the majority class in y
    b. Return the node
4. Select the best feature to split using a criterion (e.g., Gini, entropy) # Measure feature importance
5. Partition the dataset based on the selected feature's values
6. Recursively repeat steps 2–5 for each subset of data
7. Return the root node of the decision tree # This serves as the trained model
8. End

K-Nearest Neighbours: KNN works by finding the nearest data points to the test data and assigning the class label that is most common among those neighbours. Since the algorithm will look at the three closest neighbours, check their class labels, and predict the class that appears most frequently. This approach is simple but effective for classification tasks, especially when the relationships between features are non-linear.

The classification methodology employs the k-nearest neighbour's algorithm as detailed in Algorithm 2 below.

Algorithm 2: KNN

Input: Training data (X_train, y_train), Test instance x_test, Number of neighbor's k

Output: Predicted class for x_test

Steps:

1. Start
2. Calculate the distance between x_test and all instances in X_train
3. Sort the distances in ascending order
4. Select the top k nearest neighbors
5. Identify the majority class among the k neighbors
6. Assign the majority class to x_test
7. End

Feed forward Neural Network: A neural network consists of three layers (one input, two hidden, and one output) with 20 training epochs. Each neuron in the hidden layers receives weighted inputs from the previous layer and produces an output after applying the activation function. The weights are continuously updated to minimize prediction error, allowing the network to learn complex relationships between variables and accurately predict output classes.

The classification methodology employs the feed forward neural network algorithm as detailed in Algorithm 3 below.

DOI: https://doi.org/10.54216/FPA.180121
Received: July 20, 2024 Revised: October 24, 2024 Accepted: January 05, 2025

Algorithm 3: FFNN

Input: Feature matrix X (N x d), Label matrix y (N x c), Number of layers L, Epochs E, Learning rate lr

Output: Trained neural network model

Steps:

1. Start
2. Initialize weights and biases for all layers
3. For each epoch:
    a. Perform forward propagation:
        i. Compute activation for each layer using the activation function
    b. Calculate the loss (e.g., cross-entropy for classification)
    c. Perform back propagation:
        i. Compute gradients for weights and biases
        ii. Update weights and biases using the learning rate
4. Return the trained model
5. End

Each model is trained on the training data set using its respective optimization techniques. For example, KNN is trained relies on distance between points in the feature spaces to achieve the best results. During training, both accuracy and loss function are tracked to measure the model's performance and improvement over time. After training, the models are evaluated on the testing dataset to compare their accuracy. The ability of each model to predict the optimal execution sequences is assessed based on its accuracy. The accuracy of each model was calculated, and the results are used to determine which model performed best.

5. Experimental Results

The findings of the proposed method are described in this section. The case study encompasses programs from a variety of benchmarking sites. The LLVM compiler's standard optimization level –0O, -O2, and -O3 is employed to evaluate the proposed technique. To obtain precise results, it is essential to compile the programs using Clang's -O0 level, which indicates that no optimizations are implemented, before executing any optimization sequences. This is crucial in the beginning. Subsequently, the "-scalarrepl" (scalar replacement) is employed in conjunction with each pass to activate additional transformation passes.

The optimization sequences (derived sequences) are applied to the program after it has been converted to LLVM Intermediate Representation (IR), a machine-readable bit code (.bc) file format. The Clang compiler, which is a front end for the LLVM C language, is employed to convert the source code programs to the bit code file format. The algorithms are compiled using two practical tools: opt and llc. The optimization utility "opt" executes a series of iterations to optimize the source code program and subsequently stores the optimized code in a bit file format. The object code is generated by the utility llc through the conversion of the bit file format. Ultimately, the executable code is produced using Clang. Some of the details that evaluate the machine used in the proposed approach are presented in table 5.

The table 6 presents the performance of three machine-learning algorithms (KNN, DT, FFNN) on a classification task. Performance is measured by classification accuracy and loss function.

KNN: Demonstrated moderate accuracy (86%) and low loss (2.922). A simple and efficient algorithm, but performance is parameter dependent.

DT: Achieved the highest accuracy (88%) but with higher loss (5.069) compared to KNN. This may indicate overfitting or excessive complexity.

FFNN: Showed very poor performance (34% accuracy, 9.217 loss), suggesting issues with network architecture, training parameters, training data, or data preprocessing.

DOI: https://doi.org/10.54216/FPA.180121
Received: July 20, 2024 Revised: October 24, 2024 Accepted: January 05, 2025

Table 5: Machine Details

Processor model	**Intel (R) Core (TM) i7 CPU**
Processor speed	2.60 GHz
RAM	16 GB
Operating System	Linux Ubuntu 18.04.6
Compiler	Clang/LLVM

Table 6: machines learning accuracy and loss function results

Algorithms classify	**Accuracy**	**Loss**
KNN	86%	2.922
Dt	88%	5.069
FFNN	34%	9.217

Tables 7, 8 and 9 below show the performance of the set of the test programs with different machines learning approaches. The first column of these tables represents the program name, while the second indicates the program's problem size. The third column shows the sequences number, and the fourth specifies the passes involved in configuring these sequences. The sixth column displays the program's execution time after applying the predicted sequence. The seventh, and eighth and ninth columns present the program's execution time with the flags -O0, -O2, and -O3, respectively.

Table 7: the execution time the testing set (KNN).

Test	**Problem size**	**No of seq**	**Pass**	**Obtained Seq**	**O0**	**O2**	**O3**
Dsyrk.c	320	seq2	-loop-simplify -scalar-evolution -correlated-propagation -globalopt -ipsccp -no-aa -globaldce -scalarrepl -constmerge -correlated-propagation -constmerge -instsimplify -instcombine -loop-deletion -simplifycfg -dce -mergefunc -loops -prune-eh -simplifycfg -loop-unswitch -adce -instsimplify -instsimplify -ipsccp -basiccg -instsimplify -prune-eh -dse -lazy-value-info -deadargelim -early-cse -constprop -simplifycfg -gvn -simplifycfg -dce -die -simplifycfg -simplifycfg -die -globaldce -ipconstprop	0.015s	0.027s	0.027s	0.026s
2mm.c	512	seq40	-basicaa -argpromotion -simplifycfg -memcpyopt -inline -gvn -globaldce -adce -loop-rotate -loop-unswitch -globalopt	0.476s	0.646s	0.645s	0.651s

DOI: https://doi.org/10.54216/FPA.180121
Received: July 20, 2024 Revised: October 24, 2024 Accepted: January 05, 2025

Covcol.c	1350	seq81	-sink -basicaa -adce -die -globalopt -instcombine -mergereturn -dce -targetlibinfo -loops -loop-simplify -basicaa -tailcallelim -tbaa -domtree -die -functionattrs -instcombine -tailcallelim -instcombine -globalopt -memcpyopt -loop-deletion -no-aa -scalarrepl -lowerinvoke -constprop -loop-instsimplify -adce -globalopt -ipconstprop -scalarrepl -functionattrs -loop-instsimplify -deadargelim -jump-threading -gvn -tbaa -partial-inliner -sink -tailcallelim -strip-dead-prototypes -loop-instsimplify -scalar-evolution -die -codegenprepare -inline -tbaa -simplifycfg -adce -basicaa -jump-threading -memcpyopt -tbaa -loops -loops -no-aa -targetlibinfo	0.208s	4.306s	4.474s	4.621s
Floyd.c	2048	seq164	-no-aa -tbaa -basicaa -globalopt -ipsccp -deadargelim -instcombine -simplifycfg -basiccg -prune-eh -inline -functionattrs -argpromotion -sroa -domtree -early-cse -simplify-libcalls -lazy-value-info -jump-threading -correlated-propagation -simplifycfg -instcombine -tailcallelim -simplifycfg -reassociate -domtree -loops -loop-simplify -lcssa -loop-rotate -licm -lcssa -loop-unswitch -instcombine -scalar-evolution -loop-simplify -lcssa -indvars -loop-idiom -loop-deletion -loop-unroll -memdep -gvn -memdep -memcpyopt -sccp -instcombine -lazy-value-info -jump-threading -correlated-propagation -domtree -memdep -dse -adce -simplifycfg -instcombine -strip-dead-prototypes -globaldce -constmerge -preverify -domtree –verify	6.049s	19.757s	6.145s	6.116s
Covcol.c	1150	seq3	-instcombine -loop-deletion -loop-simplify -scalar-evolution -correlated-propagation -globalopt -ipsccp -no-aa -globaldce -scalarrepl -constmerge -correlated-propagation -constmerge -instsimplify -instcombine -loop-deletion -simplifycfg -dce -mergefunc -loops -prune-eh -simplifycfg -loop-unswitch -adce -instsimplify -instsimplify -ipsccp -basiccg -instsimplify -prune-eh -dse -lazy-value-info -deadargelim -early-cse -constprop -simplifycfg -gvn -simplifycfg -dce -die -simplifycfg -simplifycfg -die -globaldce -ipconstprop -dce -domtree -lowerswitch -scalarrepl -sccp -gvn -loop-unroll -partial-inliner -lowerswitch -prune-eh -basiccg	1.694s	2.032s	1.834s	1.946s
Unseen8.c		seq370	-simplifycfg -dce -mergefunc -loops -prune-eh -simplifycfg -loop-unswitch -adce -instsimplify -instsimplify -ipsccp -basiccg -instsimplify -prune-eh -dse -lazy-value-info -deadargelim -early-cse -constprop -simplifycfg -gvn -simplifycfg -dce -die -simplifycfg -simplifycfg -die -globaldce -ipconstprop -dce -domtree -lowerswitch -scalarrepl -sccp -gvn -	6.274s	9.956s	9.997s	10.099s

DOI: https://doi.org/10.54216/FPA.180121
Received: July 20, 2024 Revised: October 24, 2024 Accepted: January 05, 2025

			loop-unroll -partial-inliner -lowerswitch -prune-eh -basiccg				

Tables 8. The execution time the testing set (Dt).

Test	Problem size	Seq. No.	Pass	Obtained Seq	-O0	-O2	-O3
Dsyrk.c	320	seq2	-loop-simplify -scalar-evolution -correlated-propagation -globalopt -ipsccp -no-aa -globaldce -scalarrepl -constmerge -correlated-propagation -constmerge -instsimplify -instcombine -loop-deletion -simplifycfg -dce -mergefunc -loops -prune-eh -simplifycfg -loop-unswitch -adce -instsimplify -instsimplify -ipsccp -basiccg -instsimplify -prune-eh -dse -lazy-value-info -deadargelim -early-cse -constprop -simplifycfg -gvn -simplifycfg -dce -die -simplifycfg -simplifycfg -die -globaldce -ipconstprop	0.015s	0.027s	0.027s	0.026s
2mm.c	512	seq40	-basicaa -argpromotion -simplifycfg -memcpyopt -inline -gvn -globaldce -adce -loop-rotate -loop-unswitch -globalopt	0.476s	0.646s	0.645s	0.651s
Covcol.c	1350	seq81	-sink -basicaa -adce -die -globalopt -instcombine -mergereturn -dce -targetlibinfo -loops -loop-simplify -basicaa -tailcallelim -tbaa -domtree -die -functionattrs -instcombine -tailcallelim -instcombine -globalopt -memcpyopt -loop-deletion -no-aa -scalarrepl -lowerinvoke -constprop -loop-instsimplify -adce -globalopt -ipconstprop -scalarrepl -functionattrs -loop-instsimplify -deadargelim -jump-threading -gvn -tbaa -partial-inliner -sink -tailcallelim -strip-dead-prototypes -loop-instsimplify -scalar-evolution -die -codegenprepare -inline -tbaa -simplifycfg -adce -basicaa -jump-threading -memcpyopt -tbaa -loops -loops -no-aa –targetlibinfo	0.208s	4.306s	4.474s	4.621s
Floyd.c	2048	seq1	-targetlibinfo -early-cse -loops -lcssa -mergefunc -indvars -loop-unswitch -deadargelim -indvars -jump-threading -loweratomic -reassociate -scalar-evolution -mergereturn -scalarrepl -loop-idiom -loop-instsimplify -basicaa -strip-dead-prototypes -loop-unroll -scalarrepl-ssa -correlated-propagation -loop-instsimplify -licm -ipconstprop -prune-eh	7.493s	19.567s	19.589s	19.764s
Covcol.c	1150	seq3	-instcombine -loop-deletion -loop-simplify -scalar-evolution -correlated-propagation -globalopt -ipsccp -no-aa -globaldce -scalarrepl -constmerge -correlated-propagation -constmerge -	1.694s	2.032s	1.834s	1.946s

DOI: https://doi.org/10.54216/FPA.180121
Received: July 20, 2024 Revised: October 24, 2024 Accepted: January 05, 2025

			instsimplify -instcombine -loop-deletion -simplifycfg -dce -mergefunc -loops -prune-eh -simplifycfg -loop-unswitch -adce -instsimplify -instsimplify -ipsccp -basiccg -instsimplify -prune-eh -dse -lazy-value-info -deadargelim -early-cse -constprop -simplifycfg -gvn -simplifycfg -dce -die -simplifycfg -simplifycfg -die -globaldce -ipconstprop -dce -domtree -lowerswitch -scalarrepl -sccp -gvn -loop-unroll -partial-inliner -lowerswitch -prune-eh -basiccg				
Unseen.c		seq3	-instcombine -loop-deletion -loop-simplify -scalar-evolution -correlated-propagation -globalopt -ipsccp -no-aa -globaldce -scalarrepl -constmerge -correlated-propagation -constmerge -instsimplify -instcombine -loop-deletion -simplifycfg -dce -mergefunc -loops -prune-eh -simplifycfg -loop-unswitch -adce -instsimplify -instsimplify -ipsccp -basiccg -instsimplify -prune-eh -dse -lazy-value-info -deadargelim -early-cse -constprop -simplifycfg -gvn -simplifycfg -dce -die -simplifycfg -simplifycfg -die -globaldce -ipconstprop -dce -domtree -lowerswitch -scalarrepl -sccp -gvn -loop-unroll -partial-inliner -lowerswitch -prune-eh -basiccg	6.284s	9.977s	9.951s	9.985s

Table 9: the execution time the testing set (FFNN).

Test	**Problem size**	**No of seq**	**Pass**	**Obtained Seq**	**O0**	**O2**	**O3**
Dsyrk.c	320	Seq1	-targetlibinfo -early-cse -loops -lcssa -mergefunc -indvars -loop-unswitch -deadargelim -indvars -jump-threading -loweratomic -reassociate -scalar-evolution -mergereturn -scalarrepl -loop-idiom -loop-instsimplify -basicaa -strip-dead-prototypes -loop-unroll -scalarrepl-ssa -correlated-propagation -loop-instsimplify -licm -ipconstprop -prune-eh	0.013s	0.027s	0.027s	0.027s
Floyd.c	2048	seq1	-targetlibinfo -early-cse -loops -lcssa -mergefunc -indvars -loop-unswitch -deadargelim -indvars -jump-threading -loweratomic -reassociate -scalar-evolution -mergereturn -scalarrepl -loop-idiom -loop-instsimplify -basicaa -strip-dead-prototypes -loop-unroll -scalarrepl-ssa -correlated-propagation -loop-instsimplify -licm -ipconstprop -prune-eh	7.493s	19.567s	19.589s	19.764s

DOI: https://doi.org/10.54216/FPA.180121
Received: July 20, 2024 Revised: October 24, 2024 Accepted: January 05, 2025

2mm.c	512	seq1	-targetlibinfo -early-cse -loops -lcssa -mergefunc -indvars -loop-unswitch -deadargelim -indvars -jump-threading -loweratomic -reassociate -scalar-evolution -mergereturn -scalarrepl -loop-idiom -loop-instsimplify -basicaa -strip-dead-prototypes -loop-unroll -scalarrepl-ssa -correlated-propagation -loop-instsimplify -licm -ipconstprop -prune-eh	0.572s	0.649s	0.648s	0.650s
Covcol.c	1350	seq1	-targetlibinfo -early-cse -loops -lcssa -mergefunc -indvars -loop-unswitch -deadargelim -indvars -jump-threading -loweratomic -reassociate -scalar-evolution -mergereturn -scalarrepl -loop-idiom -loop-instsimplify -basicaa -strip-dead-prototypes -loop-unroll -scalarrepl-ssa -correlated-propagation -loop-instsimplify -licm -ipconstprop -prune-eh	4.333s	4.334s	4.143s	4.239s
Covcol.c	1150	seq1	-targetlibinfo -early-cse -loops -lcssa -mergefunc -indvars -loop-unswitch -deadargelim -indvars -jump-threading -loweratomic -reassociate -scalar-evolution -mergereturn -scalarrepl -loop-idiom -loop-instsimplify -basicaa -strip-dead-prototypes -loop-unroll -scalarrepl-ssa -correlated-propagation -loop-instsimplify -licm -ipconstprop -prune-eh	1.807s	1.794s	1.775s	1.780s
Unseen.c		seq1	-targetlibinfo -early-cse -loops -lcssa -mergefunc -indvars -loop-unswitch -deadargelim -indvars -jump-threading -loweratomic -reassociate -scalar-evolution -mergereturn -scalarrepl -loop-idiom -loop-instsimplify -basicaa -strip-dead-prototypes -loop-unroll -scalarrepl-ssa -correlated-propagation -loop-instsimplify -licm -ipconstprop -prune-eh	6.107s	9.996s	9.984s	9.968s

Figures 3, 4, and 5 provide a comparison of three models: DT, KNN, and FFNN, based on the test program results, highlighting the superiority of the KNN model compared to the others

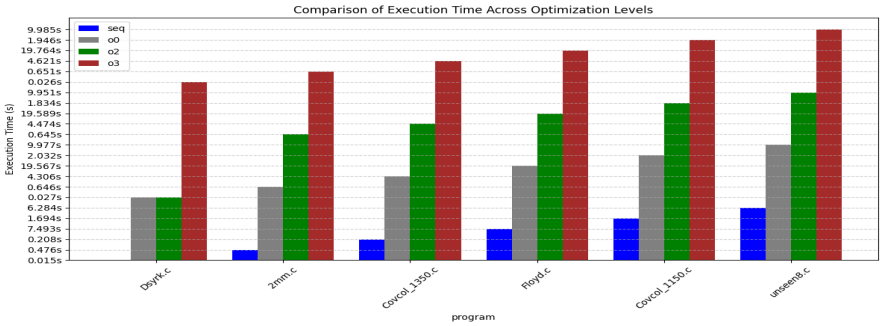

Figure 3. The result of DT test program

DOI: https://doi.org/10.54216/FPA.180121
Received: July 20, 2024 Revised: October 24, 2024 Accepted: January 05, 2025

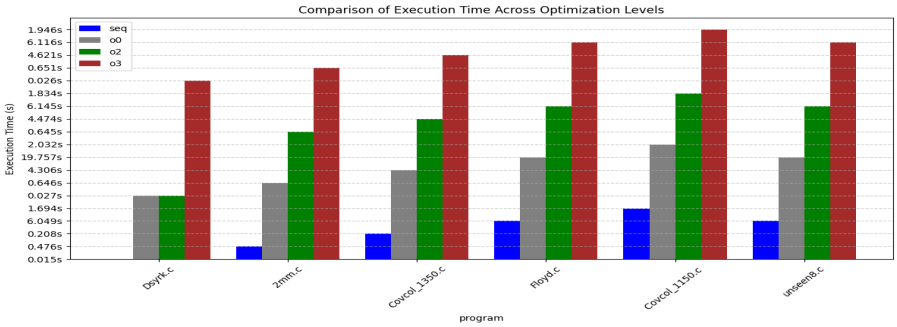

Figure 4. The result of KNN test program

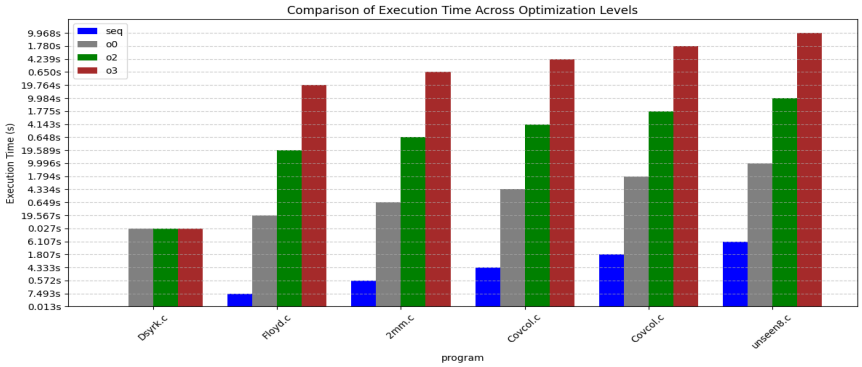

Figure 5. The result of FFNN test program

5.1 Model Comparsion

The models were compared based on their performance on the test data. For instance, the KNN algorithm demonstrated the best performance, achieving an average sequence execution time of 2.415 ms, while the FFNN model performed the worst, with an average execution time of 3.704 ms Moreover, these results indicate that, although the DT model achieved the highest accuracy, the KNN model demonstrated better average execution time. Table 10 presents the results for the three models.

Table 10: The comparison of average execution time results of the proposed ML models and O3 flag

Model Classifier	Obtained sequences	O3 sequence
DT	2.695	6.17
KNN	2.415	
FFNN	3.704	

6. Conclusion

In this work, we deliberated on formulating a predictive approach to determine the appropriate optimization sequence via machine learning approaches, with the objective of maximizing performance regarding execution time in comparison to -O2 and -O3 optimizations. The research seeks to identify the ideal sequence that minimizes execution time specifically emphasizing the utilization of machine learning to accomplish this objective. We constructed an extensive dataset of benchmark programs with diverse problem sizes and trained machine-learning models, including K-Nearest Neighbors (KNN), Decision Tree (DT Feed forward Neural Network (FFNN). The findings indicate that the suggested predictive technique may generate optimization sequences that yield shorter execution times than O3 optimization in numerous instances, hence illustrating the method's efficacy in enhancing

DOI: https://doi.org/10.54216/FPA.180121
Received: July 20, 2024 Revised: October 24, 2024 Accepted: January 05, 2025

performance and minimizing execution time. On average, the KNN predicted sequence is faster than O3 by 2.5x. This research validates that machine-learning methodologies offer robust instruments for selecting and optimizing the sequence of enhancements, exceeding conventional optimization methods like -O2 and -O3, hence improving overall program performance.

Funding: "This research received no external funding"

Conflicts of Interest: "The authors declare no conflict of interest."

References

[1] L. H. Alhasnawy, E. H. Alwan, and A. B. M. Fanfakh, "Using Machine Learning to Predict the Sequences of Optimization Passes," in Proceedings of a Conference, 2020, pp. 139–156. doi: 10.1007/978-3-030-55340-1_10.

[2] E. H. Alwan and A. K. M. Al-Qurabat, "Candidate Best Optimizations Sequences for Code Size Reduction," Ingénierie des systèmes d'information, vol. 29, no. 4, pp. 1611–1617, Aug. 2024, doi: 10.18280/isi.290434.

[3] P. Ghimire, "Machine learning-based prediction models for budget forecast," Machine Learning-Based Prediction Models for Budget Forecast, 2023.

[4] T. C. de Souza Xavier and A. F. da Silva, "Exploration of compiler optimization sequences using a hybrid approach," Computing and Informatics, vol. 37, no. 1, pp. 165–185, 2018.

[5] N. L. Queiroz, L. G. A. Rodriguez, and A. F. da Silva, "Combining machine learning with a genetic algorithm to find good compiler optimizations sequences," in Proceedings of ICEIS, vol. 1, 2017, pp. 397–404.

[6] M. H. Almohammed, A. B. M. Fanfakh, and E. H. Alwan, "Parallel Genetic Algorithm for Optimizing Compiler Sequences Ordering," NTICT 2020, CCIS 1183, pp. 128–138, 2020.

[7] Z. Pan and R. Eigenmann, "Fast and effective orchestration of compiler optimizations for automatic performance tuning," in Proceedings of the International Symposium on Code Generation and Optimization, IEEE Computer Society, 2006, pp. 319–332.

[8] S. Purini and L. Jain, "Finding good optimization sequences covering program space," ACM Transactions on Architecture and Code Optimization (TACO), vol. 9, no. 4, article 56, 2013.

[9] M. H. Almohammed, E. H. Alwan, and A. B. M. Fanfakh, "Programs features clustering to find optimization sequence using genetic algorithm," in ICICCT 2019, LAIS, vol. 9, Springer, Cham, 2020, pp. 40–50. doi: 10.1007/978-3-030-38501-9_4.

[10] Z. S. Alkaaby, E. H. Alwan, and A. B. M. Fanfakh, "Finding a good global sequence using multi-level genetic algorithm," Journal of Engineering and Applied Sciences, vol. 13, pp. 9777–9783, 2018.

[11] R. M. Al Baity, E. H. Alwan, and A. B. M. Fanfakh, "A top popular approach for the automatic tuning of compiler optimizations," in AIP Conference Proceedings, vol. 2398, no. 1, AIP Publishing, 2022.

[12] 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, New York, NY, USA: ACM, 2012, pp. 147–162. doi: 10.1145/2384616.2384628.

[13] H. Ashouri, G. Mariani, G. Palermo, and C. Silvano, "A Bayesian network approach for compiler auto-tuning for embedded processors," in 2014 IEEE 12th Symposium on Embedded Systems for Real-time Multimedia (ESTIMedia), IEEE, 2014, pp. 90–97. doi: 10.1109/ESTIMedia.2014.6962349.

[14] N. L. Queiroz Junior, L. G. A. Rodriguez, and A. F. da Silva, "Combining Machine Learning with a Genetic Algorithm to Find Good Compiler Optimizations Sequences," in Proceedings of the 19th International Conference on Enterprise Information Systems, SCITEPRESS, 2017, pp. 397–404. doi: 10.5220/0006270403970404.

[15] L. G. A. Martins, R. Nobre, J. M. P. Cardoso, A. C. B. Delbem, and E. Marques, "Clustering-Based Selection for the Exploration of Compiler Optimization Sequences," ACM Transactions on Architecture and Code Optimization, vol. 13, no. 1, pp. 1–28, Apr. 2016, doi: 10.1145/2883614.

DOI: https://doi.org/10.54216/FPA.180121
Received: July 20, 2024 Revised: October 24, 2024 Accepted: January 05, 2025

[16] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. F. P. O'Boyle, and O. Temam, "Rapidly Selecting Good Compiler Optimizations using Performance Counters," in International Symposium on Code Generation and Optimization (CGO'07), IEEE, 2007, pp. 185–197. doi: 10.1109/CGO.2007.32.

[17] J. R. Quinlan, "Induction of decision trees," Machine Learning, vol. 1, pp. 81–106, 1986.

[18] Z. Yu, F. Haghighat, B. C. Fung, et al., "A decision tree method for building energy demand modeling," Energy and Buildings, vol. 42, no. 10, pp. 1637–1646, 2010.

[19] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone, Classification and Regression Trees, Belmont, CA: Wadsworth, 1986.

[20] A. A. Amra and A. Y. Maghari, "Students performance prediction using KNN and Naive Bayesian," in 2017 8th International Conference on Information Technology (ICIT), IEEE, 2017, pp. 909–913.

[21] S. V. Kozyrev, "Classification by ensembles of neural networks," Steklov Mathematical Institute, Feb. 21, 2012.

[22] Jaafar, N. Ismail, M. Tajjudin, R. Adnan, and M. H. F. Rahiman, "Z-score and feed-forward neural network (FFNN) for flood modeling at Kuala Krai Station," in 2016 7th IEEE Control and System Graduate Research Colloquium (ICSGRC), IEEE, 2016, pp. 92–97. doi: 10.1109/ICSGRC.2016.7813308.

DOI: https://doi.org/10.54216/FPA.180121
Received: July 20, 2024 Revised: October 24, 2024 Accepted: January 05, 2025