IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 15 August 2024, accepted 21 August 2024, date of publication 29 August 2024, date of current version 10 September 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3451516

RESEARCH ARTICLE

Selecting the Best Compiler Optimization by Adopting Natural Language Processing

HAMEEZA AHMED[1], MUHAMMAD FAHIM UL HAQUE[2], HASHIM RAZA KHAN[3,4], GHALIB NADEEM[3], KAMRAN ARSHAD[5,6], (Senior Member, IEEE), KHALED ASSALEH[5,6], (Senior Member, IEEE), AND PAULO CESAR SANTOS[7]

[1]Department of Computer and Information Systems Engineering, NED University of Engineering and Technology, Karachi 75270, Pakistan
[2]Department of Telecommunications Engineering, NED University of Engineering and Technology, Karachi 75270, Pakistan
[3]Department of Engineering Sciences and Technology, Iqra University, Karachi 75500, Pakistan
[4]Neurocomputation Lab, National Centre of Artificial Intelligence, NED University of Engineering and Technology, Karachi 75270, Pakistan
[5]Department of Electrical and Computer Engineering, College of Engineering and Information Technology, Ajman University, Ajman, United Arab Emirates
[6]Artificial Intelligence Research Centre, Ajman University, Ajman, United Arab Emirates
[7]Institute of Informatics, Federal University of Rio Grande do Sul, Porto Alegre 91509-900, Brazil

Corresponding authors: Hameeza Ahmed (hameeza@neduet.edu.pk) and Kamran Arshad (k.arshad@ajman.ac.ae)

This work was supported by Ajman University Internal Research under Grant 2023-IRG-ENIT-3.

ABSTRACT *Compiler* is a tool that converts the high-level language into assembly code after enabling relevant optimizations. The automatic selection of suitable optimizations from an ample optimization space is a non-trivial task mainly accomplished through hardware profiling and application-level features. These features are then passed through an intelligent algorithm to predict the desired optimizations. However, collecting these features requires executing the application beforehand, which involves high overheads. With the evolution of Natural Language Processing (NLP), the performance of an application can be solely predicted at compile time via source code analysis. There has been substantial work in source code analysis using NLP, but most of it is focused on offloading the computation to suitable devices or detecting code vulnerabilities. Therefore, it has yet to be used to identify the best optimization sequence for an application. Similarly, most works have focused on finding the best machine learning or deep learning algorithms, hence ignoring the other important phases of the NLP pipeline. This paper pioneers the use of NLP to predict the best set of optimizations for a given application at compile time. Furthermore, this paper uniquely studies the impact of four vectorization and seven regression techniques in predicting the application performance. For most applications, we show that *tfidf* vectorization and *huber* regression result in the best outcomes. On average, the proposed technique predicts the optimal optimization sequence with a performance drop of 18%, achieving a minimum drop of merely 0.5% compared to the actual best combination.

INDEX TERMS Compiler, optimization, source code analysis, natural language processing, vectorization, regression.

I. INTRODUCTION

The emergence of new hardware paradigms and the widespread use of high-level languages make the role of the compiler even more critical than ever before. A *compiler* is a tool that translates a high-level language program into the assembly and incorporates the relevant optimizations into the code [1]. It is true that the hardware resources can only be utilized to the fullest if the generated assembly code is efficient. Therefore, this implies that despite the presence of powerful hardware designs, performance goals still need to be met due to a lack of competent software solutions. Among the millions of optimizations available inside the compilers, the appropriate optimizations for the given application can extract higher performance, energy efficiency, and reduced development time by efficiently

The associate editor coordinating the review of this manuscript and approving it for publication was Mu-Yen Chen.

utilizing the hardware resources [2], [3], [4], [5]. However, the automatic selection of suitable optimization for compiling an application is a troublesome job. In literature, it has been done by predicting the appropriate optimization through any intelligent algorithm trained by employing both application and hardware-level features [4], [5], [6], [7], [8]. The collection of these dynamic features requires executing an application, thus incurring high execution costs, especially for big data applications [3], [8].

Natural Language Processing (NLP) has recently gained significant attention for computationally representing and analyzing human language. It has wide applications in various fields such as summarization, email spam detection, machine translation, information extraction, medical, and question answering [9]. Source code analysis [10] is an interesting field in which application code, written in various programming languages, is analyzed to allow appropriate decisions at compile time. This technique has been actively adopted for code offloading, whether to run the code on CPU or any heterogeneous device, and detecting several vulnerabilities present in the code [11], [12], [13], [14], [15], [16], [17], [18]. However, opportunities to predict the best optimization sequence through source code analysis have yet to be addressed.

In this work, we have employed source code analysis for extracting the application features through NLP on a large number of codes, which can make the machine learn the actual code map and its associated metrics at compile time, thus saving the overall cost. NLP-based source code analysis comprises multiple phases like tokenization, atomization, vectorization, and machine learning. Previously, these techniques were applied directly to high-level language code like C, Java, and Python. With the advent of LLVM [19] which implements a rich and powerful Intermediate Representation (IR), the analysis can be done on LLVM IR. There are several advantages of analyzing an IR, mainly because it is a neutral language that is not specific to any high-level or architecture-specific language. Any high-level language can be converted into IR. Hence, the rules written once for IR are valid for all high-level and assembly languages. Besides, it is easier to analyze than high-level languages.

Several works in the literature analyze source codes for objectives like selecting heterogeneous devices and detecting vulnerability at compile time [11], [12], [13], [14], [15], [16], [17], [18]. These works have focused on machine learning and deep learning models to make the best predictions. However, they have neglected other stages of the NLP pipeline. In this regard, and to the best of our knowledge, this work is the first to study the impact of vectorization techniques on predicting the best optimization sequence at compile time.

Due to the inherent benefits of LLVM, the LLVM IR has been used for source code analysis. Our work used *tfidf*, *countvec*, *ngram*, and *nmf* vectorizers along with *random forest*, *bagging*, *linear regression*, *lasso*, *huber*, *bayesian ridge*, and *adaboost* regressors. For evaluating the performance, CBench benchmark suite [4], [20], [21], [22] has been used comprising 15 applications, where each application has more than 4k optimized codes, making the overall dataset size around 90k with 100 features. We have used the leave-one-out cross-validation technique. Therefore, an application under analysis is excluded from the training set, which ensures not predicting a seen application. Therefore, an application under analysis is excluded from the training set, which ensures not predicting a seen application.

It is observed that *tfidf* leads in nine applications, *ngram* shows the slightest error in three, *countvec* error rate is minimum in two, while *nmf* leads in one application only. On average, *countvec*, *ngram*, and *nmf* showed 34%, 34%, and 30% relative error, respectively. On the other hand, the error rate of *tfidf* is the lowest at 26%. Our discovered vectorization and regression combinations make decent estimations by predicting the best optimization sequence with an average performance drop of only 18%. For *security_blowfish_e*, *tfidf*, and *ridge discover*, the optimal optimization sequence presents a performance drop of only 0.5% compared to the best combination.

Our work is highly cost-effective as the prediction is automatically made at compile time, requiring only the application-specific features without using hardware features that require executing the application. Additionally, it involves minimal human intervention, significantly reducing costs and energy as the features are automatically extracted from the provided code. In recent years, specialized hardware has emerged, showing great reliance on compilers for higher performance, energy efficiency, and reduced development time. In this regard, the presented technique is generic enough and can easily be adopted by cloud, edge, IoT devices, or other big machines, as mentioned in [23], [24], and [25] for discovering the best performing and energy-efficient application codes through NLP-based analysis. For instance, our technique can be finetuned to find the best optimization for reducing the data size, which significantly facilitates memory-restricted devices. Also, energy-limited devices can demand such optimized codes to avoid certain specific operations. On the contrary, if the proposed technique is not used, then all the optimizations must be tested to discover the best one, which would highly exaggerate the time and resources, particularly in big machines. The main contributions of this paper can be listed as follows:

1) To the best of our knowledge, the first work that facilitates selecting the best optimization sequence at compile time via application features only.
2) To the best of our knowledge, the first work that studies the impact of various vectorization and regression techniques in predicting the best optimization sequence.
3) Saves from the hassle of collecting features as NLP automatically extracts them via the provided application code.

4) It employs a large set of application-specific features for training the model, which requires no prior execution, such as profiling.
5) Explores the power of NLP in predicting code performance by utilizing a large set of codes involving minimal overhead.

Furthermore, Section II discusses the background and motivation, followed by related work in Section III. The adopted methodology is presented in Section IV. Section V discusses the experimental setup and results. The conclusion is presented in Section VI.

II. BACKGROUND AND MOTIVATION

A. COMPILER TECHNOLOGY

The compiler is the tool responsible for translating the high-level language code into architecture-specific assembly and enables imperative optimizations for exploiting the hardware resources. The compilation life cycle consists of passing the source code through different stages: front-end, middle-end (optimizer), and back-end. The front-end stage emits the Intermediate Representation (IR) code, which is passed through the middle end to perform specific optimizations like inlining and unrolling. In the end, the back end generates the machine code [2], [3], [26], [27], [28].

Low Level Virtual Machine (LLVM) is an open-source compiler infrastructure containing reusable and modular compilation technologies. It provides extensive optimization opportunities and is widely adopted due to modularity in its design, hence offering ease of use and programming [3], [19]. The LLVM code is represented by Static Single Assignment (SSA)-based Intermediate Representation (IR), which provides low-level operations, type safety, portability, and flexibility. The LLVM IR resembles a universal IR, as all the phases of the LLVM compilation use this IR. Due to these benefits, LLVM has been selected as a framework to prototype this research work [1], [2], [3].

B. NLP AND TEXT MINING

Natural Language Processing (NLP) is a fusion of linguistics and artificial intelligence designed to make computers understand the words or statements written in human languages. It can analyze enormous amounts of data with the help of syntactic and semantic algorithms [29]. It has wide applications in various fields, such as summarization, email spam detection, machine translation, information extraction, medical, and question answering [9], [30].

Text Mining is the process of extracting information from text data by transforming text into numeric data that can be processed using machine learning algorithms. To build inputs of the machine learning model, text mining systems use numerous NLP techniques such as *tokenization*, *atomization*, and *vectorization*, as discussed below [31].

C. TOKENIZATION & ATOMIZATION

Given a defined document unit and character sequence, *tokenization* is responsible for chopping it into pieces. It is the act of breaking a stream of textual content up into words, symbols, terms, or other meaningful elements known as tokens. Certain characters, such as punctuation, are removed during this process [31], [32].

Atomization transforms code sequences by replacing the characters that compose a *token* with the integer identifier of the token in the dictionary. Afterwards, it performs a filtering procedure to select the most informative tokens [13].

D. VECTORIZATION

During text analysis, the unstructured documents must be transformed into structured data. It is necessary to create feature vectors in a specified feature space for applying machine learning. In this regard, a *vectorization* technique is employed, which stores previously prepared text in a specified vector space. Vectorization helps transform text corpus into a vector representation. Different vectorization techniques are discussed below [33], [34], [35].

1) COUNTVECTORIZER

It is a method that converts a document into vectors by counting the number of occurrences of each word in that document [36]

2) N-GRAM

It is a successful and accurate extension of a *bag of words* technique. It operates by turning the column of the adjacent words of length **n** to form meaningful context. With the small value of **n**, sufficient information can not be extracted. However, the system learns more accurately with the higher value of n [35].

3) TF-IDF

Tf-idf is one of the most popular vectorization techniques. Term Frequency (TF) counts the occurrence of a term (e.g., word) appearing in a document. The vector dimension corresponds to a word found at least one time. TF produces vectors based on the occurrence of words in a single document.

Some words appear in multiple documents, making a helpful relationship necessary among documents. TF-IDF (Term Frequency and Inverse Document Frequency) is a technique considering it. TF is expressed as the ratio of the number of times the word *wi* occurs in the document *dj*, and the total number of words in the document *dj* as represented by Equation 1 [13], [33], [34], [35].

$$TF(w_i, d_j) = \frac{Number\ of\ times\ w_i\ occurs\ in\ document\ d_j}{Total\ number\ of\ words\ in\ d_j} \quad (1)$$

FIGURE 1. Optimized code fragments of *network dijkstra*. Each optimization represents different code and execution time. Opt A has less instructions and time while opt C has more lines and takes maximum time.

The more frequent word has a higher *tf*, and the less frequent one has a lower *tf*. Meanwhile, the significance of words in the text corpus can be acquired by *idf*.

idf is obtained by the logarithmic ratio of the documents that contain the word, which is the fraction of the total number of documents (N) contained in the text corpus and the number of documents (ni) containing the required term as shown in Equation 2. D_c is the set of all documents.

$$IDF(w_i, D_c) = log\frac{N}{n_i} \tag{2}$$

The combined *tf-idf* can be represented by Equation 3. *IFIDF* forms a meaningful context by giving importance to rare and more frequent words together.

$$TF - IDF = TF * IDF \tag{3}$$

4) NON-NEGATIVE MATRIX FACTORIZATION

Non-negative matrix factorization (nmf) is a linear algebraic optimization algorithm that can extract meaningful information about topics without prior knowledge of the underlying meaning in the data. It decomposes high-dimensional vectors into a lower-dimensional representation, where the coefficients of matrices are constrained to be non-negative. Nmf emits W and H matrices, where W is the topics found, and H is the coefficients (weights) for those topics [37].

E. REGRESSION

Regression analysis is the widely used statistical technique for modeling and investigating the relationship between variables. It estimates the relationships between a dependent variable and one or more independent variables [38], [39]. Several regression techniques used in this paper are discussed below.

1) RANDOM FOREST

It is an ensemble learning technique, which can predict the outcome by merging multiple trees. A random subset of data points and predictors are selected to build the individual trees, and the overall prediction is done by taking the average of individual predictions [39], [40].

2) BAGGING

Bagging can be employed with classification and regression methods to reduce the variance associated with prediction, hence improving the prediction process. It works by drawing many bootstrap samples from the available data. A prediction technique is applied to each bootstrap sample, and the results are combined by averaging in regression to obtain the overall prediction, with the variance reduced due to the averaging [41].

3) LASSO

Least Absolute Shrinkage and Selection Operator (lasso) regression is a variable selection and shrinkage technique for regression models. LASSO regression identifies the variables and corresponding regression coefficients, leading to a model that minimizes the prediction error [42].

4) HUBER

Huber regression plays an important role in robust inference and estimation. It is an alternative approach based on a slightly modified loss function, called Huber loss [43].

5) BAYESIAN RIDGE

Bayesian regression employs Bayes' theorem to find the best estimate of the parameters of a linear model that describes the relationship between the independent and the dependent variables. It is a Bayesian method in which all regression coefficients are assumed to have a common variance [44].

6) ADABOOST

Adaboost or *Adaptive Boosting* is a meta-estimator that starts by fitting a regressor on the original dataset and then fits additional copies of the regressor on the same dataset but where the weights of instances are adjusted according to the error of the current prediction [45].

F. ANALYZING LLVM IR VIA NLP

Figure 1 shows LLVM IR fragments of the network *dijkstra* application. To illustrate, three optimization sequences, A, B, and C, are presented and the execution time for each sequence. Each optimization sequence generates different code for the same application, resulting in different characteristics. Depending on the optimization sequence, the generated code can contain, for instance, more memory instructions, more branch operations, or reduced memory footprint. Therefore, each optimization sequence can generate a code that demands more energy or less execution time, as illustrated in Figure 1.

By providing IR code and its corresponding metrics (e.g., time, energy), there is a high potential for a machine to learn the mapping of these parameters through NLP. The accuracy can be increased if the training is applied to a large set of IR code inputs. This way, the algorithm can learn all the different code and metrics mapping possibilities. The discussed approach is expected to be highly automatic as NLP itself takes the responsibility of extracting features from IR code instead of manually defining them like conventional approaches. This way, the system only needs the IR code, leaving the rest to the NLP's responsibility.

III. RELATED WORK

The conventional approaches [4], [5], [6], [8] relied on hardware features to predict an application's performance. These features were computed from application profiling, which requires execution. Collecting these dynamic features is quite a time-consuming and energy-cost task, especially for large datasets [3]. In contrast, the proposed work trains the machine with an application map and then tries to distinguish amongst applications without prior execution. For this, Natural Language Processing (NLP) is employed to differentiate among multiple applications by a large number of application codes as training samples.

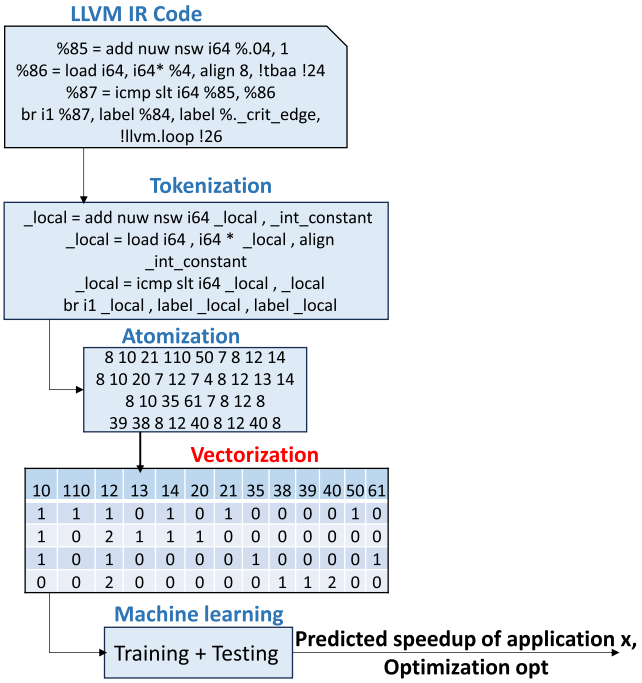

FIGURE 2. Flow chart of prediction mechanism.

In recent years, NLP has gained significant attention in analyzing language source code, as reported via multiple works in literature [12], [13], [14], [15]. Several works have employed NLP to decide heterogeneous devices favorable applications [11], [12], [13], [15], while others have detected code vulnerabilities [17], [18]. These works have focused on finding the best machine-learning model to analyze the given source code. Compared to these works, our proposal, to the best of our knowledge, studies the impact of different vectorization techniques on source code analysis for the first time. We have focused on a regression problem where the speedup of different optimized codes is predicted using NLP.

Our work is motivated by [13], which used a single vectorization technique, *tfidf*, with a neural network to decide whether to run the code on CPU or GPU. In our approach, we have analyzed the LLVM IR using the rules defined in this work. Moreover, we used multiple vectorization and regression techniques to select the best option since *tfidf* is not the best in all scenarios. Furthermore, the presented mechanism can predict the best application optimization sequence.

IV. METHODOLOGY

Before applying machine learning, the compiled LLVM-IR code (ref. Sec II-A) is required to pass through a preprocessing stage as shown in Figure 2. The preprocessing stage is comprised of *tokenization*, *atomization*, and *vectorization* (ref. Sec II-C & Sec II-D) [13]. The detailed working of each phase can be understood with the help of Algorithm 1.

The *tokenization* procedure is performed to break the text stream into tokens. We have performed the *tokenization*

similarly to [13], consisting of pre-tokenization and post-tokenization phases. The pre-tokenisation stage operates on each kernel line and removes empty lines, comments, and all lines outside the function body. It also simplifies complex data types and replaces constants with a placeholder which reduces the length of code fragment. Whereas, the post-tokenisation is applied directly to the tokens. During this phase, unnamed meta-data and attribute groups are removed along with replacing function and variable names with a placeholder. It also identifies special labels, integer constants, and global and local unnamed identifiers. As observed via Figure 2, LLVM IR code when passed through tokenization is greatly simplified such that all local, and global identifiers and constants are transformed and additionals symbols are removed which reduces the overall code length.

Atomization replaces the characters in a *token* with the integer identifier of the *token*. At this stage, we have transformed the tokens into unique numbers. For instance, all the local identifiers are replaced with the number 8, and the "=" characters are replaced with 10, as shown in Figure 2.

After this, *vectorization* is performed, which transforms the text corpus into a vector representation. This can be understood via Figure 2, where the bag of word *vectorization* is shown, which converts the unstructured text into a structured table with columns representing features and rows showing the frequency of those features. For instance, feature 12 has a count of 1, 2, 1, 2 as 12 appears single times in rows 1 and 3 and 2 times in rows 2 and 4 respectively.

The acquired feature vector is then passed through multiple regression techniques (ref. Sec II-E), where training is conducted with the feature vector and speedup. The training has been done using cross-validation where an application under analysis is excluded from the training set. The trained models are then tested keeping no evidence of test application in the training set.

V. EXPERIMENTAL SETUP AND RESULTS

This section details the evaluation setup and the results. Also, the results are presented and analyzed.

A. EVALUATION SETUP

All the implementation has been done using Python scripts. A single LLVM IR has been optimized using multiple optimization sequences to construct the dataset. The count of optimized codes for each application is mentioned in Table 1. The final dataset is comprised of total 90, 750 codes. These sequences are generated by pre-built Mitigates the Compiler Phase-ordering (MiCOMP) technique [5] with **5** optimization clusters and a maximum sequence length of **6**. The generated MiCOMP clusters for LLVM 12.0 are shown in Table 2. Each application has a different number of optimizations after suppressing the redundant codes using the approach present in [2]. The suppression of the redundant codes increases the proposed design's efficiency. There exists a total of 15 applications and a total 90750×100 size of the dataset, since vectorization only picks the best 100 features.

Algorithm 1 Procedure for Source Code Regression

```
Input: High level source code dataset
Output: Predicted metrics of given code
dataset ← new list;
foreach src in dataset do
    llvmir ← compile(src);
    llvmir_tokens ← new list;
end
foreach line in llvmir do
    /* Tokenization                          */
    line ← simplify_line(line);
    foreach token in line do
        token ← simplify_token(token);
        if token is valid then
            llvmir_tokens ← add token;
        end
    end
end
dataset ← add llvmir_tokens;
/* Atomization                               */
dataset ← atomise(dataset);
/* Vectorization                             */
dataset ← VECTORIZE(dataset);
/* Regression                                */
parameters ← model_training(dataset);
predicted_outcomes ←model_testing(parameters);
```

TABLE 1. cBench benchmark suite details [4], [21], [22].

cBench Programs	Description	Optimized Codes
automotive_bitcount	Bit counter	4731
automotive_susan_c	Smallest Univalue Segment Assimilating Nucleus Corner	8251
automotive_susan_e	Smallest Univalue Segment Assimilating Nucleus Edge	8251
automotive_susan_s	Smallest Univalue Segment Assimilating Nucleus S	8251
network_dijkstra	Dijkstra's algorithm	3323
network_patricia	Patricia Trie data structure	7623
office_stringsearch1	Boyer-Moore-Horspool pattern match	8311
security_blowfish_d	Symmetric-key block cipher Decoder	6449
security_blowfish_e	Symmetric-key block cipher Encoder	6449
security_rijndael_d	AES algorithm Rijndael Decoder	7144
security_rijndael_e	AES algorithm Rijndael Encoder	7144
security_sha	NIST Secure Hash Algorithm	4695
telecom_adpcm_c	Intel/dvi adpcm coder/decoder Coder	2997
telecom_adpcm_d	Intel/dvi adpcm coder/decoder Decoder	4328
telecom_CRC32	32 BIT ANSI X3.66 crc checksum files	2803
	Total	90, 750

Where each row represents a particular optimization for a specific application. For each sequence, the execution time is measured by using the Linux time command in the bash script. Further, experimental setup details are shown in Table 3.

For evaluating the proposed technique, embedded workloads belonging to automotive, security, office, and telecommunication categories from Collective Benchmark (cBench) programs [4], [20], [21], [22] are used as described in Table 1. The prediction accuracy is measured utilizing mean relative error (MRE), which evaluates the proximity of predicted outcomes (y_i') to the actual ones (y_i) as represented in

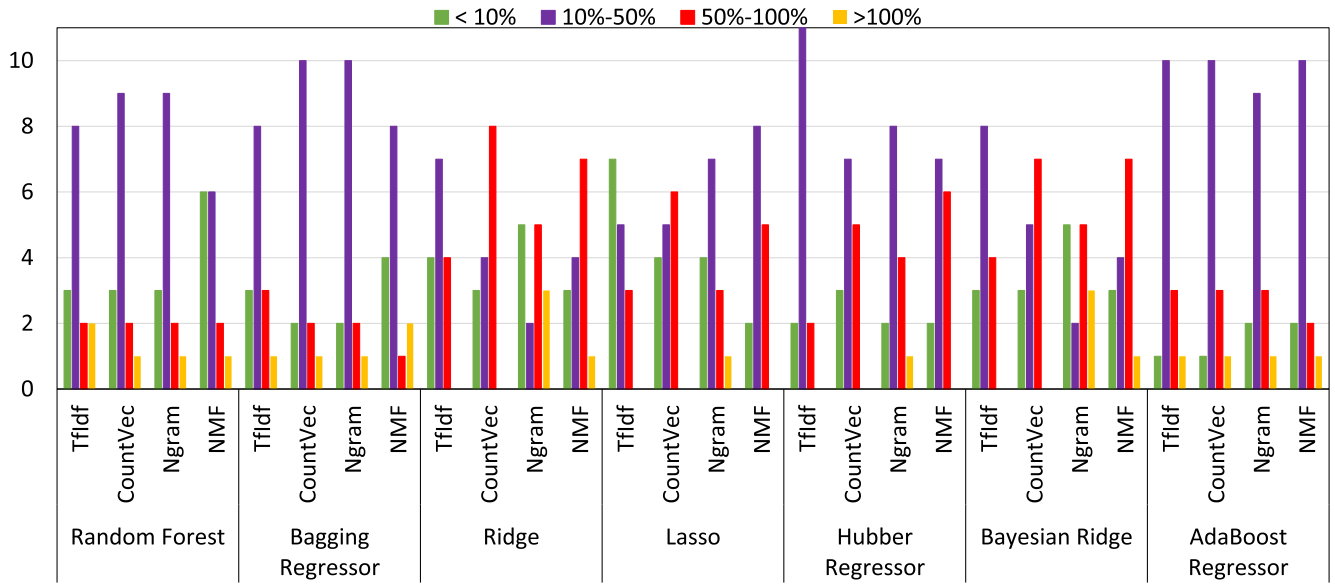

FIGURE 3. Error rates by different techniques.

TABLE 2. Compiler optimizations clusters using MiCOMP for LLVM-12.0 -O3 [5].

Sub-seq	Our derived Compiler Passes (LLVM-12.0)
A	-assumption-cache-tracker -profile-summary-info -annotation2metadata -forceattrs -callsite-splitting -ipsccp -called-value-propagation -mem2reg -lazy-block-freq -opt-remark-emitter -instcombine -basiccg -prune-eh -inline -openmpopt -function-attrs -lazy-value-info -jump-threading -correlated-propagation -libcalls-shrinkwrap -postdomtree -branch-prob -tailcallelim -loop-rotate -licm -loop-unswitch -loop-idiom -indvars -loop-deletion -memcpyopt -sccp -dse -barrier -elim-avail-extern -rpo-function-attrs -globaldce -float2int -loop-distribute -inject-tli-mappings -loop-vectorize -slp-vectorizer -vector-combine -transform-warning -alignment-from-assumptions -strip-dead-prototypes -constmerge -instsimplify -div-rem-pairs -annotation-remarks
B	-basic-aa -sroa -early-cse-memssa -aggressive-instcombine -pgo-memop-opt -phi-values -bdce -adce -loop-load-elim
C	-loops -lazy-branch-prob -memoryssa -block-freq -loop-unroll -memdep -demanded-bits -loop-accesses -loop-sink
D	-inferattrs -domtree -globalopt -deadargelim -simplifycfg -globals-aa -argpromotion -reassociate -lower-constant-intrinsics -cg-profile
E	-mldst-motion -gvn

TABLE 3. Experimental setup.

Parameters	Embedded Workloads
Total RAM	8 GB
Total Swap	2 GB
Disk Cache	1 GB
Model Name	Intel(R) Core(TM) i7-8550U CPU @ 1.80 GHz
Page Size	4 kB
Hard Disk	1TB SATA Harddisk
Operating System	Linux Ubuntu 18.04.4 LTS
L3 Cache	8192 KiB Associativity: 16-way Set-associative
L2 Cache	256 KiB Associativity: 4-way Set-associative
L1I, D cache	32 KiB Associativity: 8-way Set-associative
Compiler	LLVM-12.0
Benchmark	Ctuning cBench suite v1.1 [4], [20]–[22] dataset one
LLVM-12.0	*k*-means (5 clusters), Python-3.8.1 scikit-learn

Equation 4 [39].

$$Mean\ Relative\ Error\ (MRE) = \frac{1}{N}\sum_{i=1}^{N}\frac{|y_i' - y_i|}{y_i} \quad (4)$$

TABLE 4. Mean relative error in different vectorization techniques after optimization with random forest.

Application	TfIdf	CountVec	Ngram	Nmf
automotive_bitcount	39.93	33.04	41.29	31.58
automotive_susan_c	51.19	51.23	51.18	51.16
automotive_susan_e	8.23	8.28	8.27	8.30
automotive_susan_s	34.01	34.32	34.35	34.34
network_dijkstra	47.92	17.77	31.92	7.61
network_patricia	20.85	16.78	16.68	8.29
office_stringsearch1	49.47	48.00	47.62	40.45
security_blowfish_d	5.70	5.98	5.99	5.99
security_blowfish_e	6.01	6.20	6.20	6.19
security_rijndael_d	94.78	95.89	95.94	98.25
security_rijndael_e	107.23	109.13	109.34	110.21
security_sha	12.38	24.21	28.40	15.74
telecom_adpcm_c	21.43	16.26	20.02	15.03
telecom_adpcm_d	19.19	20.06	12.74	12.46
telecom_CRC32	115.37	35.76	10.16	9.65

Besides, the performance drop of the predicted sequence with respect to the best sequence is computed using Equation 5.

$$Performance\ Drop\ (PD) = \frac{|Exection\ Time_{best} - Exection\ Time_{predicted}|}{Exection\ Time_{best}} * 100 \quad (5)$$

B. RESULTS AND DISCUSSION

Results are analyzed in two parts. Firstly, the performance of various vectorization and regression models is compared, and the performance of predicting speedup is studied.

1) VECTORIZATION AND REGRESSION ANALYSIS

All **15** applications are experimented with four vectorization and eight regression techniques whose results are reported in Tables 4, 5, 6, 7, 8, 9, and 10. It can be observed via Table 4

TABLE 5. Mean relative error in different vectorization techniques after optimization with bagging regressor.

Application	TfIdf	CountVec	Ngram	Nmf
automotive_bitcount	40.27	31.10	43.01	33.20
automotive_susan_c	51.11	51.28	51.37	51.27
automotive_susan_e	9.99	10.06	10.02	9.95
automotive_susan_s	34.00	34.32	34.34	34.34
network_dijkstra	35.09	20.89	26.70	9.33
network_patricia	22.43	15.25	11.50	30.21
office_stringsearch1	48.39	48.05	49.28	41.06
security_blowfish_d	5.70	5.97	6.01	5.99
security_blowfish_e	6.02	6.21	6.21	6.20
security_rijndael_d	97.30	98.58	98.80	101.36
security_rijndael_e	109.73	111.26	111.56	111.77
security_sha	14.69	28.78	29.55	20.90
telecom_adpcm_c	21.09	17.02	19.43	14.50
telecom_adpcm_d	18.90	18.85	13.80	12.17
telecom_CRC32	89.62	33.83	15.06	12.20

TABLE 6. Mean relative error in different vectorization techniques after optimization with ridge.

Application	TfIdf	CountVec	Ngram	Nmf
automotive_bitcount	27.26	71.37	69.72	68.48
automotive_susan_c	50.86	51.07	51.27	51.35
automotive_susan_e	6.74	8.43	8.94	9.04
automotive_susan_s	34.34	34.80	34.68	34.66
network_dijkstra	16.55	85.67	8.49	17.30
network_patricia	11.96	18.13	129.59	352.64
office_stringsearch1	25.88	66.56	513602.81	55.67
security_blowfish_d	5.88	6.30	6.57	6.29
security_blowfish_e	5.78	6.03	7.13	6.82
security_rijndael_d	69.46	69.76	69.90	70.14
security_rijndael_e	84.23	84.98	85.32	85.22
security_sha	8.44	50.00	67.08	52.23
telecom_adpcm_c	17.29	13.74	12.49	14.03
telecom_adpcm_d	11.36	11.28	8.37	11.96
telecom_CRC32	61.53	84.60	109.93	62.47

TABLE 7. Mean relative error in different vectorization techniques after optimization with lasso.

Application	TfIdf	CountVec	Ngram	Nmf
automotive_bitcount	30.29	69.30	41.95	39.34
automotive_susan_c	50.18	51.11	51.23	51.78
automotive_susan_e	6.01	8.11	9.15	10.86
automotive_susan_s	34.36	34.75	34.68	34.54
network_dijkstra	4.95	32.13	17.84	18.40
network_patricia	5.38	13.58	49.09	51.89
office_stringsearch1	30.94	57.75	39.07	34.28
security_blowfish_d	5.32	6.34	7.02	5.19
security_blowfish_e	6.15	6.10	8.39	8.13
security_rijndael_d	69.84	69.83	69.92	70.75
security_rijndael_e	84.83	84.91	85.65	86.52
security_sha	9.72	6.91	32.56	45.90
telecom_adpcm_c	16.05	13.61	13.02	13.27
telecom_adpcm_d	7.45	12.90	9.34	12.10
telecom_CRC32	42.21	72.76	102.78	75.04

TABLE 8. Mean relative error in different vectorization techniques after optimization with huber regressor.

Application	TfIdf	CountVec	Ngram	Nmf
automotive_bitcount	22.71	37.67	31.02	44.61
automotive_susan_c	48.75	56.36	53.81	54.42
automotive_susan_e	11.40	18.25	16.40	19.21
automotive_susan_s	33.99	35.11	33.81	34.63
network_dijkstra	10.50	5.24	6.26	5.72
network_patricia	19.64	64.09	52.49	68.65
office_stringsearch1	33.31	55.53	198.26	95.88
security_blowfish_d	6.46	3.87	5.96	9.02
security_blowfish_e	6.36	7.07	10.18	13.15
security_rijndael_d	54.28	73.48	67.22	65.76
security_rijndael_e	74.25	82.39	85.77	85.37
security_sha	11.51	29.82	40.10	50.81
telecom_adpcm_c	18.01	26.68	25.02	20.44
telecom_adpcm_d	12.17	26.06	25.37	16.93
telecom_CRC32	46.70	25.08	28.37	22.22

TABLE 9. Mean relative error in different vectorization techniques after optimization with bayesian ridge.

Application	TfIdf	CountVec	Ngram	Nmf
automotive_bitcount	31.72	55.84	59.54	63.23
automotive_susan_c	50.94	51.06	51.25	51.34
automotive_susan_e	7.37	8.33	8.81	8.98
automotive_susan_s	34.11	34.79	34.67	34.66
network_dijkstra	42.81	69.22	9.19	16.72
network_patricia	18.83	17.84	120.91	213.03
office_stringsearch1	27.79	68.13	80959.29	51.32
security_blowfish_d	5.87	6.29	6.46	6.30
security_blowfish_e	6.11	6.04	6.98	6.82
security_rijndael_d	69.51	69.75	69.87	70.12
security_rijndael_e	84.64	84.96	85.32	85.23
security_sha	22.43	31.47	63.63	52.33
telecom_adpcm_c	15.52	13.51	12.46	14.02
telecom_adpcm_d	16.71	11.63	8.32	11.49
telecom_CRC32	55.88	76.91	106.41	64.15

TABLE 10. Mean relative error in different vectorization techniques after optimization with AdaBoost regressor.

Application	TfIdf	CountVec	Ngram	Nmf
automotive_bitcount	39.57	34.31	35.12	35.40
automotive_susan_c	50.19	53.02	50.28	49.88
automotive_susan_e	6.18	6.64	7.32	7.24
automotive_susan_s	34.03	35.72	36.83	36.61
network_dijkstra	37.51	26.53	54.39	47.38
network_patricia	42.87	18.90	5.20	13.71
office_stringsearch1	49.57	32.15	38	34.84
security_blowfish_d	20.04	18.41	17.69	17.32
security_blowfish_e	13.08	10.16	11.74	9.12
security_rijndael_d	96.18	91.44	93.72	94.93
security_rijndael_e	101.25	107.86	113.09	113.50
security_sha	15.18	18.74	28.85	24.16
telecom_adpcm_c	14.20	22.73	22	17.02
telecom_adpcm_d	26.24	15.12	18.94	10.25
telecom_CRC32	72.41	55.25	26.94	50.48

that when *random forest* regression is used, for the majority of applications, the *nmf* (highlighted in Table 4) shows the lowest error rate. *ngram* also depicts decent performance, while *tfidf* shows a larger error compared to others. *tfidf* shows the best outcome for *automotive_susan_s* application.

Error rates also depend on the application. For instance, the same models show the highest error for *security_rijndael_d* and *security_rijndael_e*, possibly because the training dataset does not cover these application characteristics. Compared to other regressors, *random forest* depicts the slightest error

TABLE 11. Application wise best regressor and vectorizer.

Techniques	TfIdf	CountVec	Ngram	Nmf
Random Forest	-	-	-	telecom_CRC32
Bagging	-	-	-	-
Ridge	office_stringsearch1, security_blowfish_e	-	-	-
Lasso	automotive_susan_e network_dijkstra telecom_adpcm_d	security_sha	-	-
Huber	automotive_bitcount, automotive_susan_c security_rijndael_d, security_rijndael_e	security_blowfish_d	automotive_susan_s	
Bayesian Ridge	-	-	telecom_adpcm_c	-
Adaboost	-		network_patricia	

for *telecom_CRC32* by using *nmf* vectorizer as highlighted in Table 4.

The vectorization performance is changed by adopting the *bagging* regression technique, as shown in Table 5. It can be seen that error rates slightly increase compared to *random forest*. *Bagging* technique performs moderately, with no application showing the least error rate compared to other regressors. Still, *security_rijndael_d* and *security_rijndael_e* show the highest error rates by all vectorizers.

The application behavior is changed with *ridge* regressor shown in Table 6 as *tfidf* depicts promising outcomes for most applications. However, for *security_rijndael_d* and *security_rijndael_e* the error rates are still high. *Ridge* regressor seems to favor *tfidf*, where the error rates are reduced drastically for *automotive_bitcount*, *automotive_susan_e*, *network_patricia*, *office_stringsearch1*, *security_blowfish_d*, *security_blowfish_e*, *security_sha*, and *telecom_CRC32* in comparison to other vectorization techniques. Compared to other regressors, the *ridge* regressor shows the lowest error with *tfidf* for *office_stringsearch1* and *security_blowfish_e* as highlighted in Table 6. Similarly, with the *lasso* regressor, shown in Table 7, all the vectorization techniques show reasonable performance with an overall drop in error rates compared to other regression methods. Overall, concerning other regressors *lasso* shows the lowest error rate for *automotive_susan_e*, *network_dijkstra*, and *telecom_adpcm_d* with *tfidf*. In contrast, the error rate is minimal in *security_sha* with *countvec* as highlighted by Table 7.

Overall, *huber* shows the lowest error rates for most applications, as pointed out in Table 8. Here, *tfidf* shows minimum error for *automotive_bitcount, automotive_susan_c*, *security_rijndael_d*, and *security_rijndael_e*, while *ngram* and *countvec* error rate is minimal for *automotive_susan_s* and *security_blowfish_d*, respectively. With *huber*, the error rates are significantly reduced for *security_rijndael_d* and *security_rijndael_e*. The error rate is slightly increased with the *bayesian ridge* regressor, as depicted in Table 9. Here *tfidf* shows the best performance for the majority of applications, while *ngram* gives the slightest error for *telecom_adpcm_c* in comparison to other regressors, as highlighted by Table 9.

Adaboost regressor also shows reasonable error rates for most applications, as shown in Table 10. *tfidf* performs slightly less than other vectorizers. As pointed out by Table 10, *adaboost* shows minimal error for *network_patricia* with *ngram* in comparison to other techniques.

The performance of the vectorizers and regressors is analyzed by considering the applications in Table 11 and Figure 3. It is possible to observe how *tfidf* gives the lowest relative error for nine applications, for four of them when coupled to *huber* regressor. In comparison, *lasso* and *ridge* dominate for the other three and two applications, respectively.

Ngram vectorizer dominates in three applications, while *countvec* and *nmf* are the best for two and one other applications, respectively. As observed by Figure 3, ittfidf shows a more significant number of less than 10% errors with the *lasso* regressor. *Nmf* also presents interesting results of less than 10% errors with *random forest*. However, most errors lie in the 10-20% range, and only some are greater than 100% in most techniques. The error rates also depend on the application; for some, it is lower, and for others, it is higher.

This behavior may be because some application characteristics are covered well by the training data, while others properties are not covered. For instance, *security_blowfish_d* shows lower errors in all vectorization and regression methods. Its lowest value is 3.87% with *countvec* and *huber* regressor. Its values are reasonable, with the majority of techniques, although high with *Adaboost* regressor. It can be seen how a vectorizer and regressor combination works best for an application, and the same combo acts poorly for other applications. This way, a single technique is not enough for all applications, and multiple combinations are required to minimize the error rate for different applications.

2) PREDICTION PERFORMANCE ANALYSIS

The performance difference between the actual and our predicted sequences is reported in Table 12. It can be seen for *automotive_susan_c* that the best sequence is **EDCAA** and our predicted sequence using *tfidf* and *huber* is **AAA** giving a significantly less drop of 1.61%. Similarly, the performance drop is less for other applications except

TABLE 12. Performance difference between actual and predicted sequences.

Application	Actual best combo	Predicted best combo	Perf drop
automotive_bitcount	BADAEC	ADCAEB	23.77%
automotive_susan_c	EDCAA	AAA	1.61%
automotive_susan_e	BDBADC	AAAA	22.67%
automotive_susan_s	AACAAA	BDAEAD	12.47%
network_dijkstra	BECABA	BDABEC	6.34%
network_patricia	DAEACC	A	8.94 %
office_stringsearch1	ABDDCB	BDACAC	1.28%
security_blowfish_d	AAAAAC	A	1.52%
security_blowfish_e	EBEADB	AC	0.536%
security_rijndael_d	ABAB	AEEC	25%
security_rijndael_e	EACBAB	BAAA	72.63%
security_sha	EBDCAC	D	44.60%
telecom_adpcm_c	DABAEA	EAACAC	39.28%
telecom_adpcm_d	DBABCD	DBABCE	8.83%
telecom_CRC32	AACD	ACEBCB	3.13%

for *security_rijndael_e* and *security_sha*, which might be because the training data do not cover the characteristics of these applications. Otherwise, our proposed technique can make reasonable predictions, showing an average drop of 18%. It only depends on the application features requiring no prior execution. This way, it can be known at compile time which optimizations should be adopted for a specific application, saving the before-mentioned costs and efforts on time and energy.

VI. CONCLUSION

The automatic prediction of best compiler optimizations is a challenging job. In this regard, Natural Language Processing (NLP) has done reasonably well in selecting the suitable optimization sequence by analyzing the code. The sequence predicted using the proposed approach shows a minimum drop of 0.5% in comparison to the actual best sequence, which is highly accurate due to the discovery of the most favorable vectorization and regression combination explored for an application. By coupling *tfidf* and *huber regressor*, the proposed technique can predict the best optimization with minimal error for most experimented applications. Moreover, efficiently utilizing a wide range of optimizations, the presented technique can generate code according to specific requirements, such as performance, energy, and memory size. Therefore, by adopting the NLP for analyzing the applications, the expected behaviors of code can be automatically predicted at the compile time with minimal effort, overhead, and overall costs.

ACKNOWLEDGMENT

The research findings presented in this article are solely the author(s)' responsibility.

REFERENCES

[1] H. Ahmed, M. F. Hyder, M. F. U. Haque, and P. C. Santos, "Exploring compiler optimization space for control flow obfuscation," *Comput. Secur.*, vol. 139, Apr. 2024, Art. no. 103704.

[2] H. Ahmed and M. A. Ismail, "REDUCER: Elimination of repetitive codes for accelerated iterative compilation," *Comput. Informat.*, vol. 40, no. 3, pp. 543–574, 2021.

[3] H. Ahmed and M. A. Ismail, "Toward a novel engine for compiler optimization space exploration of big data workloads," *Software, Pract. Exper.*, vol. 52, no. 5, pp. 1262–1293, May 2022.

[4] A. H. Ashouri, G. Mariani, G. Palermo, E. Park, J. Cavazos, and C. Silvano, "COBAYN: Compiler autotuning framework using Bayesian networks," *ACM Trans. Archit. Code Optim.*, vol. 13, no. 2, pp. 1–25, Jun. 2016.

[5] A. H. Ashouri, A. Bignoli, G. Palermo, C. Silvano, S. Kulkarni, and J. Cavazos, "MiCOMP: Mitigating the compiler phase-ordering problem using optimization sub-sequences and machine learning," *ACM Trans. Archit. Code Optim.*, vol. 14, no. 3, pp. 1–28, Sep. 2017.

[6] A. H. Ashouri, W. Killian, J. Cavazos, G. Palermo, and C. Silvano, "A survey on compiler autotuning using machine learning," *ACM Comput. Surveys*, vol. 51, no. 5, pp. 1–42, Sep. 2019.

[7] M. Zhu, D. Hao, and J. Chen, "Compiler autotuning through multiple phase learning," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 4, pp. 1–38, Apr. 2024.

[8] H. Liu, J. Xu, S. Chen, and T. Guo, "Compiler optimization parameter selection method based on ensemble learning," *Electronics*, vol. 11, no. 15, p. 2452, Aug. 2022.

[9] D. Khurana, A. Koli, K. Khatter, and S. Singh, "Natural language processing: State of the art, current trends and challenges," *Multimedia Tools Appl.*, vol. 82, no. 3, pp. 3713–3744, Jan. 2023.

[10] D. Binkley, "Source code analysis: A road map," *Future Softw. Eng.*, vol. 1, pp. 104–119, May 2007.

[11] E. Parisi, F. Barchi, A. Bartolini, and A. Acquaviva, "Making the most of scarce input data in deep learning-based source code classification for heterogeneous device mapping," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 41, no. 6, pp. 1636–1648, Jun. 2022.

[12] Y. Hakimi, R. Baghdadi, and Y. Challal, "A hybrid machine learning model for code optimization," *Int. J. Parallel Program.*, vol. 51, no. 6, pp. 309–331, Dec. 2023.

[13] F. Barchi, E. Parisi, G. Urgese, E. Ficarra, and A. Acquaviva, "Exploration of convolutional neural network models for source code classification," *Eng. Appl. Artif. Intell.*, vol. 97, Jan. 2021, Art. no. 104075.

[14] T. Sharma, M. Kechagia, S. Georgiou, R. Tiwari, I. Vats, H. Moazen, and F. Sarro, "A survey on machine learning techniques applied to source code," *J. Syst. Softw.*, vol. 209, Mar. 2024, Art. no. 111934.

[15] F. Barchi, E. Parisi, A. Bartolini, and A. Acquaviva, "Deep learning approaches to source code analysis for optimization of heterogeneous systems: Recent results, challenges and opportunities," *J. Low Power Electron. Appl.*, vol. 12, no. 3, p. 37, Jul. 2022.

[16] R. Mammadli, M. Selakovic, F. Wolf, and M. Pradel, "Learning to make compiler optimizations more effective," in *Proc. 5th ACM SIGPLAN Int. Symp. Mach. Program.*, Jun. 2021, pp. 9–20.

[17] F. Subhan, X. Wu, L. Bo, X. Sun, and M. Rahman, "A deep learning-based approach for software vulnerability detection using code metrics," *IET Softw.*, vol. 16, no. 5, pp. 516–526, Oct. 2022.

[18] Z. Bilgin, M. A. Ersoy, E. U. Soykan, E. Tomur, P. Çomak, and L. Karaçay, "Vulnerability prediction from source code using machine learning," *IEEE Access*, vol. 8, pp. 150672–150684, 2020.

[19] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. Int. Symp. Code Gener. Optim., CGO*, 2004, pp. 75–86.

[20] G. Fursin and O. Temam, "Collective optimization: A practical collaborative approach," *ACM Trans. Archit. Code Optim.*, vol. 7, no. 4, p. 20, 2010.

[21] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proc. 4th Annu. IEEE Int. Workshop Workload Characterization*, Jul. 2001, pp. 3–14.

[22] G. Fursin. (2010). *Cbench Benchmark*. [Online]. Available: https://sourceforge.net/projects/cbenchmark/files/cBench/V1.1/

[23] X. Han and Y. Zhang, "Decomposition-Coordination-Based voltage control for high photovoltaic-penetrated distribution networks under cloud-edge collaborative architecture," *Int. Trans. Electr. Energy Syst.*, vol. 2022, pp. 1–20, Jan. 2022.

[24] T. Yang, X. Han, H. Li, W. Li, and A. Y. Zomaya, "Parallel scientific power calculations in cloud data center based on decomposition-coordination directed acyclic graph," *IEEE Trans. Cloud Comput.*, vol. 11, no. 3, pp. 1–12, Jun. 2022.

[25] X. Han, Z. Li, and Y. Xu, "Quantum assisted stochastic economic dispatch for renewables rich power systems," 2024, *arXiv:2404.13073*.

[26] K. Hoste and L. Eeckhout, "Cole: Compiler optimization level exploration," in *Proc. 6th Annu. IEEE/ACM Int. Symp. Code Gener. Optim.*, Apr. 2008, pp. 165–174.

[27] K. Cooper and L. Torczon, *Engineering a Compiler*. Amsterdam, The Netherlands: Elsevier, 2011.
[28] A. V. Aho, S. L. Monica, and D. U. Jeffrey, *Compilers: Principles, Techniques and Tools*. Pearson Education, 2007.
[29] R. Ferreira-Mello, M. André, A. Pinheiro, E. Costa, and C. Romero, "Text mining in education," *Wiley Interdiscipl. Reviews, Data Mining Knowl. Discovery*, vol. 9, no. 6, p. e1332, 2019.
[30] J. Hirschberg and C. D. Manning, "Advances in natural language processing," *Science*, vol. 349, no. 6245, pp. 261–266, 2015.
[31] F. Zhang, H. Fleyeh, X. Wang, and M. Lu, "Construction site accident analysis using text mining and natural language processing techniques," *Autom. Construction*, vol. 99, pp. 238–248, Mar. 2019.
[32] S. Vijayarani and R. Janani, "Text mining: Open source tokenization tools—An analysis," *Adv. Comput. Intell., Int. J. (ACII)*, vol. 3, no. 1, pp. 37–47, Jan. 2016.
[33] U. Krzeszewska, A. Poniszewska-Marałda, and J. Ochelska-Mierzejewska, "Systematic comparison of vectorization methods in classification context," *Appl. Sci.*, vol. 12, no. 10, p. 5119, May 2022.
[34] P. Leelaprute and S. Amasaki, "A comparative study on vectorization methods for non-functional requirements classification," *Inf. Softw. Technol.*, vol. 150, Oct. 2022, Art. no. 106991.
[35] D. Rani, R. Kumar, and N. Chauhan, "Study and comparision of vectorization techniques used in text classification," in *Proc. 13th Int. Conf. Comput. Commun. Netw. Technol. (ICCCNT)*, Oct. 2022, pp. 1–6.
[36] R. Goyal, "Evaluation of rule-based, CountVectorizer, and Word2Vec machine learning models for tweet analysis to improve disaster relief," in *Proc. IEEE Global Humanitarian Technol. Conf. (GHTC)*, Oct. 2021, pp. 16–19.
[37] Zoya, S. Latif, F. Shafait, and R. Latif, "Analyzing LDA and NMF topic models for Urdu tweets via automatic labeling," *IEEE Access*, vol. 9, pp. 127531–127547, 2021.
[38] E. Alpaydin, *Introduction to Machine Learning*. Cambridge, MA, USA: MIT Press, 2020.
[39] H. Ahmed and M. A. Ismail, "Towards a novel framework for automatic big data detection," *IEEE Access*, vol. 8, pp. 186304–186322, 2020.
[40] A. Liaw and M. Wiener, "Classification and regression by randomforest," *R News*, vol. 2, no. 3, pp. 18–22, 2002.
[41] C. D. Sutton, "Classification and regression trees, bagging, and boosting," *Handbook Statist.*, vol. 24, pp. 303–329, May 2005.
[42] J. Ranstam and J. A. Cook, "Lasso regression," *J. Brit. Surgery*, vol. 105, no. 10, p. 1348, 2018.
[43] Y. Feng and Q. Wu, "A statistical learning assessment of Huber regression," *J. Approximation Theory*, vol. 273, Jan. 2022, Art. no. 105660.
[44] F. A. da Silva, A. P. Viana, C. C. G. Correa, E. A. Santos, J. A. V. S. de Oliveira, J. D. G. Andrade, R. M. Ribeiro, and L. S. Glória, "Bayesian ridge regression shows the best fit for SSR markers in psidium guajava among Bayesian models," *Sci. Rep.*, vol. 11, no. 1, p. 13639, Jul. 2021.
[45] D. P. Solomatine and D. L. Shrestha, "AdaBoost. RT: A boosting algorithm for regression problems," in *Proc. IEEE Int. Joint Conf. neural Netw.*, vol. 2, Jun. 2004, pp. 1163–1168.

HAMEEZA AHMED received the B.E., M.Engg., and Ph.D. degrees in computer and information systems from the NED University of Engineering and Technology, Pakistan, in 2012, 2015, and 2021, respectively. She is currently an Assistant Professor with the NED University of Engineering and Technology. Her research interests include big data computing, compiler optimizations, and machine learning.

MUHAMMAD FAHIM UL HAQUE received the bachelor's and master's degrees in telecommunications engineering from the NED University of Engineering and Technology, Karachi, Pakistan, in 2006 and 2009, respectively, and the Ph.D. degree in electrical engineering from Linköping University, Sweden, in 2017. He is currently an Assistant Professor with the Department of Telecommunications Engineering, NED University of Engineering and Technology. His current research interests include CMOS power amplifier, high speed digital circuits, all digital transmitter, RF-DACs, power efficient transmitter, software-defined radios, massive MIMO, wireless communication, and computer vision.

HASHIM RAZA KHAN received the B.E. degree in electrical engineering from the NED University of Engineering and Technology, Karachi, in 2002, the M.Sc. degree in communications engineering from RWTH Aachen, Germany, in 2006, and the Ph.D. degree in electronic engineering from the NED University of Engineering and Technology, in 2014. During the M.Sc. degree, he was with Agilent Technologies and Infineon Technologies, on frequency synthesizers design. From 2009 to 2011, he was a Visiting Researcher with Linköping University, Sweden, where he involved in the development of switching power amplifiers. He is currently an Associate Professor with the NED University of Engineering and Technology, where he is responsible for the Instrumentation Centre, RF Laboratory, Electronics Design Centre, and the Neurocomputation Laboratory, National Centre of Artificial Intelligence. He is involved as the principal investigator or the co-principal investigator in several grants and funding from industry, as well as local and international agencies. He has 19 journals and 17 conference papers to his credit. His research interests include circuits and system design for wide range of applications, including IC design, the IoT, AI, robotics, electric mobility, and power electronics.

GHALIB NADEEM received the B.E. degree in electronic engineering from Iqra University, Pakistan, in 2018, and the M.E. degree in electrical and computer engineering from NEDUET, Pakistan, in 2023. He was an Electronic Engineer with ORAIT, Dammam, Saudi Arabia, in 2019. He is currently a Senior Lecturer with the Faculty of Engineering Sciences and Technology, Iqra University. His significant contributions include spearheading several noteworthy projects within the High-Performance Research (HPR) Group. Notably, he played a pivotal role in the development of the groundbreaking EFG tiles, a revolutionary concept that harnesses piezoelectric floor tiles to generate electricity in Pakistan. He has a life time membership of Pakistan Engineering Council.

KAMRAN ARSHAD (Senior Member, IEEE) is currently holds the distinguished position of the Dean of the Research and Graduate Studies and a Professor of electrical engineering with Ajman University, United Arab Emirates. He has played a pivotal role in leading numerous locally and internationally funded research projects that encompass the fields of cognitive radio, LTE/LTE-advanced, 5G, optimization, and cognitive machine-to-machine communications. He has made significant contributions to several European and international large-scale research projects and has over 150 technical peer-reviewed papers published in esteemed journals and international conferences. He was a recipient of three best paper awards and one Best Research and Development Track Award and has chaired technical sessions in several leading international conferences.

KHALED ASSALEH (Senior Member, IEEE) received the B.Sc. degree in electrical engineering from The University of Jordan, in 1988, the master's degree in electronic engineering from Monmouth University, New Jersey, in 1990, and the Ph.D. degree in electrical engineering from Rutgers, The State University of New Jersey, in 1993. From 2002 to 2017, he was with American University of Sharjah (AUS), as a Professor of electrical engineering, and the Vice Provost of the Research and Graduate Studies. Prior to joining AUS, he had a nine-year research and development career in Telecom Industry in USA with Rutgers, Motorola, and Rockwell/Skyworks. He is currently the Vice Chancellor of the Academic Affairs and a Professor of electrical engineering with Ajman University. He holds 11 U.S. patents and has published over 150 articles in signal/image processing and pattern recognition and their applications. His research interests include bio-signal processing, biometrics, speech and image processing, and AI and machine learning. He has served on organization committees for several international conferences, including ICIP, ISSPA, ICCSPA, MECBME, and ISMA. He served as the guest editor for several special issues of journals.

PAULO CESAR SANTOS holds the degree in digital systems engineering from the State University of Rio Grande do Sul in 2011, and the master's degree in microelectronics and the Ph.D. degree from the Federal University of Rio Grande do Sul in 2014 and 2019, respectively. He worked on projects with themes focused on processing-in-memory, compilers, and processing and memory systems using 3D technology at the Federal University of Rio Grande do Sul. Working mainly on the following topics: multiprocessor system on chip, network on chip, SoC, network interface, and MPSoC.