Compilation Optimization Pass Selection Using Gate Graph Attention Neural Network for Reliability Improvement

JIANG WU1, JIANJUN XU1, XIANKAI MENG1, HAOYU ZHANG1, ZHUO ZHANG2, AND LONG LI2, (Member, IEEE)

1College of Computer, National University of Defense Technology, Changsha 410000, China
2Guangxi Key Laboratory of Trusted Software, Guilin University of Electronic Technology, Guilin 541000, China

Corresponding author: Zhuo Zhang (zz8477@126.com)

This work was supported in part by the China National Science Foundation under Grant 61872445, Grant 61472440, and Grant 61672529; and in part by the Guangxi Key Laboratory of Trusted Software under Grant kx202008.

ABSTRACT When dealing with different programs or applications, it is necessary to select the appropriate compilation optimization pass or subsequence for the program. Machine learning is widely used as an efficient technological means of solving this problem. However, the most important problem when using machine learning is the extraction of program features. Obtaining more semantic and syntax information and complex transitions among code segments from the source code are obviously necessary in this context, and is also an area that may have been neglected by previous work. Ensuring the integrity and effectiveness of program information is key to this problem. Moreover, when performing and improving the selection, the measurement indicators are often program performance, code size, etc.; there is limited research on program reliability in this context, which requires both the longest measurement time and the most complicated measurement methods. Accordingly, this paper establishes a combined program feature extraction model and proposes a graph-based compilation optimization pass selection model that learns heuristics for program reliability. This experiment was performed using the clang compilation framework. The alternative compilation optimization pass adopts the C language standard compilation optimization passes. Compared with traditional machine learning methods, our model improves the average accuracy by between 5% and 11% in the optimization pass selection for program reliability. Our experiments also demonstrate the strong scalability of our proposed model.

INDEX TERMS Compilation optimization, pass selection, extraction of program features, GGNN-based model, program reliability.

I. INTRODUCTION

Over the past few decades, compiler developers have designed and implemented a large number of compilation optimization options in response to related needs in various complex situations. In actual development, it is difficult for the standard compilation optimization pass provided by the compiler to adapt the program requirements needing to be compiled in complex scenarios. On one hand, programs to be compiled have different semantics and compilation goals, meaning that it is difficult to obtain the optimal optimization effect through direct use of the standard compilation optimization pass. If an inappropriate optimization pass is used, it may even have negative effects on program performance, etc. On the other hand, due to the continuous development of the hardware architecture, the compilation environment is becoming increasingly complex, and the compilation optimization pass should be adjusted accordingly. Therefore, the question of how to choose the best compilation optimization pass out of the many intricate optimization passes available to compile a given program has become a challenging scientific problem. The algorithms used in this field include primarily heuristic search algorithms and machine learning algorithms. The former use a heuristic method to
search the optimal compilation optimization pass in the compilation optimization option combination space. For example, the VISTA interactive compilation system [1] uses a combination of genetic algorithms and human-assisted guidance to search for optimal compilation optimization passes; moreover, the open-source framework “OpenTuner” [2] uses a variety of evolutionary algorithms, including genetic algorithms, to obtain a speedup of up to 2.8 times. For their part, Jantz et al. [3] select the optimal compilation optimization pass by utilizing genetic algorithms for the JIT compiler. There are also some other selection schemes based on certain multi-objective optimization algorithms; for example, Lokuciejewski et al. [4], [5] use SPEA2, NSGA-II and IBEA to select the compilation optimization pass for the program to be compiled that meets the target code execution speed, scale, and other goals.

However, while the heuristic search algorithm can generate efficient compilation optimization sequences, running the entire process is highly time-consuming. Gradually, researchers have therefore begun to use machine learning algorithms to select compilation optimization sequences. A large number of algorithms based on SVM and LR are in widespread use. Reference [6] used code runtime characteristics to characterize the program to be compiled in order to train the logistic regression model. Ashouri et al. [7] analyzed the dependencies between optimization options in the compiler’s LLVM, used program dynamic characteristics to train a Bayes network, then went on to use this model to predict the optimization options that should appear in the next stage until the prediction is completed. Moreover, the open-source compiler “Milepost GCC” [8] is a modularized, modified form of the GCC4.4 scalable compiler; this compiler supports static feature extraction of the program to be compiled, trains machine learning models, and then predicts the compilation effect of the compiled optimization sequence. There are a large number of machine learning algorithms that perform feature extraction (of both dynamic and static features) on programs. It is however difficult to extract program information completely and efficiently. Most existing works have attempted to transfer natural language methods and fail to capitalize on the unique opportunities offered by the code’s known semantics. For example, these works often do not consider the long-range dependencies induced by the use of the same variable or function in distant locations. Such models therefore miss the opportunity to capitalize on the rich and well-defined semantics of source code. Accordingly, a more effective way to ensure the integrity of the program information to the greatest possible extent is to construct a graph that represents the complete program information and training, in conjunction with a graph neural network.

In addition, from the perspective of compiling optimization goals, most research has focused on the execution speed of the target machine code [9], [10]. Statistics show that works using the acceleration ratio as the optimization target account for more than 80% of the research in this field. Another optimization goal that has attracted research interest is the size of the target code [11], [12]. This is a very important optimization goal, especially as regards the current widespread application of embedded programs, where reducing the storage space as much as possible can bring about significant benefits. However, there is limited research addressing the use of machine learning for compilation optimization orienting program reliability. As the scale of application software systems becomes larger and more complex, reliability becomes increasingly difficult to guarantee. However, the application itself has increasingly higher requirements regarding the reliability of system operation; in some key application areas, such as aviation and aerospace, these reliability requirements are particularly important.

To address the above problems, while taking advantage of the gate graph neural network (GGNN), we combine the GGNN model, program reliability analysis, and compilation optimization pass selection in the present research. First, in order to completely extract information from the program, we utilize data flow and function-call information on the basis of AST, which enables us to obtain semantic and syntax information from the source codes. Our work replaces the need for compile-time or static code features, merging general information and heuristic construction into a graph which is then sent to a graph neural network. An attention mechanism is then added to our GGNN; this mechanism is able to learn the weight of each individual node in the program graph, along with the aggregated representation of the entire graph, allowing it to guide the selection of compilation optimization passes. Our goal is to learn which standard compilation optimization pass can provide the highest reliability gain for a specific C code under the clang compilation framework. Through the use of verification tools, our model can attain an average accuracy improvement of 5% ∼ 11% compared to traditional machine learning algorithms without our extended GGNN. Our literature research further suggests that our team is the first to use a GGNN-based model to implement compilation optimization pass selection for program reliability.

II. RELATED WORK

The purpose of designing the graph model is to better represent and learn the program features. However, our ultimate goal is to find a reliable compilation optimization pass for the program. This idea was inspired by the increasing use of machine learning technology to address the compilation optimization selection or phase-ordering problem over recent years. These two issues are also the main issues we need to concern ourselves with.

Experimental results reveal that the use of any compilation optimization pass will directly affect the final optimization results, without considering the order of the compilation optimization sequence [13], [14]. This is the problem of compilation optimization selection: that is, whether a certain pass should be used. Ashouri et al. [15] used the Bayesian network to realize the selection of the best compilation optimization options for specific applications. In this work, he takes the acceleration ratio as the optimization goal; as noted above,
this is one of the most widely used goals for compilation optimization. Liu et al. [16] further proposed a machine learning-based prediction model named ALIC. Experimental results demonstrate that the use of this model greatly improves the accuracy of optimization parameter prediction.

Many other avenues of research focus not on learning models specifically, but rather on optimizing and improving the feature extraction process. The feature extractor designed by Li et al. [17] can extract dynamic features at specific stages of program execution, which are then used in combination with static features. Moreover, the quality of program features directly affects the quality of learning; thus, [18] designed a mechanism that pushes features in order to maximize the learning effect of the model to be learned.

Beginning from the two aspects of the learning model and feature extraction in an attempt to make improvements is a common research method. However, most of the compilation optimization goals used in the literature are traditional, such that another innovation of the present article is to depart from this framework. At present, and as discussed above, the most widely used compilation optimization goal is the acceleration ratio of the target code [19]–[21], with code size [22], [23] also being a relatively common optimization goal. Other optimization goals include CPU consumption [24], power efficiency [25], and resilience [26]. However, as mentioned in the introduction, there is still limited research on program reliability as an optimization goal. One of the main reasons for this is that the measurement of program reliability is simply a more complicated proposition. We have spent significant time in this article creating the dataset, and accordingly hope to achieve a balance between the reliability measurement effect and the time cost.

Many types of research also continue to focus on solving the phase-ordering [27] problem. The phase-ordering problem is relatively complicated; this is because it is necessary to address not only the order of the compilation optimization sequence, but also the interactions between different passes [28]. Moreover, the optimization search space increases exponentially as the number of candidate passes and the length of the optimization sequence also increase, making it difficult to reach the global optimum. More discussion on the compilation optimization sequence is therefore merited. However, as this article focuses on compilation optimization selection, we will not analyze the compilation optimization sequence here in depth.

III. PROGRAM AS GRAPH

Graph-based representation makes data dependencies explicit for each operation in a program. Data dependency graphs provide an explicit representation of the implicit definition-use relationships present in a particular piece of source code. The purpose of Allamanis’ [29] research is to predict variable names; accordingly, their work similarly adopted the method of AST-based integration to make the program graph information more complete. However, their work only supplements the data flow information while ignoring the function-call relations in the program. Moreover, because their work does not utilize the attention mechanism in GGNN, they are unable to calculate the different effects of individual nodes with their different weights in an effective manner. It is also difficult for their work to obtain the information of the integrated program graph and the dynamic distributed vector representations for edges in the program graph. Complete and effective feature extraction is a highly important part of the program; thus, we have improved the feature extraction construction in our work.

A. ABSTRACT SYNTAX TREE (AST)

As an intermediate representation of the source code for parsing and semantic analysis, AST [30] is a tree-structured data representation method that describes the syntax rules and execution order of the code, and can be obtained after the code is parsed using irrelevant context rules.

In an AST, the leaf nodes represent identifiers in the source code, while non-leaf nodes represent syntactic structures. AST also does not depend on the syntax of the source language; that is to say, context-free syntax is used in the syntax analysis stage. This is because, when writing the syntax, the syntax is often converted equivalently (such as by eliminating left recursion, backtracking, and the ambiguity); this will introduce additional components to the syntax analysis, adversely affecting the subsequent stages and possibly even making some stages chaotic. Thus, many compilers often need to construct a parse tree independently in order to establish a clear interface for the front and back ends. Therefore, AST, as an intermediate means of representing source code, can effectively retain the syntactic context information related to the programming language.

The example presented in Figure 1 briefly describes how AST may be generated from source code. Figure 1(a) contains the source code itself, while Figure 1(b) depicts the AST generated from the code fragment in Figure 1(a). The elements in Figure 1(a) are represented as nodes in Figure 1(b). The code fragment in Figure 1(a) contains integer declarations, variable declarations, function declarations, and operators. The AST therefore needs to establish a clear connection between these elements and its nodes. For example, the node named VarDecl in Figure 1(b) represents the variable declaration, the node named FunctionDecl represents the function declaration, the node named IntegerLite represents the integer declaration, and the node named BinaryOperator represents the operator. To reflect the corresponding relationship more clearly, the blue part in the node in Figure 1(b) is the element in Figure 1(a) that corresponds to the node.

B. FUNCTION CALL GRAPH (FCG)

The FCG [31] is used to characterize information related to the control flow in source code. Each node in a function call graph represents a function, while its edges represent the calling relationships between functions.

Understanding these calling relationships between functions is highly beneficial to understanding the hierarchical
structure of the program; thus, clarifying the function calling relationship represents a key part of program analysis. The structure of an FCG is illustrated in Figure 2. Here, each node in the figure represents a function in Figure 1. If a directed edge exists between node A and node B, this indicates that a calling relationship exists between the functions represented by these nodes. For example, the function ‘add’ calls the function ‘Foo’ in Figure 1; thus, there is a directed edge between the nodes ‘add’ and ‘Foo’ in Figure 2.

C. DATA FLOW GRAPH (DFG)

A DFG [32] explicitly represents the data logic of the two aspects of data transfer and data processing in the source code. Each node in a DFG represents an entity (such as variable declarations, operands, operators, structures, etc.), while the edges represent the data relationships existing between these entities. DFG can describe the data logic and program functions of the source code and is used to analyze the dynamic runtime data flow information of the program.

The use of DFG is very helpful in obtaining program information. In order to more clearly explain the DFG construction process, we generated a DFG (see Figure 3) based on the program fragment in Figure 1. Again, the nodes in this DFG are entities, while the direction of a directed edge between two nodes represents the data flow direction. For example, the edge from ‘Foo’ to \( x^1 \) in Figure 3 indicates that the value of \( x^1 \) comes from the function ‘Foo’. In addition, variables such as \( x \), \( y \), \( m \) are superscripted, which is a valuable design decision. This is because, in the absence of any special requirements, most programming languages will not deliberately use single static assignment (SSA), meaning that the same variable may have different values at different states of program execution; thus, it is helpful to use superscripts to distinguish between them. For example, the variable \( x \) in Figure 3 has two states, while the variable \( y \) has three states.

If we do not distinguish between different variable states, it becomes very difficult to describe the data flow without using SSA.

In Figure 3, we further distinguish between different types of edges using different colors. In more detail, we use five different types of edges in Figure 3. The name and corresponding color of the edges are “ComputeEdge” (purple), “ReturnEdge” (red), “OperandEdge” (black), “LastUseEdge” (dark blue) and “FormalEdge” (orange).

Evidently, “ComputeEdge” means that the variable is computed from other expressions (including other variables)
or functions. In Figure 3, \( y^2 \) comes from the expression 
\[ y^2 = x^2 + 1 \]
containing the variable \( x^2 \); thus, there is an edge from \( x^2 \) to \( y^2 \) in the figure. Moreover, as the variable \( x^1 \) is derived from the statement 
\[ \text{int} x^1 = \text{Foo}(m^2) \]
there are two edges pointing from \( m^2 \) and \( \text{Foo} \) to \( x^1 \). The “ReturnEdge” represents the return value of the function, so that its representation in the DFG is a directed edge from the variable name to the function name; this indicates that the return value of the function is the value stored by the variable name, such as the edge from \( y^2 \) to ‘add’ in Figure 3. The “OperandEdge” represents the relationship between the operator and the variable. In Figure 3, the value of the variable \( y^2 \) comes from the expression 
\[ y^2 = x^2 + 1 \]
containing ‘+’; thus, there is an “OperandEdge” from ‘+’ to \( y^2 \).

The “LastUseEdge” literally refers to the last use. The purpose of introducing this edge type is to describe the connection between different states of the same variable. Here, we will take the edge of \( y^3 \) to \( y^2 \) in Figure 3 as an example. The last occurrence of the variable \( y \) before \( y^3 \) is similar to the expression “\( y^2 = x^2 + 1 \)” containing ‘+’; thus, we can clearly describe the change process of the same variable in different states.

The “FormalEdge” describes the transfer of parameters during the function call. Take the edge from \( m^2 \) to \( m^1 \) in Figure 3 as an example. In the function ‘add’, ‘add’ is the calling function, ‘\( \text{Foo} \)’ is the called function, \( m^1 \) is the formal parameter of the caller function, and \( m^2 \) is the actual parameter of the called function. This edge therefore represents the relationship between the actual parameters of the called function and the formal parameters of the caller function. Moreover, taking the edge from \( m^1 \) to \( m^{\text{Foo}} \) in Figure 3 as a further example, the function ‘add’ calls the function ‘\( \text{Foo} \)’, \( m^1 \) is the formal parameter of the caller function ‘add’, and \( m^{\text{Foo}} \) is the formal parameter of the called function ‘\( \text{Foo} \)’ when it is declared; this represents the relationship between the formal parameters of the caller function and the formal parameters of the called function. Clearly, the main purpose of the “FormalEdge” is to describe the relationship between function calls. There is also no strict requirement for the specific parameter passing form; thus, it can also represent other general forms of parameter passing.

**D. COMBINATION GRAPH**

A comparison of the characteristics of AST, FCG, and DFG makes it clear that each different form of intermediate expression only describes the program source code from a certain perspective. For example, the AST only contains static information related to the grammatical structure, while the latter two forms are used to describe the runtime dynamic information related to the control flow and the data flow respectively. In particular, the angles of the latter two references are different: FCG starts with a coarse-grained function, while DFG starts with fine-grained variables, operators, and operands. As source code is a special type of executable text, both the static syntax information and dynamic runtime information are important; therefore, the fusion of the code information contained in the AST, FCG, and DFG helps reduce the information loss caused by the transformation of source code to intermediate expressions.

In addition, AST as a basic structure can facilitate more comprehensive analysis of the source code, while the information contained in the DFG and FCG is closely related to the reliability of the program. Among the methods for improving program reliability, data flow hardening and control flow hardening are the most commonly used; evidently, the information in the DFG represents the data flow information, while the function call information contained in the FCG is a very important part of the control flow. In control flow analysis, a program is often first divided into basic blocks, with most of the exit statements of these blocks being jump instructions that map to function calls in high-level languages. Accordingly, the information contained in the DFG and FCG is very helpful for analyzing the reliability of the program.

We have established a combination program analysis graph, co-AFD, which combines the characteristics of AST, FCG, and DFG. We have constructed the co-AFD graph in Figure 4 based on the source code in Figure 1. The complete co-AFD graph has a total of seven types of edges. We introduce the edges of FCG and DFG into the original AST structure. The solid arrow is the function call identifier, while the dashed curve arrow is the identification of the data stream. This combination significantly enriches the information in the program graph, thereby speeding up the spread of information in GGNN and improving the model training effectiveness.
IV. CONSTRUCTION OF GGANN MODEL

Obviously, we need to build a directed graph model to carry out our experiments. We consequently define a directed graph \( G = (V, E) \) here; \( V \) represents the set of nodes, the size of which is \(|V|\), while \( E \) represents a set of directed edges, the size of which is \(|E|\). The nodes in the node set are distinguished by number (e.g. node \( i \) and node \( j \)). A directed edge from node \( i \) to node \( j \) is represented by \( e_{ij} \), so that similar directed edges constitute a set of directed edges. We use the edge type set \( L_k = \{l_1, l_2, \ldots, l_k\} \) to represent the different types of edges in the graph. The connection relationship between the graph nodes is further represented by the adjacency matrix \( A \). There are two design schemes for the dimension of \( A \). The first of these is \( A \in \mathbb{R}^{(|V| \times 2)|V|} \); here, the directed edge \( e_{ij} \) in the figure is conceptualized as two different types of access edges, namely the outgoing edge of node \( i \) and the incoming edge of node \( j \). The second design is \( A \in \mathbb{R}^{|V| \times |V|} \), which only considers the directed edge \( e_{ij} \) as the incoming edge of node \( j \). The adjacency matrix in this paper employs the second scheme.

The element \( A_{ij} \) in the \( i \)th row and the \( j \)th column is a matrix of size \( d \times d \) (where \( d \) represents the node state vector dimension). \( A_{ij} \) is also referred to as the propagation matrix on edge \( e_{ij} \), while the information propagation rule from node \( i \) to node \( j \) is represented by \( A_{ij} \). To facilitate more intuitive description, we generated the DFG in Figure 5(b) according to the adjacency matrix in Figure 5(a), while the connection matrix in Figure 5(c) is generated according to the DFG in Figure 5(b). As we can see from the figure, the two rectangular-framed matrices are the propagation matrices corresponding to \( e_{3y} \) and \( e_{yz} \) respectively.

\[
\begin{align*}
\begin{array}{c|c|c|c|c|c|c|
\hline
i & y & z \\
\hline
3 & 0 & 0 & 1 & 0 & 0 & 0 \\
\hline
y & 0 & 0 & 0 & 0 & 1 & 0 \\
\hline
z & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
\end{array}
\end{align*}
\]

**FIGURE 5.** The adjacency matrix derived from the code fragment and DFG above.

In the code fragment presented in Figure 5(a), our concern is whether the number 3 can be passed to the variable \( z \), as shown in Figure 5(b). To this end, nodes 3 and \( z \) can be regarded as the source node and the target node respectively. Their feature vectors are initialized as \( h_3^0 = [0, 1] \) and \( h_z^0 = [0, 1] \) (the first dimension of the two-dimensional vector is 1, meaning that 3 can reach the node), while the feature vector table of node \( y \) is initialized as \( h_y^0 = [0, 0] \). The propagation matrix \( A_{ij} \) determines how the information of each dimension of node \( i \) is propagated to the various dimensions of node \( j \). Here, ‘0’ represents no propagation and ‘1’ represents complete propagation. For example, \( A_{3y} \) in Figure 5(c) indicates that node 3 passes only the information of its first dimension to the first dimension of node \( y \). Accordingly, the result of multiplying vector \( h_3 \) and \( A_{3y} \) is still \( h_y = [1, 0] \), which indicates that the data has not been passed to the target node \( z \). However, \( A_{yz} \) in Figure 5(c) indicates that the information of the first dimension of node \( y \) is to be transferred to the first dimension of node \( z \). Therefore, the result of multiplying vector \( h_y \) and \( A_{yz} \) is \( h_z = [1, 0] \), which indicates that the data can reach the target node \( z \).

In GGNN [33], the propagation of node information through different types of edges is achieved through different types of multilayer perceptron, while the propagation matrix on the edges is represented by the trainable multilayer perceptron weights \( W_e \in \mathbb{R}^{d \times d} \). It should be noted here that the connection matrix shown in Figure 5(c) represents only the weight of the multi-layer perceptron after the graph model has converged on the reachability task. The GGNN model has a node state information propagation process that occurs over \( t \) iterative rounds, which is conducted as follows.

First, the state information of node \( i \) is initialized to a vector \( h_i^{(1)} \in \mathbb{R}^d \). During the \( t \)-th round of iteration, each central node \( i \) gathers all neighbor node information to obtain the node interaction context \( m_i^{(t)} \in \mathbb{R}^d \), as shown in (1) (here, \( N_i \) represents the set of \( i \)'s neighbor nodes). The node \( i \) updates its own state information \( h_i^{(t)} \) after \( t \) rounds in response to the current interaction context. Moreover, the GRU unit is used in GGNN in a different way to GNN. More specifically, the GRU unit considers the relationship between pieces of node state information in different update rounds; that is, when the node updates during the \( t \)-th round, there is a time series relationship between the node hidden layer vector expression \( h_i^{(t-1)} \) and the state information \( h_i^{(t-1)} \) of the previous round, as shown in (2) below. By contrast, GNN only uses edges as a means of propagation, but does not distinguish between the functions of different edges; moreover, GNN does not set independent learnable parameters for edges, meaning that some characteristics of edges cannot be learned through this model.

\[
\begin{align*}
\begin{align*}
m_i^{(t)} &= \sum_{j \in N_i} A_{ij} \cdot h_j^{(t-1)}, \quad \text{(1)} \\
h_i^{(t)} &= \text{GRU} (h_i^{(t-1)}, m_i^{(t)}). \quad \text{(2)}
\end{align*}
\end{align*}
\]

During the graph model’s information propagation, \( m_i^{(t)} \) is the interaction context of node \( i \) throughout the whole graph. Regardless of whether GNN or GGNN is being utilized, \( m_i^{(t)} \) is obtained by directly accumulating the product of the feature information \( h_i^{(t-1)} \) of the neighbor node \( j \) and the propagation matrix \( A_{ij} \) on the edge \( e_{ij} \). Moreover, different nodes have different properties in the graph topology. In the GNN and
GGNN models, the topological properties of the nodes are expressed directly as hidden nodes. Based on this, in the substructure composed of the central node $i$ and its neighbor nodes $N_i$, our model abandons the method of directly accumulating the product of $h^{(l-1)}_{ij}$ and $A_{ij}$ to calculate $m^{(l)}_{ij}$. Instead, we expect that the model will automatically learn how to calculate $m^{(l)}_{ij}$ and that the central node can therefore pay more attention to those neighbor nodes with important topological information; this is relevant because these neighbor nodes determine the interaction context of the node $i$ on the graph to a greater extent.

Accordingly, we extend GGNN to the gate graph attention neural network (GGANN). We assign different weights to each neighbor node in order to characterize its importance to the central node $\alpha_{ij}$, with function mapping performed through the neural network $a$: $\mathbb{R}^d \times \mathbb{R}^d \rightarrow \mathbb{R}$, where $a$ calculates the correlation coefficient between the central node $i$ and its neighbor $j$ and then uses the softmax function to normalize the correlation coefficients of all neighboring nodes, as shown in (3). The weight parameter of the neural network $a$ is related only to the round of information propagation.

$$\alpha_{ij} = \text{softmax}(a(h^{(l-1)}_i, h^{(l-1)}_j)).$$

(3)

In the same round of information propagation, $a$ is shared by all nodes. However, different propagation rounds have different parameters for $a$. The uncertainty regarding the number of neighbor nodes $j$ possessed by a node $i$ results in a variable number of $\alpha_{ij}$, and it is not possible to directly implement the softmax function provided by the Tensorflow framework. This paper implements softmax to adapt to the changes in the number of neighbor nodes:

$$\alpha_{ij} = \frac{e^{a_{ij}}}{\sum_{k \in N_i} e^{a_{ik}}}.$$

(4)

Accordingly, the interaction context $m^{(l)}_i$ of the node in of our expanded GGNN is shown in (6):

$$m^{(l)}_i = \sum_{j \in N_i} \alpha_{ij} \cdot A_{ij} \cdot h_j.$$

(6)

Because the program compilation optimization pass selection process is intended to analyze the program according to the embedded expression of the program algorithm graph co-AFD, after the final graph node embedding vector expression $h_i^{(T)}$ is obtained, the embedding vector $h_G$ of the entire co-AFD graph needs to be calculated. This paper accordingly proposes a node vector probability fusion method designed to generate a graph embedding vector from the node embedding vector. As shown in (7), $f(h_i^{(T)}, h_j^{(1)})$ is a fully-connected neural network, which learns the probability of node $i$ being fused based on the node attributes $h_i^{(1)}$ and topology information $h_j^{(T)}$. The activation function in $f$ uses the sigmoid function, whose final output is a value of $[0, 1]$. Moreover, $g$ is also implemented using a fully connected layer neural network, which in turn uses the $\text{tanh}$ function to activate the output. $h_G$ is calculated in a similar way to (6).

Finally, the program compilation optimization pass selection $l_G$ is derived from the softmax function as follows:

$$h_G = \sum_{i \in V} f(h_i^{(T)}, h_j^{(1)}) \cdot g(h_i^{(T)}).$$

(7)

$$l_G = \text{softmax}(h_G).$$

(8)

In GGANN, the computation of the $\alpha_{ij}$ of the central node and its neighbor nodes can be parallelized in a computationally efficient manner. Moreover, this approach implements a calculation method whereby the model automatically learns the interaction context, without having to consider changes in the number of neighbor nodes. If neural networks are used to learn to calculate the interaction context, moreover, it will be necessary to deal with the problem of the neural network weight dimensions’ inability to be unified due to the inconsistent number of neighbors in each node. This is also the main reason we opt to use GGANN (as shown in Figure 6).

![FIGURE 6. The GGANN architecture. Our work involves every step in this figure.](image)

V. EXPERIMENT

In order to evaluate the effect of our improved GGANN model and program graph co-AFD, we designed a number of experiments to verify the performance of our proposed solution. The experimental results are displayed below in the form of figures and tables. During these experiments, we have not only verified the accuracy of the compilation optimization pass selection, but have also conducted an in-depth analysis of the effectiveness of the program graph, the convergence speed of different combinations, and the model’s learning ability.

We employ the method proposed in [34] to divide the optimization pass for compilation. First, we divide all compilation optimization channels into four major categories: namely, code modification, code motion, code elimination, and loop-related optimizations. The number of passes (or subsequence) contained in these four categories is 35, 6, 20, and 14 respectively. Based on the classification of these compilation optimization passes, we set up two sets of experiments. In the first set, we verified the accuracy of our GGANN model and co-AFD graph for the four major categories. In the second set, we verified the accuracy of the selection of individual compilation optimization pass in each category. As compilation optimization passes belonging to the same
category will deal with the code in similar ways, this second group of experiments can comprehensively evaluate both the robustness and effectiveness of the proposed GGANN model and co-AFD graph.

We selected 6,000 complete programs written in C from 862 projects on GITHUB and they are suited to reliability evaluation. As these 862 projects all have different functions, the possibility that there are too many similarities in the testing set can be ruled out. We measure the reliability of different programs before and after using the compilation optimization pass. For a particular program, the pass resulting in the most significant improvement in reliability is used as the label of the program. Accordingly, the final dataset contains 6000 pieces of C source code and their corresponding most reliable compilation optimization pass. The generated dataset is subsequently divided into a training set, validation set and testing set with a ratio of 6:1:1. We employ the widely used open-source compilation framework *clang*. We first extract the AST from the C language source code, then add data-flow information and function-call information on the basis of the AST in order to obtain the complete co-AFD program graph.

Our reliability measurement is for intermediate code: that is, code obtained after the source code has been compiled and optimized. Accordingly, it is possible to investigate whether the compilation optimization pass has had an impact. Moreover, the intermediate code following clang compilation is extremely structural sound and can thus better reflect the code characteristics. The reliability evaluation of our code is further conducted according to the theoretical basis outlined in [35]. Program reliability is usually measured using mean work to failure (MWTF) as the indicator. This indicator is calculated as shown in (9):

\[
MWTF = \frac{1}{\lambda \times SDC_{rate} \times T_{exec}}. \tag{9}
\]

Here, \( SDC_{rate} \) refers to the probability that silent data corruption (SDC) has occurred during the execution of the program; we can obtain this using the analysis tool Trident [36]. \( T_{exec} \) refers to the program execution time. Moreover, \( \lambda \) denotes the so-called soft error rate of the program, which refers to the probability of soft errors occurring when the electronic device is irradiated by high-energy particles. Since our programs do not take the effects of irradiation into account, \( \lambda \) remains unchanged. We thus measure program reliability using MWTF by measuring \( SDC_{rate} \) and \( T_{exec} \), which is a recognized measure of program reliability in the field.

**A. EXPERIMENTAL CONFIGURATION**

The optimization algorithm used for model training is the SGD of the ADAM optimizer [37]. The loss function used is cross-entropy loss. Moreover, the weight parameter initialization uses the Glorot and Bengio [38] initialization method; here, we set the batch size to 10000 and the number of epochs to 3000. To avoid overfitting, we apply dropout [39] to the input of each layer with probability \( \rho = 0.6 \), and the L2 regularization term with the initial value \( \lambda = 0.0005 \) is adopted. We further use a linear learning-rate decay to adjust the learning rate from the initial learning rate 1 to the final learning rate \( l \times F \), with an attenuation coefficient \( F \) in the range \([0, 1]\).

In our experiment, the information propagation layer (information iteration round) is set to 4 layers, while the number of neurons in each propagation layer—that is, the propagation matrix vector dimension \( d \)—is a hyperparameter. The choice of this hyperparameter is primarily based on the speed of model convergence and the model’s loss value. We determined empirically that when the hidden layer vector dimension \( d \) is 270, the model’s convergence loss value is relatively small, while its training speed remains relatively fast; therefore, we set the hidden layer vector dimension \( d \) to 270 in subsequent experiments. More details regarding the determination of the \( d \) value are presented in Figure 7. The same dataset was also used during the experiment. We have carefully adjusted the parameters of each model to ensure an optimal effect during testing and to guarantee fairness. All hyperparameters were verified on the verification set. The testing set is used to test the generalization performance of the model. Notably, if the testing set was to be used to select the hyperparameters, the model would approximate the

![Figure 7](image-url)
testing set very well, with the result that the existence of the testing set would lose its meaning; therefore, we set up the verification set to complete the parameter tuning.

B. EXPERIMENTAL RESULTS

In our experiments, we examine the performance of our GGANN model and co-AFD graph in the compilation optimization pass selection context. Our goal is for our model to obtain the corresponding compilation optimization pass of different programs by learning. As noted above, each program in the dataset is assigned a label denoting the corresponding most reliable compilation optimization pass. In line with our experimental framework, we divided the experiment into two sets: namely, the accuracy of major categories (AMC) and accuracy in each category (AEC).

Moreover, to facilitate better verification of our conclusions, we chose two models as controls, namely GGNN and the tree-based convolution neural network (TBCNN) [40]. The reason for choosing GGNN is obvious: since GGANN is an improvement on GGNN, we need to assess the performance of GGNN in order to understand the magnitude of this improvement. For its part, TBCNN is a classic model that is used widely for program classification, indicating that it has a good effect in program processing and thereby justifying its selection as a comparison method. In the first set of experiments, we verified the AEC results; the test results are presented in Table 1. It should be noted here that TBCNN is able to use AST but not co-AFD; as it was originally designed for AST only, adding other information to the tree structure is not suitable for TBCNN.

As shown in Table 1, we provide statistics for maximum, minimum, and average AMC, and also distinguish between different program graphs. From the data, we can conclude that the graph neural network achieves higher accuracy than TBCNN when AST is used as the program graph at the same time. When using AST and co-AFD, GGANN achieves better results than GGNN and TBCNN. Firstly, this shows that graph neural networks are more effective for the problems we are investigating; secondly, the superiority of GGANN relative to GGNN is also evident, since a significant improvement in accuracy has been achieved.

In order to further verify the effectiveness of our model in the selection of similar compilation optimization passes, we next conducted a second set of experiments. We verified the performance of our model for each specific pass in the same category. The results of this experimental set are presented in Table 2. From the data, it can be concluded that when faced with more similar pass selection, TBCNN yields a significantly lower accuracy than the graph neural network. Moreover, TBCNN’s accuracy when solving similar problems is also greatly reduced relative to the first set of experiments (by up to 9%). The difference between the accuracy of GGANN and GGNN when solving similar problems is 3.5% in this set of experiments (compared to 1.1% in the first set).

It can further be concluded from the experimental results that the difference in the accuracy of GGANN across different experimental groups is relatively small. Accordingly, these results support the following conclusions: the graph neural network achieves better performance on similar problems, indicating that its robustness is stronger, while the performance of GGANN is superior to that of GGNN, indicating that our improvement has achieved its purpose. The use of the attention mechanism in GGANN can thereby be deemed a success.

At the same time, the average running time of GGANN + co-AFD is very close to that of GGNN + co-AFD, while the running times of GGANN + AST and TBCNN + AST are also very similar, such that the differences between these two groups are almost negligible. Moreover, the running time of GGANN + co-AFD differs from that of TBCNN + AST by no more than an order of magnitude, which is completely within the range of acceptability and a small price to pay for the concomitant increase in accuracy and program reliability.

C. RESULTS ANALYSIS

To obtain more convincing results, it is necessary to conduct statistical analysis on the data through hypothesis testing. We opt to use the Wilcoxon-Signed-Rank Test [41] to achieve this goal. The Wilcoxon-Signed-Rank Test is a non-parametric statistical hypothesis test for determining the differences between pairs of measurements.

Table 3 presents the statistical results for this relationship (TBCNN + AST vs. GGANN + AST) at a significance level of 0.05. We use the accuracy in four categories as the measurements. Let us take TBCNN + AST vs. GGANN + AST in Table 3 as an example. Here, the p values for the 2-tailed, 1-tailed (right) and 1-tailed (left) tests are 6.79E-02, 5.02E-02, and 9.78E-01 respectively. This indicates that the

| TABLE 1. Accuracy of major categories. |
|---------------------------------------|
|                                      |
| TBCNN      | GGNN      | GGANN      |
| Accuracy   | AST       | Minimum    | 0.940 | 0.963 | 0.971 |
|           | Maximum   | 0.925      | 0.935 | 0.945 |
| co-AFD     | Average   | 0.951      | 0.976 | 0.981 |
|           | Minimum   | -          | 0.967 | 0.978 |
|           | Maximum   | -          | 0.944 | 0.970 |
|           | Maximum   | -          | 0.979 | 0.983 |

| TABLE 2. Accuracy in each category. |
|-------------------------------------|
|                                      |
| Accuracy                           |
|-------------------------------------|
| Average                            |
| GGANN                               |
| 0.938                               |
| Minimum                             |
| GGNN                                |
| 0.903                               |
| Maximum                             |
| TBCNN                               |
| 0.857                               |

VOLUME 8, 2020
TABLE 3. Hypothesis testing on the data in Table 1. Our hypothesis test uses all the original data, not just sampling data (such as the maximum, minimum and average values).

| Comparison       | 2-tailed | 1-tailed (right) | 1-tailed (left) | Conclusion       |
|------------------|----------|------------------|-----------------|------------------|
| TBCNN + AST      |          |                  |                 |                  |
| vs. GGANN + AST  | 6.79E-02 | 5.02E-02         | 9.78E-01        | GGANN better     |
| GGNN + co-AFD    |          |                  |                 |                  |
| vs. TBCNN + AST  | 6.79E-02 | 5.02E-02         | 9.78E-01        | GGANN better     |
| GGANN + co-AFD   |          |                  |                 |                  |

TABLE 4. Hypothesis testing on the data in Table 2. Similarly, the data used here is raw rather than sampled data.

| Comparison       | 2-tailed | 1-tailed (right) | 1-tailed (left) | Conclusion       |
|------------------|----------|------------------|-----------------|------------------|
| GGANN + AST      | 6.56E-02 | 9.79E-01         | 4.88E-02        | GGANN better     |
| vs. TBCNN + AST  |          |                  |                 |                  |
| GGANN + co-AFD   | 6.56E-02 | 9.79E-01         | 4.88E-02        | GGANN better     |

The accuracy score of TBCNN + AST is significantly lower than that of GGANN + AST. We can therefore obtain the following, more accurate conclusion: that is, GGANN + AST performs better than TBCNN + AST in terms of accuracy. In the same case of using AST, our model GGANN achieves superior results to TBCNN; moreover, in the same case of using co-AFD, our model GGANN achieves better results than GGNN. The hypothesis test in Table 4 reveals that our GGANN model also achieves better results when selecting similar passes.

However, the experiments conducted thus far do not allow us to intuitively determine whether our co-AFD design has a good effect. While it has been shown that using the GGANN and GGNN models with the co-AFD graph results in improved accuracy relative to the combination of TBCNN and AST, this is not in itself sufficient evidence to warrant the conclusion that co-AFD plays a role in improving the experimental effect. Moreover, when using the graph model (GGNN or GGANN), the improvement associated with the use of co-AFD compared to AST is not very prominent. Due to the small magnitude of this difference, we cannot confidently assert that our co-AFD is better.

We therefore designed experiments for co-AFD to evaluate whether the data flow edge and function call graph edge are useful. Specifically, we remove each one of the 7 types of edges in turn, then use the GGANN model to learn the co-AST graph in order to implement the optimization pass selection accuracy. We here use AEC as the accuracy metric, since the accuracy of selecting similar passes can better reflect the value of the graph. The experimental results are visualized in Table 5; an underline indicates that the co-AFD graph with this type of edge has been removed. Due to the large number of dataset programming rules involved, we present six representative experimental data points in the table. The method used to calculate the impact for the program is the difference between the accuracy of the pass selection of the program before and after a certain kind of edge is deleted.

TABLE 5. Impact of the deletion of certain edge types on the program. The minus sign means that the accuracy decreases after a specific type of edge is deleted, while the absence of the minus sign indicates that the accuracy has increased.

| Different Edge | Average Impact for Different Programs |
|---------------|--------------------------------------|
| P1            | P2         | P3         | P4         | P5         | P6         |
| AST           | -0.99      | -0.002     | -0.009     | -0.95      | -0.006     | -0.009     |
| Operand       | -0.12      | -0.007     | -0.007     | -0.92      | -0.002     | -0.005     |
| LastUse       | -0.29      | -0.007     | -0.007     | -0.12      | -0.002     | -0.005     |
| Compute       | -0.92      | -0.007     | -0.007     | -0.92      | -0.002     | -0.005     |
| Return        | -0.94      | 0.007      | -0.007     | -0.92      | -0.002     | -0.009     |
| Formal        | -0.11      | -0.008     | -0.008     | -0.05      | -0.002     | -0.005     |
| Call          | -0.92      | -0.007     | 0.007      | -0.92      | -0.003     | -0.005     |

In most of the tests we conducted, deleting a specific type of edge did not substantially affect the results; as can be seen from Table 5, the effect on most programs is very weak. For certain programs, deleting certain kinds of edges actually improved the accuracy; of course, this is an extremely rare case. It is possible that information redundancy exists in the seven types of edges, meaning that deletion of any one type does not have a significant impact on the program classification accuracy. For some programs, however, deleting these types of edges can significantly reduce or improve the accuracy of program classification. For example, for program 1 and program 4, the deletion of almost every edge type impacts its accuracy, with the deletion of some edges even leading directly to the collapse of accuracy. Moreover, for these affected programs, different influences of different edges on the final result can be observed. We can therefore conclude that co-AFD is functioning as an entirety in most cases, that the different types of edges do not cause too much information redundancy, and that the information...
complements each other most of the time, which reveals that our co-AFD construction is appropriate.

In order to further explore the characteristics of GGANN and co-AFD, we also used the loss as a reference factor in order to analyze the convergence speed of different models when combined with different program graphs. From the convergence curve in Figure 8, the following conclusions can be drawn. Firstly, when AST is used as the program graph, GGNN exhibits a faster convergence speed and lower loss than TBCNN. For GGNN, the convergence speed is faster when co-AFD is used as the program graph compared with when AST is used. Under the same condition in which co-AFD is used, moreover the GGANN model converges slightly faster than the GGNN model, while the minimum loss value is also smaller. In summary, we can conclude from Figure 8 that GGANN and co-AFD each play their respective roles, and moreover that the best effect is achieved when the two are combined. The main reason for this is likely to be that, in the iterative process of the graph model, the information propagation between nodes is bidirectional; however, the convolution operation in TBCNN makes the information only propagate unidirectionally. When two different propagation methods are run on thousands of nodes, the difference in performance is therefore reflected.

![FIGURE 8. The variation of loss values for the four models, along with the iterations.](image)

The GGANN model can represent the characteristics of different nodes by learning the local structure of the node. Thus, we can infer that nodes with similar local structures should have similar feature representations. We therefore adopt the unsupervised clustering method K-means [42] to cluster the node feature vectors, then analyze the nodes in different clusters to determine whether the nodes in the same cluster have similar semantic information.

Figure 9 presents the partial clustering, along with the node information contained in each cluster.

![FIGURE 9. Partial clusters and the node information contained in each cluster.](image)

The GGANN model can represent the characteristics of different nodes by learning the local structure of the node. Thus, we can infer that nodes with similar local structures should have similar feature representations. We therefore adopt the unsupervised clustering method K-means [42] to cluster the node feature vectors, then analyze the nodes in different clusters to determine whether the nodes in the same cluster have similar semantic information.

VI. CONCLUSION AND DISCUSSION

With the goal of selecting a highly reliable compilation optimization pass, we propose a learning strategy utilizing GGANN. Our experimental results demonstrate that our model achieves a higher accuracy on the pass selection problem, and also performs better than similar neural network models and classic convolutional neural networks. Our work represents the first attempt to combine graph neural networks with program reliability, and it is clear that our experimental goals have been achieved. Although a program’s running time and code size are important program evaluation indicators, program reliability also cannot be ignored; in the booming aerospace field, for example, the reliability of the program is always the first consideration. Moreover, the reliability indicator we select is related to the program’s SDC_rate and T_exec. In other words, our reliability measurement does in fact take running time into account, since there is an undeniable relationship between program reliability and running time: the longer the program runs, the higher the probability of program error.

In this paper, we propose a novel method for compilation optimization pass selection with Graph Attention Neural Network (GGANN). The purpose of using GGANN is to explore the rich transitions among statements and generate accurate latent vectors of items. GGANN is able to capture transitions among statements and generate corresponding accurate node embedding vectors, which traditional neural network methods find it difficult to reveal. Based on accurate statement
embedding vectors, the proposed method constructs more reliable statement representations. Neural networks are universal function approximators and can approximate a given arbitrary function to an arbitrary degree of precision, but only in the limit of an infinite number of hidden units. By contrast, the attention mechanism realizes multiplicative interactions, which allows us to more easily approximate complex function models. Therefore, compared to GGNN and GNN, GGANN exhibits a natural advantage in the processing of complex program graphs such as co-AFD.

Our method is also impacted by certain limitations. The first of these is that the generation of the dataset is time-consuming. In order to label the dataset, it is also necessary to measure the reliability of the program within the dataset both before and after optimization. However, the reliability measurement takes more time. At the same time, since our proposed program graph is more complicated than AST, the information transfer process between graph nodes is also more complicated, meaning that the number of required parameters and the running time also increase. Secondly, as the use of different reliability metrics may have a certain impact on the final selection results, the universality of our model also needs to be improved. Finally, the work in this paper only selects the compilation optimization pass; in the actual compilation process, the generation of the compilation optimization sequence is an issue of greater concern. This will also be the direction of our future research.

The selection of the compilation optimization pass will be the basis of our next research work. In this work, we plan to combine the graph neural network with the compilation optimization sequence generation technology. Solving the phase-ordering problem will be the primary focus. At present, we already have a very good research foundation to draw on. Iterative compilation and reinforcement learning have been used in the past to solve the phase sequence problem. If we are able to take advantage of the graph neural network in this context, this will represent a significant improvement.

ACKNOWLEDGMENT
The authors would like to thank all members of the team for their joint efforts during the experiment, especially Dr. Zhuo Zhang for his leading role in the research process.

REFERENCES
[1] P. Kulkarni, “Finding effective optimization phase sequences,” ACM SIGPLAN Notices, vol. 38, no. 7, pp. 12–23, 2003.
[2] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragam-Kelley, J. Bosboom, U.-M. O’Reilly, and S. Amarasinghe, “OpenTuner: An extensible framework for program autotuning,” in Proc. 23rd Int. Conf. Parallel Archit. Compilation, 2013, pp. 313–142.
[3] M. R. Jantz and P. A. Kulkarni, “Performance potential of optimization phase selection during dynamic JIT compilation,” in Proc. 9th ACM SIGPLAN/SIGOPS Int. Conf. Virtual execution Environ., 2013, pp. 303–316.
[4] P. Lukuciejewski, S. Plazaer, H. Falk, P. Marwedel, and L. Thiele, “Multi-objective exploration of compiler optimizations for real-time systems,” in Proc. 15th IEEE Int. Symp. Object/Component/Service-Oriented Real-Time Distrib. Comput., 2010, pp. 115–122.
[5] P. Lukuciejewski, S. Plazaer, H. Falk, P. Marwedel, and L. Thiele, “Approximating Pareto optimal compiler optimization sequences—A trade-off between WCET, ACET and code size,” Software: Pract. Exper., vol. 41, no. 12, pp. 1437–1458, Nov. 2011.
[6] 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 Proc. Int. Symp. Code Gener. Optim. (COCO), Mar. 2007, pp. 185–197.
[7] 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.
[8] G. Fursin, Y. Kashnikov, A. W. Memon, Z. Chamski, O. Temam, M. Namolu, E. Yan-Tow, B. Mendelson, A. Zaks, E. Courtois, F. Bodin, P. Barnard, E. Ashton, E. Bonilla, J. Thomson, C. K. I. Williams, and M. O’Boyle, “Milestone GCC: Machine learning enabled self-tuning compiler,” Int. J. Parallel Prog., vol. 39, no. 3, pp. 296–327, Jun. 2011.
[9] G. Martins, R. Nobre, and J. M. Cardoso, “Clustering-based selection for the exploration of compiler optimization sequences,” ACM Trans. Archit. Code Optim., vol. 13, no. 1, pp. 1–28, 2016.
[10] 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.
[11] J. H. Foleiss, A. Faustino da Silva, and L. B. Ruiz, “The effect of combining compiler optimizations on code size,” in Proc. 30th Int. Conf. Chilean Comput. Sci. Soc., Nov. 2011, pp. 187–194.
[12] D. Plotnikov, D. Melnik, M. Vardanyan, R. Buchatskiy, and R. Zhuykov, “A framework for compiler optimization predictions,” in Proc. 9th Int. Conf. Comput. Sci. Inf. Technol. Revised Sel. Papers, Sep. 2013, pp. 1–7.
[13] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O’Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams, “Using machine learning to focus iterative optimization,” in Proc. Int. Symp. Code Gener. Optim., 2006, p. 11.
[14] F. Bodin, T. Kosuki, P. Knjznenburg, B. M. O’E., and E. Rohlou, “Iterative compilation in a non-linear optimisation space,” in Workshop Profile Feedback-Directed Compilation, Paris, France, Oct. 1998, pp. 1–7.
[15] A. H. Ashouri, “A Bayesian network approach for compiler auto-tuning for embedded processors,” in Proc. IEEE ESTIMedia, Oct. 2014, pp. 90–97.
[16] H. Liu, R. Zhao, Q. Wang, and Y. Li, “ALIC: A low overhead compiler optimization prediction model,” Wireless Pers. Commun., vol. 103, no. 1, pp. 809–829, Nov. 2018.
[17] F. Li, F. Tang, and Y. Shen, “Feature mining for machine learning based compilation optimization,” in Proc. 8th Int. Conf. Innov. Mobile Internet Services Ubiquitous Comput., Birmingham, U.K., 2014, pp. 207–214, doi: 10.1109/IMIS.2014.26.
[18] H. Leather, E. Bonilla, and M. O’boyle, “Automatic feature generation for machine learning-based optimising compilation,” ACM Trans. Archit. Code Optim., vol. 11, no. 1, pp. 1–32, Feb. 2014.
[19] A. H. Ashouri, G. Palermo, and C. Silvano, “An evaluation of auto-tuning techniques for the compiler optimization problems,” Tech. Rep., 2016.
[20] C. Dubach, J. Cavazos, B. Franke, G. Fursin, M. F. P. O’Boyle, and O. Temam, “Fast compiler optimisation evaluation using code-feature based performance prediction,” in Proc. Conf. Process., 2007, pp. 131–142, doi: 10.1145/1242531.1242553.
[21] W. Killian, R. Miceli, F. Park, M. Alvarez, and J. Cavazos, (2014). Performance Improvement in Kernels by Guiding Compiler Auto-vectorization Heuristics. [Online]. Available: http://www.prace-re.eu/IMG/pdf/WP183.pdf
[22] A. H. Ashouri, V. Zaccaria, S. Xydias, G. Palermo, and C. Silvano, “A framework for compiler level statistical analysis over customized VLIW architecture,” in Proc. 21st Int. Conf. Very Large Scale Integr., Istanbul, Turkey, 2013, pp. 124–129, doi: 10.1109/VLSI-SoC.2013.6737362.
[23] T. Conte, “Praise for embedded computing: A VLIW approach to architecture, compilers and tools,” Tech. Rep.
[24] M. Boussaa, O. Barias, B. Baudry, and G. Sunye, “NOTICE: A framework for non-functional testing of compilers,” in Proc. IEEE Int. Conf. Softw. Qual., Rel. Secur. (QRS), Aug. 2016, pp. 335–346.
[25] E. Daniel de Lima, T. C. de Souza Xavier, A. Faustino da Silva, and L. Beatryz Ruiz, “Compiling for performance and power efficiency,” in Proc. 23rd Int. Workshop Power Timing Model., Optim. Simul. (PATMOS), Sep. 2013, pp. 142–149.
Jianjun Xu was born in Anhui, China, in 1980. He received the master’s and Ph.D. degrees from the National University of Defense Technology, in 2004 and 2010, respectively. He is currently an Associate Professor with the National University of Defense Technology. He has undertaken many national key projects. He has published papers in international conference journals. His main research interest includes software reinforcement technology.

Xiankai Meng received the bachelor’s degree from Fudan University, Shanghai, and the master’s degree from the National University of Defense Technology, where he is currently pursuing the Ph.D. degree. He is also affiliated with the High Trust Software Teaching and Research Section, College of Computer. He has participated in many key scientific research projects. He has outstanding engineering capabilities. He has published several papers at international conferences. His main research interests include embedded systems and software reinforcement.

Haoyu Zhang was born in Anhui, China, in 1993. He received the master’s degree in software engineering from the National University of Defense Technology, in 2016, where he is currently pursuing the Ph.D. degree. He has internship experience in many Internet companies, such as Microsoft and Google. He has published a paper as first author at the top conference ACL. His main research interests include natural language processing and machine learning.

Zhuo Zhang was born in Henan, China, in 1984. He received the master's degree in software engineering from the National University of Defense Technology, in 2014. He is currently affiliated with the Guangxi Key Laboratory of Trusted Software, Guilin University of Electronic Technology. He has been undertaking major scientific research tasks in multiple engineering projects. His main research interests include program error localization and software reinforcement technology.

Long Li (Member, IEEE) received the Ph.D. degree from the Guilin University of Electronic Technology, Guilin, China, in 2018. He is currently a Lecturer with the School of Computer Science and Information Security, Guilin University of Electronic Technology. He undertakes Postdoctoral Research with Jinan University. His research interests include cryptographic protocols and privacy-preserving technologies in big data and the IoT.

Jiang Wu was born in Shanxi, China. He received the bachelor’s degree in software engineering from the National University of Defense Technology, in 2018, where he is currently pursuing the master’s degree with the High Trust Software Department, College of Computer. His research interests include software testing technology, combination of machine learning and software reinforcement, and compilation optimization.

150434
VOLUME 8, 2020