GraalNN: Context-Sensitive Static Profiling with Graph Neural Networks

Lazar Milikic
Oracle Labs
Switzerland
lazar.milikic@oracle.com

Milan Cugurovic
Oracle Labs
Serbia
milan.c.cugurovic@oracle.com

Vojin Jovanovic
Oracle Labs
Switzerland
vojin.jovanovic@oracle.com

Abstract

Accurate static profile prediction is crucial for achieving optimal program performance in the absence of dynamic profiles. However, existing static profiling methods struggle to fully exploit the complex structure of the compiler's intermediate representation and fail to effectively utilize the calling context information needed for accurate profile inference. To address these limitations, we introduce *GraalNN*, a Graph Neural Network-based static profiling framework that directly learns structural information from control-flow graphs. This reduces the reliance on handcrafted features and minimizes the effort required for feature engineering while improving the model's ability to predict profiles.

GraalNN adopts a two-stage approach: it predicts context-insensitive profiles during parsing and uses contextual information from the call graph to refine profiles during inlining. This methodology achieves a 10.13% runtime speedup across a diverse set of industry-standard benchmarks, surpassing state-of-the-art static profiling techniques by more than 2.5%. Furthermore, GraalNN improves throughput by 3.7% over other static profiling methods on real-world microservices.

CCS Concepts: • **Software and its engineering** → **Just-in-time compilers**; • **Computing methodologies** → **Machine learning**.

Keywords: Profile-Guided Optimization, Context-Sensitive Profile Inference, Machine Learning, Graph Neural Networks

ACM Reference Format:
Lazar Milikic, Milan Cugurovic, and Vojin Jovanovic. 2025. GraalNN: Context-Sensitive Static Profiling with Graph Neural Networks. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization (CGO '25), March 01–05, 2025, Las Vegas, NV, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3696443.3708958

CGO '25, March 01–05, 2025, Las Vegas, NV, USA

ACM ISBN 979-8-4007-1275-3/25/03
https://doi.org/10.1145/3696443.3708958

1 Introduction

Collecting accurate program profiles is an essential step in optimizing program performance, as profile-guided optimization (PGO) can leverage these profiles to tailor the compilation process to a program's runtime behavior [30, 40, 41]. Compilers that utilize dynamic profiling, where profiles are collected by executing the program with representative inputs, achieve substantial performance gains [8, 51]. For instance, BOLT reports up to a 20.4% speedup for GCC and Clang compilers [42], while the Graal compiler achieves a 33.14% performance boost [54].

Despite the clear benefits of PGO, the complexity and resource demands of collecting dynamic profiles limit its widespread adoption [8, 42]. Identifying representative inputs that accurately reflect typical application use cases is challenging, and collecting profiles based on these inputs requires two compilations and a profile-collection run, which can be time-consuming. As a result, many large-scale software systems, including MySQL [20], opt not to use PGO in production due to the difficulties associated with obtaining program profiles.

Static profiling techniques have emerged as an alternative to the challenges of dynamic profiling, allowing program profiles to be predicted without the need for runtime profiling and the overhead of repeated compilations. These methods, instead, rely either on heuristic-based approaches or Machine Learning (ML) models. Though efficient and straightforward, heuristic models depend heavily on the compiler-developer's intuition and often fail to cover corner cases [2, 53]. In contrast, ML-based models capture more complex program patterns and consistently outperform heuristic models in profile prediction accuracy [39, 47, 54].

State-of-the-art ML-based static profilers rely on traditional ML models, such as XGBoost [9], which face two key limitations: they struggle to directly process and exploit the structure of the compiler's intermediate representation (IR) and fail to use the information from the calling context effectively. These models rely entirely on handcrafted features to access structural data from the program's IR; however, this manual feature engineering is tedious and restricts the model's ability to fully represent the IR's complexity. Additionally, these models can only make predictions based on information from individual functions, neglecting the broader calling context.

To address the key limitations of traditional ML-based static profilers, we introduce the GraalNN framework, a Graph Neural Network (GNN)-based static profiling solution that directly exploits structural information from control-flow graphs (CFGs) while enabling context-sensitive profile predictions. Unlike previous methods that rely on manually engineered structural features, GNNs [49] allow us to capture structural information directly from the CFG. This reduces the effort needed for feature engineering and improves the model's ability to learn and represent complex control-flow dependencies [36]. By working directly on the graph structure, GNNs inherently learn the relationships between nodes in the CFG, resulting in more accurate and efficient profile predictions [18].

In addition to capturing structural information, GraalNN extends static profiling by predicting context-sensitive profiles. During the inlining phase, the context-insensitive profiles inferred during the parsing phase are refined by combining the caller's CFG with the callee's CFG. This process creates a unified graph that integrates critical contextual information, enabling GraalNN to predict more accurate profiles by considering the broader execution context in which each method is invoked [44, 52].

Our key contributions are as follows:

- We present GraalNN, an end-to-end GNN-based static profiling framework that directly leverages structural information from control-flow graphs (CFGs), addressing the limitations of traditional static profilers. The GraalNN static profiler is generally applicable within any ahead-of-time (AOT) compiler, as it operates on CFGs, but in this work, we apply it to the GraalVM Native Image compiler (§2).
- We reduce the reliance on manually engineered features by enabling GNNs to directly capture structural dependencies from CFGs, lowering feature engineering effort while improving the model's ability to represent complex control-flow relationships (§2.1).
- We enhance profiling accuracy by introducing context-insensitive and context-sensitive predictions enabling GraalNN to capture broader execution contexts and improve the accuracy of static profiling (§3).
- We achieve a 10.13% geometric mean runtime speedup on various industry-standard benchmarks, surpassing previous static profilers by over 2.5%, while keeping the binary size increase capped at 2.94%. Furthermore, we demonstrate the quality of GraalNN in a real-world cloud scenario, increasing the throughput of microservices demo apps by 13.16% while limiting the binary size increase to 5.05% (§4.3 and §4.4).

Section 2 introduces the GraalNN static profiler, while Section 3 details its implementation. Section 4 presents the evaluation results, and Section 5 reviews related work. Finally, Section 6 concludes the paper.

2 GraalNN Static Profiler

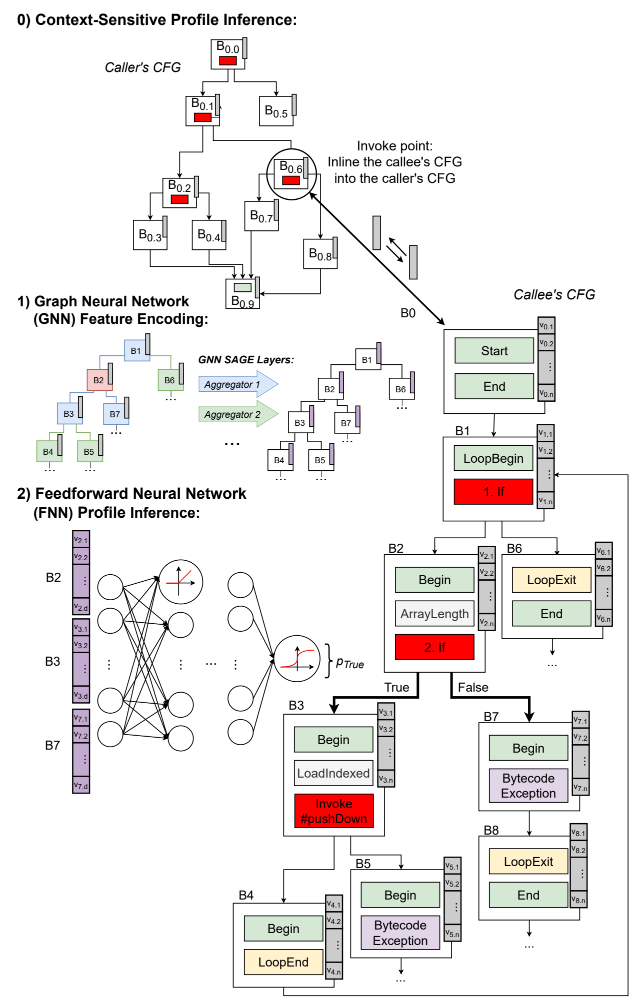

Figure 1. The overview of the GraalNN static profiler. The figure illustrates the steps required to infer context-insensitive profiles, as well as, context-sensitive profiles at the callee's CFG when the caller's CFG is provided.

GraalNN is a static profiler that leverages GNNs to predict both context-insensitive and context-sensitive branch execution probabilities (i.e., static program profiles [53]). The framework uses control-flow graphs (CFGs) of a method and its callers to estimate execution probabilities for control flow branches in `If` statements. Figure 1 illustrates the main stages of GraalNN's methodology.

In the first stage, GraalNN encodes the basic blocks of the CFG using predefined feature vectors that capture each block's control flow, low-level code, and structural characteristics. It employs a GNN model with GraphSAGE convolutions [27], which drive information exchange and aggregation between neighboring blocks across multiple layers (where each layer corresponds to one step of message passing). This process generates rich, learned representations for each CFG block, reflecting its structural role and positional significance within the broader CFG.

In the second stage, the learned representations of blocks ending with `If` nodes and their immediate successors are extracted and passed through a feedforward neural network (FNN) [3]. This network then infers the execution probabilities of the `True` branches for `If` statements[1] during parsing, enabling the generation of context-insensitive profiles.

To predict context-sensitive profiles, GraalNN incorporates the caller's CFG during inlining. As illustrated in step zero of Figure 1, the callee's graph is inlined in the caller's graph, enriching the context for profile prediction.

2.1 GNN Model Input

Using the CFG as input to GNN models offers several practical advantages for profile inference. By designing GraalNN to work on top of the CFG, we enable GNN models to utilize a minimal graph representation that preserves the essential control-flow information of the program source code [11, 15]. The CFG's compact structure reduces the computational overhead associated with graph processing, leading to faster training and inference times (see Appendix A for details on how inference times scale with the number of vertices in a graph). Furthermore, the CFG offers a cleaner and less noisy representation of control flow, improving the model's learning efficiency and facilitating faster convergence [26, 48]. Finally, using CFGs and CFG-extracted features as the input ensures that GraalNN's architecture is largely GraalVM-agnostic, as CFGs are standard in most optimizing compilers.

To process CFG data for use in GraalNN, we define input vectors to be fed into its GNN profile-inference models. In the context of GNNs, these inputs consist of two key components: the vertex[2] feature vector, which captures the characteristics of each graph vertex (i.e., each CFG block), and the edge matrix, which encodes the connections between vertices.

2.1.1 Vertex Features. Each CFG block is represented as a vector of nine features, as detailed in Table 1. These features are adopted from GraalSP [54], selecting only those that specifically characterize CFG blocks. Unlike GraalSP, which employs non-graph-based machine learning models requiring the input feature vector to be flattened and supplemented with additional features (seven more than our approach) to describe global structural information, our GNN-based method inherently learns structural and topological characteristics directly from the graph data. This significantly reduces the reliance on complex, handcrafted features used in previous works [39, 54].

As detailed in Table 1, the vertex features are categorized into three groups based on the characteristics of the CFG blocks they describe:

1. *Node Types*: The internal structure of each CFG block is represented by histograms of fixed and floating IR nodes (features 1 and 2 in Table 1). These histograms count the occurrences of each node type within a block. For instance, the histogram for the fixed nodes in block `B1` from Figure 1 would indicate one `If` node and one `LoopBegin` node. Node Types features can be extracted in any compiler that utilizes CFGs.
2. *Code Characteristic Estimation*: Low-level code characteristics are estimated using the Graal static cost model, which provides high-level statistical measures of a block's computational cost [33]. Specifically, we estimate the number of CPU cycles required to execute a CFG block and its corresponding assembly size (features 3 and 4 in Table 1). These estimates are obtained by summing the CPU cycles and assembly sizes for all fixed and floating nodes within the block. To enhance these features, nodes within a CFG block are further categorized based on their CPU cycle counts: nodes requiring 0 or 1 CPU cycle are labeled as *CPU Cheap*, while those requiring 64 or more cycles are labeled as *CPU Expensive*. The counts of these categorized nodes are then included as vertex features (features 5 and 6 in Table 1). While these features are specific to Graal, they can be generalized if the cost model proposed by Leopoldseder et al. [33] is incorporated into other compilers.
3. *Control-Flow Characteristics*: Features such as control split depth, loop depth, and dominator depth (features 7–9 in Table 1) characterize the control flow position of a CFG block within the overall graph. These features have been widely utilized in prior ML-based profile inference studies [7, 19, 39, 47, 54]. They enable GNNs to explicitly capture critical control-flow patterns by integrating these values with the extracted structural features, enhancing the model's ability to represent the broader program context.

Once the features of a CFG block are collected, they are organized into a vector form, as shown in Figure 1, where each CFG block has an associated feature vector. The vertex feature vector allocates fixed positions for all features: seven positions are reserved for numerical features (features 3–9 in Table 1), while features 1 and 2, which are categorical histograms, are converted into fixed-size vectors using one-hot encoding [28]. Each unique fixed and floating node type is assigned a specific position in the vector, representing the count of that node type in the CFG block. With 106 fixed

[1]The execution probability of the `False` branch is then implicitly defined.
[2]To clarify the distinction between intermediate representation (IR) nodes (see Figure 1, where each CFG block is constructed out of at least two IR nodes) and the vertices of the CFG, we use the term "vertex" or "CFG block" exclusively to refer to the nodes of the CFG. In contrast, "node" is reserved for IR nodes throughout this work.

Table 1. Summary of vertex features used for CFG blocks [54]. For histogram features, the total number of possible entries is indicated in brackets. Features marked with (*) are specific to Graal, while unmarked features can be generally extracted in any compiler that utilizes CFGs.

#	Feature Name	Type	Category	Description
1.	Fixed Nodes Histogram	Histogram (106)	Node Types	Histogram representing the distribution of fixed nodes in the block
2.	Floating Nodes Histogram	Histogram (64)	Node type	Histogram representing the distribution of floating nodes in the block
3.	Estimated CPU Cycles*	Numeric	Code Characteristic	Estimated number of CPU cycles required for executing nodes in the block
4.	Estimated Assembly Size*	Numeric	Code Characteristic	Estimated assembly size of nodes in the block
5.	CPU Cheap Count*	Numeric	Code Characteristic	Number of nodes in a block requiring 0 or 1 CPU cycle
6.	CPU Expensive Count*	Numeric	Code Characteristic	Number of nodes in a block requiring 64 or more CPU cycles
7.	Control Split Depth	Numeric	Control-Flow	Number of blocks containing *cfs*-nodes with a successor that dominates the given block
8.	Loop Depth	Numeric	Control-Flow	Number of nested loops that enclose the given block
9.	Dominator Depth	Numeric	Control-Flow	Depth of a block in the dominator tree

node types, 64 floating node types, and seven numerical features, the vertex feature vector has a total length of 177.

Additionally, numerical features are normalized to ensure stable and efficient model training [35]. This is achieved by transforming each numerical feature x using $x = (x - \mu_x)/\sigma_x$, where μ_x and σ_x represent the mean and standard deviation of x calculated from the training data. Histogram features, however, are not normalized since they typically consist of only zeros and ones.

2.1.2 Edge Matrix.

To encode the graph's structure for the GNN, we use an edge matrix that represents the connections between graph vertices. This matrix is constructed following the PyTorch Geometric (PyG) [22] specifications, as we leverage this library for implementing our GNN models. The edge matrix, `edgeMatrix`, is structured as a 2-by-M matrix, where M represents the number of edges in the graph. Each column i in the matrix denotes a directed edge, specifying a connection from block `edgeMatrix`$[0][i]$ to block `edgeMatrix`$[1][i]$. For example, the `edgeMatrix` for the callee's CFG depicted in Figure 1 is:

$$\texttt{edgeMatrix} = \begin{bmatrix} \texttt{B0} & \texttt{B1} & \texttt{B1} & \texttt{B2} & \texttt{B2} & \texttt{B3} & \texttt{B3} & \texttt{B4} & \texttt{B7} \\ \texttt{B1} & \texttt{B2} & \texttt{B6} & \texttt{B3} & \texttt{B7} & \texttt{B4} & \texttt{B5} & \texttt{B1} & \texttt{B8} \end{bmatrix}$$

For simplicity, the edge matrix shown above represents connections in only one direction. However, in practice, our GNN profile-inference models use an edge matrix that includes bidirectional connections for each pair of neighboring CFG blocks.

2.2 GNN Profile-Inference Model

Given the CFG, $\mathcal{G} = (\mathcal{B}, \mathcal{E})$, defined as a directed graph with vertices $\mathcal{B}$ (CFG blocks) and edges $\mathcal{E}$ (control-flow connections), we construct input representations for the GNN-based Profile Inference Model, as detailed in Section 2.1. Each CFG block is described by a feature vector of length 177, capturing its structural, computational, and control-flow characteristics. These feature vectors form the feature matrix $\mathbf{X} \in \mathbb{R}^{|\mathcal{B}| \times 177}$, where each row corresponds to the feature vector of a graph vertex.

The graph's connectivity is encoded in an edge matrix $\mathbf{E} \in \mathbb{R}^{2 \times |\mathcal{E}|}$. Each column of $\mathbf{E}$ specifies a directed edge, indicating a connection from vertex $\mathbf{E}[0, i]$ to vertex $\mathbf{E}[1, i]$. Together, $\mathbf{X}$ and $\mathbf{E}$ provide the structural and feature-based representation of the CFG, serving as input to the GNN model.

The GNN model consists of layers that combine `SAGEConv` and `GraphNorm`, formally defined as $f_{\text{layer}} = \texttt{GraphNorm} \circ \texttt{SAGEConv}$. Specifically:

$$f_{\text{in}} : \left(\mathbb{R}^{|\mathcal{B}| \times 177}, \mathbb{R}^{2 \times |\mathcal{E}|}\right) \rightarrow \mathbb{R}^{|\mathcal{B}| \times D}, \tag{1}$$

$$f_{\text{hidden}} : \left(\mathbb{R}^{|\mathcal{B}| \times D}, \mathbb{R}^{2 \times |\mathcal{E}|}\right) \rightarrow \mathbb{R}^{|\mathcal{B}| \times D}, \tag{2}$$

where D is the hidden feature size of the GNN. The `SAGEConv` graph convolution operation, derived from the GraphSAGE framework [27], exchanges and aggregates features from a vertex's local neighborhood, while `GraphNorm` [6] normalizes vertex features for stable and efficient training.

GraphSAGE is chosen for its ability to inductively learn vertex representations, enabling it to generalize to unseen vertices and CFG structures. This capability is critical in our application, where CFG blocks may contain IR nodes that appear infrequently in the training data or are entirely novel. By sampling and aggregating features from a block's local neighborhood, GraphSAGE efficiently captures the structural and topological properties of the CFG. Its lightweight and scalable design ensures computational efficiency, even for large graphs, while its robustness to out-of-distribution data enhances model reliability.

Additionally, the use of `GraphNorm` improves model performance by stabilizing training and accelerating convergence. Normalizing vertex features within the graph prevents feature magnitude imbalances, ensuring the model focuses on meaningful structural patterns in the CFG.

To compute execution probabilities, the extracted input representations are processed in two phases: GNN Feature Encoding and FNN Profile Inference, corresponding to steps one and two in Figure 1.

2.2.1 GNN Feature Encoding.

The input feature matrix $\mathbf{X} \in \mathbb{R}^{|\mathcal{B}|\times 177}$ and edge matrix $\mathbf{E} \in \mathbb{R}^{2\times|\mathcal{E}|}$ are passed through N GNN layers to encode the features of CFG blocks, where $N \geq 2$. Each layer alternates with a non-linear activation function, ϕ, where we use $\phi = \text{GELU}$ [29] due to its smooth activation outputs and superior performance compared to ReLU [1]. The GNN encoding is formally defined as:

$$\mathbf{H}^{(N)} = f_{\text{hidden}} \circ \underbrace{\phi\left(f_{\text{hidden}}\right) \circ \cdots \circ \phi\left(f_{\text{hidden}}\right)}_{(N-2)\text{-times}} \circ \phi\left(f_{\text{in}}\left(\mathbf{X}, \mathbf{E}\right)\right),$$

where $\mathbf{H}^{(N)} \in \mathbb{R}^{|\mathcal{B}|\times D}$ represents the final output matrix containing the encoded features of all CFG blocks after N layers. The functions f_{in} and f_{hidden} are defined in Equations 1 and 2, respectively. These encoded features, $\mathbf{H}^{(N)}$, encapsulate both the intrinsic characteristics of each CFG block and their contextual roles, relationships, and positions within the graph.

2.2.2 FNN Profile Inference.

Using the encoded features, we infer branch execution probabilities for all CFG blocks that terminate with `If` nodes. For a given block, such as `B2` in Figure 1, which ends with an `If` node, we consider the feature vectors of `B2` and its outgoing branches (`B3` for the `True` branch and `B7` for the `False` branch).

The encoded features of these blocks, denoted as $\mathbf{H}^{(N)}_{\texttt{B2}}$, $\mathbf{H}^{(N)}_{\texttt{B3}}$, $\mathbf{H}^{(N)}_{\texttt{B7}} \in \mathbb{R}^{D}$, are concatenated and passed through a feedforward neural network (FNN) [3] to compute the execution probability of the `True` branch:

$$p_{\texttt{B2,True}} = \sigma\left(\text{FNN}\left(\mathbf{H}^{(N)}_{\texttt{B2}} \oplus \mathbf{H}^{(N)}_{\texttt{B3}} \oplus \mathbf{H}^{(N)}_{\texttt{B7}}\right)\right),$$

where $\oplus$ denotes vector concatenation, and $\sigma(x) = \frac{1}{1+\exp(-x)}$ is the sigmoid activation function.

This process is repeated for all CFG blocks that terminate with `If` nodes.

2.3 Context-Sensitive Profile Inference

To predict *context-sensitive profiles*, we enhance the feature extraction process by incorporating additional contextual information from the caller's CFG, provided the specific call chain is available (step zero in Figure 1). Instead of relying solely on the CFG of the method where branch probabilities are being inferred, we substitute the caller's CFG block containing the `Invoke` node—representing the call site of the method—with the CFG of the callee. By inlining the callee's CFG into the caller's CFG, we construct a context-sensitive CFG that integrates both the method and its calling context. Features extracted from this enriched, context-sensitive CFG are then passed to GraalNN, enabling more accurate and informed predictions of branch probabilities.

To illustrate the inlining procedure, consider the caller's CFG $\mathcal{G}^*_{\text{caller}} = (\mathcal{B}^*_{\text{caller}}, \mathcal{E}^*_{\text{caller}})$ and the callee's CFG $\mathcal{G}'_{\text{callee}} = (\mathcal{B}'_{\text{callee}}, \mathcal{E}'_{\text{callee}})$, where the call site in the caller's CFG is represented by the block $b^*_{\texttt{Invoke}} \in \mathcal{B}^*_{\text{caller}}$. To construct a context-sensitive CFG, we replace $b^*_{\texttt{Invoke}}$ in $\mathcal{G}^*_{\text{caller}}$ with the entire graph $\mathcal{G}'_{\text{callee}}$, forming a new graph $\mathcal{G} = (\mathcal{B}, \mathcal{E})$. The new set of blocks, $\mathcal{B}$, is defined as:

$$\mathcal{B} = \left(\mathcal{B}^*_{\text{caller}} \cup \mathcal{B}'_{\text{callee}}\right) \setminus \{b^*_{\texttt{Invoke}}\},$$

and the new set of edges, $\mathcal{E}$, is:

$$\begin{aligned}\mathcal{E} = &\left(\mathcal{E}^*_{\text{caller}} \setminus \{(b^*_{\texttt{parent}}, b^*_{\texttt{Invoke}}), (b^*_{\texttt{Invoke}}, b^*_{\texttt{child}})\}\right)\\ &\cup \mathcal{E}'_{\text{callee}} \cup \{(b^*_{\texttt{parent}}, b'_{\texttt{entry}}), (b'_{\texttt{exit}}, b^*_{\texttt{child}})\},\end{aligned}$$

where $b'_{\texttt{entry}}$ and $b'_{\texttt{exit}}$ are the entry and exit blocks of $\mathcal{G}'_{\text{callee}}$.

The integration redirects the edges that previously connected $b^*_{\texttt{Invoke}}$ to its parent block $b^*_{\texttt{parent}}$ and child block $b^*_{\texttt{child}}$[3]. Specifically, the edge $(b^*_{\texttt{parent}}, b^*_{\texttt{Invoke}})$ is replaced with $(b^*_{\texttt{parent}}, b'_{\texttt{entry}})$, connecting the parent block to the entry block of $\mathcal{G}'_{\text{callee}}$. Similarly, the edge $(b^*_{\texttt{Invoke}}, b^*_{\texttt{child}})$ is replaced with $(b'_{\texttt{exit}}, b^*_{\texttt{child}})$, connecting the exit block of $\mathcal{G}'_{\text{callee}}$ to the child block.

Once constructed, the new graph $\mathcal{G}$ allows feature extraction as described in Section 2.1. This enables the prediction of branch probabilities at `If` nodes in $\mathcal{G}'_{\text{callee}}$ while incorporating the contextual information from the caller's CFG.

2.4 Branch-Probability Prediction Heuristics

To enhance the reliability of GraalNN, we incorporate branch probability heuristics to refine predicted profiles for both context-insensitive and context-sensitive cases. These heuristics are critical for mitigating the impact of outliers, which can cause significant degradation in program execution speed when predicted profiles are inaccurate [31]. By applying

[3] We assume that $b^*_{\texttt{Invoke}}$ has only one parent block and one child block. In practice, blocks ending with *cfs* nodes may have multiple child blocks, all of which must be appropriately connected to the exit blocks of $\mathcal{G}'_{\text{callee}}$.

these heuristics, we reduce the risks associated with deploying GraalNN in practical compiler optimization tasks, ensuring robust and reliable performance.

We adopt two heuristics proposed by Čugurović et al. as part of the GraalSP framework [54]: the loop heuristic and the dead-end heuristic.

The *loop heuristic* assigns a minimum probability, referred to as the *loop guard* (LG), to branches that lead into loop bodies, prioritizing these paths in execution predictions. This ensures that the compiler can always recognize the importance of loops, which are typically executed multiple times.

The *dead-end heuristic* addresses branches leading to program termination, capping their probabilities using an *exit guard* (EG). Specifically, it limits the probability of terminating branches when non-terminating branches have substantial assembly sizes. This is controlled by two thresholds: the maximum exit probability and the minimum non-terminating branch assembly size, where the former is conditioned on the latter. By applying this heuristic, the compiler avoids overestimating paths leading to early termination, ensuring that non-terminating branches with optimization potential are not overlooked.

GraalNN achieves greater robustness and reliability in profile-guided optimizations by refining predicted probabilities with these heuristics. This approach minimizes the chance of generating suboptimal code due to inaccurate predictions and prevents the compiler from making overly aggressive optimizations that could result in significant performance penalties. The combination of these heuristics ensures a balance between predictive accuracy and practical usability in real-world compiler scenarios.

3 Implementation

GraalNN is integrated into the compiler through a two-phase process. The first phase is an offline stage where the models are trained and validated to ensure their quality and accuracy. The second phase is an online stage where the trained models are utilized within the compiler to predict branch probabilities at `If` nodes.

3.1 Training and Validation

Before integrating the GraalNN into the Graal compiler, it is necessary to first train the GNN profile-inference models.

3.1.1 Training and Validation Dataset. To ensure high-quality training data for GNN-based static profilers, we build our training dataset from real-world projects [39, 54]. We construct our training set using various libraries included in the GraalVM Reachability Metadata Repository. These libraries cover a wide range of applications, including database management, data compression, logging, networking, web servers, HTTP protocols, and email support [17].

Control-flow graphs are collected in two phases: the parsing phase for context-insensitive cases and the inlining phase for context-sensitive cases. During the inlining phase, we gather the CFGs requiring profile predictions and incorporate the caller graphs by inlining them, as described in Section 2.3, to form a single context-sensitive CFG.

In total, we collected 227,683 CFGs during the parsing phase and 970,693 CFGs during the inlining phase. Each graph contains at least one `If` node with branch probabilities labeled using instrumentation-based profiling, resulting in a total of 3,043,875 labeled `If` nodes. The 90% of the collected graphs are used for training, with the remaining 10% reserved for validation and model selection.

3.1.2 Probability Labels. To generate labels for training our model, we leverage the PGO workflow. This process involves compiling programs from the training dataset and collecting execution counts for branches at `If` nodes during runtime. For each `If` node, the execution counts of the `True` and `False` branches are denoted as $n_{\texttt{True}}$ and $n_{\texttt{False}}$, respectively. The execution probability of the `True` branch, which serves as the label for the corresponding `If` node, is computed as $p_{\texttt{True}} = \frac{n_{\texttt{True}}}{n_{\texttt{True}}+n_{\texttt{False}}}$.

3.1.3 Distribution of Execution Counts. As explained in Section 3.1.2, labels are derived from the execution counts of branches at each `If` node. The total execution count for an `If` node is calculated as the sum of its `True` and `False` branch counts, $n_{\texttt{True}}+n_{\texttt{False}}$. For each `If` node, we normalize the execution count of each `If` node by dividing it by the total execution count across all observed methods in the program.

Figure 2 illustrates that the distribution of normalized execution counts at `If` nodes is long-tailed and highly skewed. This indicates that a small subset of `If` nodes is executed frequently, while the large majority of branches are executed only a handful of times, making their contribution negligible in comparison to the most frequently executed nodes.

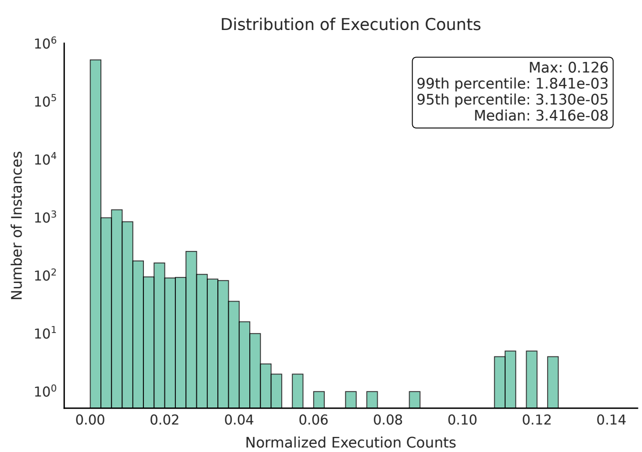

Figure 2. Distribution of normalized execution counts for `If` nodes from the training dataset (Log Scale, 50 bins).

3.1.4 Model Training.

To optimize program performance, it is crucial that GraalNN accurately predicts profiles for the most frequently executed `If` nodes, as prediction of their profiles has the greatest impact on overall runtime [54]. In contrast, predictions for less frequently executed nodes are less critical to performance.

To ensure that the GNN profile-inference models during training prioritize frequently executed `If` nodes, we use a weighted binary cross-entropy (BCE) loss [38] with the normalized execution counts from Section 3.1.3 serving as training weights for each `If` node. This approach enables the model to implicitly learn to recognize and prioritize hot code regions. We optimize the model's parameters using the AdamW optimizer [37] with a learning rate of 0.001.

Training is conducted for 200 epochs, with the model checkpointed at the epoch that achieves the lowest validation loss. This approach ensures optimal performance regardless of the total number of training epochs.

We train separate models tailored to each phase to accommodate both context-insensitive and context-sensitive profile inference, which are performed during the parsing and inlining phases. The need for distinct models arises because the underlying nature of profiles varies significantly between these two phases. In the parsing phase, execution counts are aggregated across all possible call paths for which a method is reachable, resulting in context-insensitive profiles. Conversely, in the inlining phase, execution counts are specific to the context of the particular call graph from which a method is invoked, leading to context-sensitive profiles. Given these differences, we develop one profile inference model using context-insensitive labels and training graphs from the parsing phase, and another using context-sensitive labels and context-enhanced CFGs from the inlining phase.

3.1.5 Model Implementation.

Our GNN model is implemented using the PyTorch Geometric [22] library, with `SAGEConv` layers serving as the primary graph convolution mechanism. The feedforward neural network (FNN) head, responsible for probability prediction, is implemented using the PyTorch [43] library.

Both context-insensitive and context-sensitive GNN profile inference models are configured with $N = 8$ `SAGEConv` layers and a hidden feature size of $D = 64$. The FNN head is a two-layer neural network with hidden sizes of 128 and 64. Overall, each model contains 115,265 trainable parameters, determined through extensive hyperparameter tuning to minimize the weighted binary cross-entropy (BCE) loss on the validation set.

3.2 Model Deployment

Once the GNN profile inference models (context-insensitive and context-sensitive) are trained, they are deployed into the compiler to predict branch probabilities under the GraalNN framework. These models are integrated into two distinct compiler phases:

1. **Parsing Phase.** The context-insensitive model is deployed during the parsing phase, immediately after constructing the CFG for each program method. Deploying the model at this stage enables compilers to use the predicted profiles and perform PGO at the earliest possible point, unlocking optimization opportunities throughout the subsequent compilation stages.
2. **Inlining Phase.** The context-sensitive model is deployed during the inlining phase, when the call context becomes available. At this stage, the compiler constructs context-sensitive CFGs, refining the profiles generated during the parsing phase to enhance their accuracy based on the specific context under which the method is invoked.

The trained models are integrated into the Graal compiler using the ONNX Java Runtime [16], which facilitates running PyTorch-trained models across Windows, Linux, and Darwin OS distributions.

4 Evaluation

In this section, we present the experiments conducted to evaluate the impact of deploying GraalNN within GraalVM Native Image. Our primary focus is to assess the framework's effect on program execution time. Additionally, we analyze its impact on binary size and compilation time for the compiled applications.

We describe the evaluation setup (§4.1), assess GraalNN on industry-standard benchmark suites (§4.2), compare it with state-of-the-art static profilers (§4.3), and evaluate its performance on real-world microservices benchmarks in the cloud (§4.4).

4.1 Evaluation Setup

4.1.1 Software and Hardware Setup.

We use GraalVM Enterprise Edition 24 using the Serial Garbage Collector and running on Oracle Linux Server 7.4. We conducted experiments on a machine equipped with two Intel E5-2699 CPUs @ 2.30 GHz (18-core processors) with 756 GB of DDR4 memory clocked at 2400 MHz.

To conduct measurements, we pin the evaluated process to the first core and use only the first-core's memory to minimize measurement instability from non-uniform memory access. We run benchmarks with hyper-threading enabled, using a 40 GB RAM disk to eliminate I/O noise and set CPU C-states to 0 to prevent power-saving modes from affecting results.

4.1.2 Test Benchmarks.

We evaluate our models using standard Graal compiler benchmark suites, ensuring no overlap with the programs used during training. These include three comprehensive suites: the Renaissance suite [45], the

Table 2. Impact of deploying different GNN-based static profiler configurations on runtime, binary size, and compile time in the GraalVM Native Image compiler, expressed as percentage changes relative to a baseline that does not employ any static profiler or have access to dynamic profiles.

Configuration	Runtime Speedup (%)	Binary Size Incr. (%)	Compile Time Incr. (%)
GNN@Parsing (No-Guards)	8.20	2.60	**14.19**
GNN@Parsing (LG=20 EG=(80, 50))	7.16	2.67	14.36
GNN@Parsing (LG=20 EG=(90, 100))	7.55	**2.59**	14.22
GraalNN (No-Guards)	9.48	2.65	21.25
GraalNN (LG=20 EG=(80, 50))	9.15	2.70	20.07
GraalNN (LG=30 EG=(80, 50))	**10.13**	2.94	20.85

DaCapo suite [4], and the DaCapo con Scala suite [50], comprising a total of 29 diverse benchmarks. The benchmarks cover a wide range of tasks and applications, including cross-language static code analyzers, compilers, class file decoders, code formatters, computationally intensive tasks like ray tracing with the Sunflow rendering engine, XML processing tools, machine learning algorithms, and text search and processing using Apache Lucene. Additional workloads simulate microblogging with Twitter Finagle, test Java streams and concurrent algorithms, execute genetic algorithms with the Jenetics library, and evaluate frameworks such as Akka.

These benchmarks, representing comprehensive applications with significant codebases, are commonly employed in the literature to assess proposed optimization methods [5, 44]. Moreover, their inclusion facilitates direct performance comparisons with GraalSP [54], the current state-of-the-art static profiling framework for the Graal compiler.

4.1.3 Evaluation Metrics. For each test benchmark, we evaluate the impact of our models on *program runtime*, *binary size*, and *compile time* by measuring the percentage change in these metrics relative to a baseline, for which branch execution probabilities are uniformly distributed (i.e., no heuristic or ML-based profile predictions are applied). Each metric is averaged over n runs for a given benchmark, with n determined based on the benchmark's stability [54]. To aggregate results across multiple benchmarks, we utilize the geometric mean [23, 24].

4.2 Evaluation of GNN-based static profilers

To select the optimal configuration for our GNN-based static profiler, we evaluate the performance of the GraalNN framework, which predicts both context-insensitive and context-sensitive branch probabilities, against a configuration that uses the GNN-Profile Inference Model only during the parsing phase, referred to as GNN@Parsing.

Additionally, we assess the effects of using different guard parameters for the branch probability prediction heuristics described in Section 2.4. Specifically, we explore variations in the loop guard (LG) and exit guard (EG) values—initially set at LG=20 and EG=(80, 50) in the GraalSP approach [54]—as these have been shown to improve the runtime speedup of their baseline ML model by over 2%. Configurations not using branch probability prediction heuristics are denoted by the *No-Guards* tag.

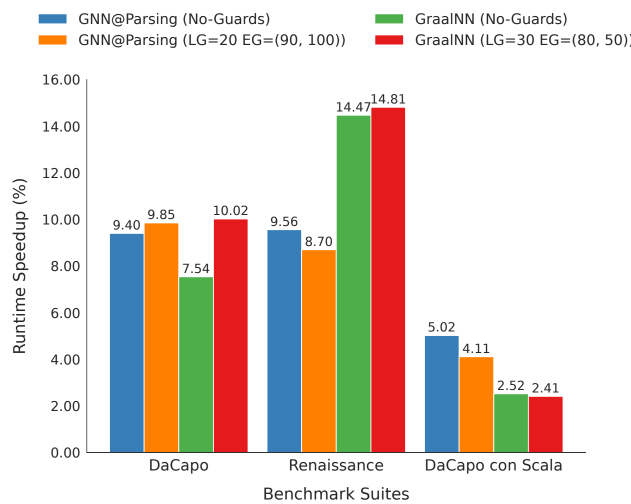

Figure 3. Comparison of runtime speedup across benchmark suites for different GNN-based configurations.

4.2.1 Runtime Performance. Table 2 demonstrates that all GraalNN configurations outperform the GNN@Parsing configurations. Notably, the best-performing GraalNN configuration, with LG=30 and EG=(80, 50), exceeds the performance of GNN@Parsing (No-Guards) configuration by approximately 2%, confirming that refining profile predictions during the inlining phase with context-sensitive profiles positively impacts the runtime performance. Additionally, tuning the branch probability heuristic guards further enhances the runtime speedup of GraalNN and improves model robustness in real-world scenarios by reducing the likelihood of critical mispredictions. Our results show that GraalNN performs optimally when the probability of staying in a loop is at least 30%. In contrast, GNN@Parsing configurations are negatively affected by aggressive guard settings, prompting us to adjust the EG values to be less aggressive.

Interestingly, branch probability heuristics do not have as strong a positive effect on GraalNN and GNN@Parsing as they do with GraalSP. This difference can be attributed to the higher descriptive power of GraalNN's GNN model compared to GraalSP's XGBoost model. The GNN is significantly more capable of capturing the distribution of training set probabilities, reducing the need for guard-based corrections. In some cases, applying guards even negatively impacts the performance of the GNN-based static profilers.

Figure 3 illustrates the runtime performance of the GNN-based static profilers across benchmark suites. GraalNN configurations perform exceptionally well on the Renaissance suite, surpassing GNN@Parsing configurations by over 5%. However, all models exhibit less pronounced performance on the DaCapo con Scala benchmark suite, likely due to its older codebase, which diverges significantly from the modern Java code used for model training.

4.2.2 Binary Size. All configurations result in a comparable impact on the binary size of compiled programs, with an average increase of approximately 2.65%, with an exception for the GraalNN (LG=30 and EG=(80, 50)), which incurs a slightly higher increase of 2.94% due to a more aggressive loop guard (Table 2). These increases in binary size are attributed to optimizations such as code duplication, which are applied more aggressively based on the inferred profiles. In contrast, the baseline scenario, where execution probabilities are uniformly distributed across branches, applies duplication less aggressively, leading to smaller binary sizes [34].

4.2.3 Compile Time. As anticipated, GraalNN configurations result in higher compile times due to the use of GNN models in two distinct compiler phases (Table 2). This increase is primarily attributed to the additional extraction of GNN input vectors, GNN inference, and the generation of higher-quality profiles that trigger PGO within the compiler. In experiments conducted on an *Intel i7-1370P* with 18 cores[4], the parsing phase required an average of 2.41 additional seconds for the test benchmarks, directly reflecting the time added by GNN input extraction and inference. The remaining increase in compile time arises indirectly as the compiler spends more time optimizing code based on the improved profile information. Moreover, GraalNN incurs additional computational overhead by predicting context-sensitive profiles on inlined CFGs, which are inherently larger and more resource-intensive to process.

4.2.4 Discussion. Based on the results, we conclude that GraalNN (LG=30, EG=(80, 50)) is the best-performing configuration, offering the highest runtime speedup with only a marginal increase in binary size and compile time compared to other evaluated configurations. Detailed results and benchmark-specific comparisons are presented in Appendix B for further insights.

[4]Developer's laptop

In circumstances where compile time-saving is crucial, we recommend using GNN@Parsing (LG=20 EG=(90, 100)), which still achieves a strong runtime speedup of 7.55% while being over 6% faster to compile than GraalNN (LG=30 EG=(80, 50)) and having a lower binary size increase. Take note that even though GNN@Parsing (No-Guards) appears better than GNN@Parsing (LG=20 EG=(90, 100)) across all three metrics, we prefer the GNN@Parsing configurations whose predictions are guarded by branch probability heuristics, as static profilers that are purely ML-based can harm compiler performance due to mispredictions on unseen data and vulnerability to outliers [47].

4.3 Comparison with Existing Profiling Techniques

For comparison with relevant static profilers, we evaluate our two best GNN-based configurations: GNN@Parsing (LG=20, EG=(90, 100)) and GraalNN (LG=30, EG=(80, 50)). We compare these configurations directly against:

- The Graal compiler utilizing the branch probability prediction heuristics proposed by Wu and Larus [53], which are similar to the heuristics adopted by Clang's default compiler settings [32].
- The Graal compiler employing GraalSP [54] as its static profiler, which serves as the default static profiling mechanism in the *GraalVM Native Image* [52].

Since GraalNN is tailored to Java bytecode, we excluded direct comparisons with VESPA [39], a static profiler specifically designed for the BOLT binary optimizer [42].

We express the impact of each static profiler relative to the compiler baseline, which uniformly assigns execution probabilities over branches (does not use any static profiler or has access to dynamic profiles). The summary of the results is presented in Table 3.

4.3.1 Runtime Performance. GNN-based solutions accomplish better runtime speedup gains compared to the other two static profilers, Wu and Larus's heuristics-based and GraalSP (Table 3). Notably, our two-phase static profiler, GraalNN (LG=30 EG=(80, 50)), outperforms these static profilers by 4.49% and 2.67%, respectively.

Figure 4 illustrates that GNN-based solutions perform best on the DaCapo suite, nearly doubling the runtime boost demonstrated by GraalSP. The GraalNN framework also shows significant improvement, being over 11% better than Wu and Larus's profiler and over 5% better than GraalSP on the Renaissance suite. Although the Wu and Larus static profiler performs best on the DaCapo con Scala suite, its performance comes at the expense of generating much larger binaries and significantly increasing compilation time (Table 3). Moreover, the benchmarks of this suite are less relevant as their codebase is considerably older and diverges from the more modern Java code used for the training of GraalSP, GNN@Parsing, and GraalNN models.

Table 3. Comparison of the impact of deploying GraalNN, GNN@Parsing, and other existing profiling techniques on runtime, binary size, and compile time in the GraalVM Native Image compiler. The results are expressed as percentage changes relative to a baseline that does not use any static profiler or have access to dynamic profiles.

Configuration	Runtime Speedup (%)	Binary Size Incr. (%)	Compile Time Incr. (%)
GNN@Parsing (LG=20 EG=(90, 100))	7.55	**2.59**	14.22
GraalNN (LG=30 EG=(80, 50))	**10.13**	2.94	20.85
Wu and Larus	5.64	31.60	78.51
GraalSP	7.46	3.91	**12.01**

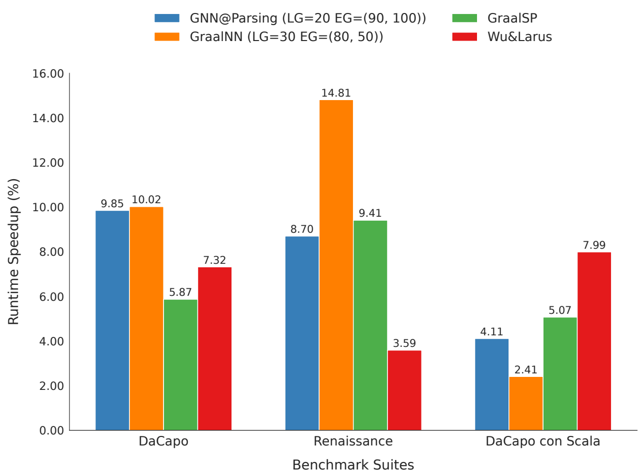

Figure 4. Runtime speedup across different benchmark suites for various configurations and profilers.

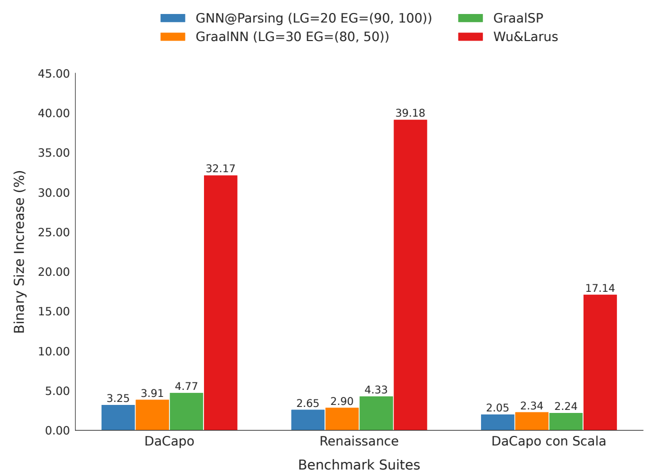

Figure 5. Binary size increase across different benchmark suites for various configurations and profilers.

4.3.2 Binary Size. In terms of binary size increase, our GNN-based solutions are considerably more efficient than the other two static profilers, generating binaries that are at least 1% smaller on average than those produced by GraalSP and over ten times smaller than the binaries produced by the Graal compiler enhanced with Wu and Larus's heuristics (Table 3). Figure 5 shows that a similar trend is observed across the test suites, except for DaCapo con Scala, where the generated binaries are of similar sizes for our GNN-based solutions and GraalSP.

4.3.3 Compile Time. As shown in Table 3 and Figure 6, our GNN-based static profilers come at the cost of increased compile time compared to GraalSP, primarily due to the computational overhead associated with GNN model inference, especially as GraalNN performs profile inference in two compilation phases. In contrast, GraalSP is built upon the XGBoost model, which is a much lighter model compared to GNNs.

While the Wu and Larus static profiler directly applies predefined heuristics without the overhead of ML model inference, its heuristics significantly increase the binary size, and a considerable amount of time is spent on generating large binaries, making it the most time-consuming method in terms of compilation.

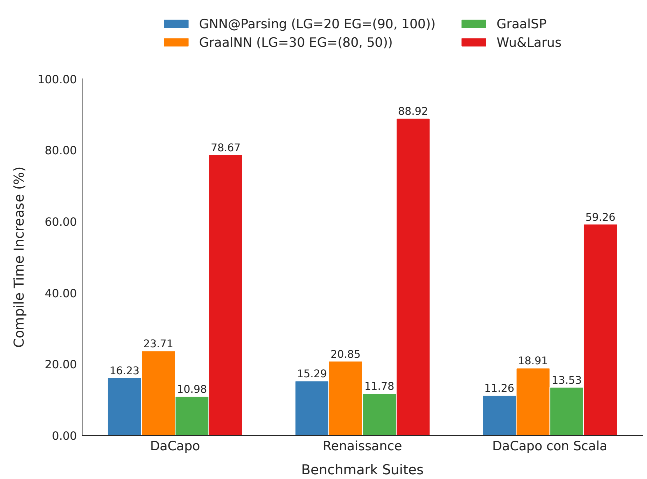

Figure 6. Compile time increase across different benchmark suites for various configurations and profilers.

4.3.4 Discussion. Our GNN-based frameworks demonstrate significant improvements over existing static profilers. Among the evaluated configurations, GraalNN (LG=30, EG=(80, 50)) stands out as the best-performing approach, achieving a runtime speedup of 10.13%—exceeding other static profiling solutions by over 2.5%, a 35% relative improvement—while keeping the binary size increase smaller than both GraalSP and Wu and Larus's profilers.

Moreover, GNN@Parsing (LG=20, EG=(90, 100)) represents the second-best configuration. It slightly surpasses GraalSP in runtime speedup while exhibiting a much smaller binary size increase and only a marginal compile time increase relative to GraalSP. Additionally, it offers a significant compile time decrease compared to GraalNN (LG=30, EG=(80, 50)). Therefore, in scenarios where the fast compilation is critical, GNN@Parsing (LG=20, EG=(90, 100)) provides a compelling alternative, balancing performance and compilation efficiency.

4.4 Evaluation on Microservices Benchmarks

To further demonstrate the applicability and robustness of our approach, we evaluate GraalNN and related configurations on a suite of modern microservices applications developed using three popular Java web service frameworks: Spring [14], Micronaut [12], and Quarkus [13]. The microservices benchmark suite includes the following: the Spring PetClinic application, a comprehensive demo for Spring that simulates a pet clinic management system; the Micronaut Shopcart application, an e-commerce system built with the Micronaut framework; and the Quarkus Tika application, which utilizes the Tika PDF and Tika ODT modules to process and extract metadata from documents using the Quarkus framework.

4.4.1 Experimental Setup. To more closely reflect real-world scenarios in the evaluation of microservices benchmarks, we utilize a cloud-based machine equipped with AMD EPYC 7742 processors, featuring a base frequency of 2.25 GHz and a maximum boost frequency of 3.4 GHz. The rest of the software and hardware setup is identical to that described in Section 4.1.1.

4.4.2 Evaluation with O2 Optimization Budget. We first evaluated the models using the default O2 optimization budget, also employed in previous experiments. As shown in Table 4, GraalNN demonstrates substantial performance gains, doubling the runtime speedup of GraalSP while achieving the smallest binary size increase among the evaluated static profilers. However, as illustrated in Figure 7, the performance gap between GraalNN and GNN@Parsing is relatively narrow. Notably, GNN@Parsing outperforms GraalNN in both Tika benchmarks, suggesting that the optimization budget in O2 constrains the full potential of the inferred profiles.

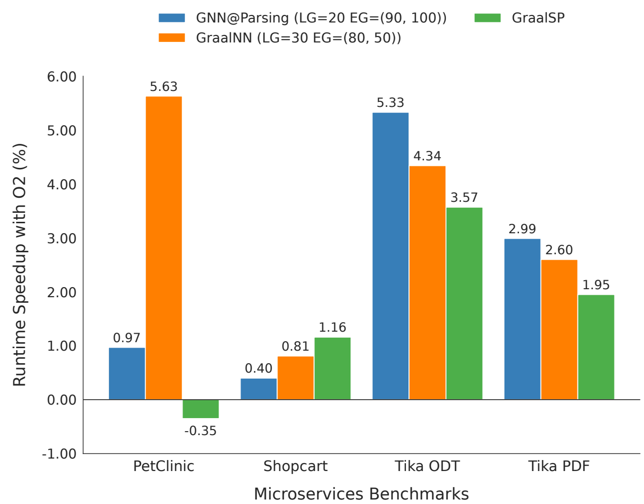

Figure 7. Runtime speedup with *O2* budget across different microservices for various configurations and profilers.

4.4.3 Evaluation with O3 Optimization Budget. To fully explore the impact of GraalNN, we extended the evaluation to the O3 optimization budget. Although O3 is not typically utilized due to its significant impact on binary size (resulting in a 42% increase across microservices benchmarks), it provides a broader evaluation of optimization potential.

Under the O3 budget, all configurations demonstrated improved performance, with the advantages of GraalNN becoming more pronounced. As detailed in Table 4 and illustrated in Figure 8, GraalNN outperformed other static profilers by an average of over 3.2% in runtime speedup and achieved the best performance across all microservices benchmarks, while maintaining the lowest binary size increase.

This improvement, however, comes at the cost of a substantial compile time increase. The dual-phase GNN inference and the subsequent computational demands of processing larger inlined CFGs contribute to this overhead. Nevertheless, the results highlight the effectiveness of GraalNN in delivering superior runtime performance with trade-offs in compile time.

5 Related Work

State-of-the-art static profilers use regression ML models, classification ML models, and handcrafted heuristics to predict profiles.

Čugurović et al. [54] introduce *GraalSP*, a static profiler that uses an XGBoost ML model [10] for regression to predict program profiles. *GraalSP* extracts features from the high-level, graph-based compiler IR. In addition to the XGBoost model, *GraalSP* incorporates two branch probability prediction heuristics to maintain program performance when the ML model encounters outliers. *GraalSP* is integrated into *GraalVM Native Image* [52].

Table 4. Impact of deploying GraalNN, GNN@Parsing, and GraalSP on runtime, binary size, and compile time in the GraalVM Native Image compiler for *microservices benchmarks*. Results are expressed as percentage changes relative to a baseline without static or dynamic profiles. The table compares two optimization budgets, O2 and O3.

Opt. Budget	Configuration	Runtime Speedup (%)	Binary Size Incr. (%)	Compile Time Incr. (%)
O2	GNN@Parsing (LG=20 EG=(90, 100))	2.40	4.20	15.84
	GraalNN (LG=30 EG=(80, 50))	**3.33**	**3.74**	62.19
	GraalSP	1.58	7.76	**12.99**
O3	GNN@Parsing (LG=20 EG=(90, 100))	9.92	5.53	17.38
	GraalNN (LG=30 EG=(80, 50))	**13.16**	**5.05**	129.11
	GraalSP	9.46	7.43	**12.70**

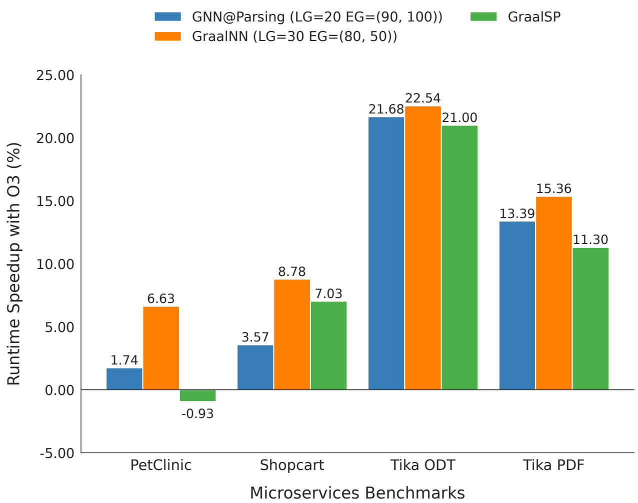

Figure 8. Runtime speedup with *O3* budget across different microservices for various configurations and profilers.

Moreira et al. [39] developed *VESPA*, a static profiler that uses deep learning, specifically a feed-forward deep neural network (DNN) [25], to predict program profiles. The proposed DNN model leverages binary-level features and extends the feature set introduced by Calder et al. [7], originally proposed for solving the static branch prediction task [53]. The authors integrated the *VESPA* static profiler into the *BOLT* binary optimizer [42].

Raman and Li [46] and Eichenberger et al. [21] utilized deep neural network ML models to predict branch profiles in the LLVM compiler. They extract features from LLVM IR and use data center workloads and modern C++ applications to create training datasets.

Rotem and Cummins [47] proposed a static profiler that utilizes an XGBoost ML model for multiclass classification. They discretized branch execution probabilities into categories, thereby transforming the branch probability prediction into a multiclass classification task. Wu and Larus [53] developed a set of static heuristics to predict branch execution probabilities. They extended the static branch prediction heuristics originally proposed by Ball and Larus [2], which are used by default in the *Clang* compiler [32].

6 Conclusion

In this work, we introduced GraalNN, a novel framework for static profiling that leverages GNNs directly on control-flow graphs (CFGs). By allowing the model to inherently capture structural information from CFGs, GraalNN reduces the reliance on handcrafted features, simplifying the profiling process and enhancing the accuracy of profile predictions.

A key contribution of our approach is the two-stage profiling process: GraalNN predicts context-insensitive profiles during the parsing phase and refines them with context-sensitive profiles during the inlining phase. This context-aware profiling framework enables GraalNN to capture both local and global control-flow patterns, significantly improving the precision of static profiles. As a result, GraalNN achieves a runtime speedup of 10.13% on a wide range of industry-standard benchmarks, outperforming previous approaches by over 2.5%, while also reducing binary size by over 0.9% when compared to other static profiling solutions. Finally, GraalNN demonstrates its effectiveness on modern microservices benchmarks, achieving a runtime speedup of up to 13.16% with minimal binary size increase, highlighting its practicality for real-world cloud applications.

7 Acknowledgments

We want to express our heartfelt gratitude to Milena Vujosevic Janicic (University of Belgrade, Oracle Labs Serbia), Boris Spasojevic (Oracle Labs, Zurich), and Aleksandar Pejovic (Oracle Labs, Serbia) for their support, insightful discussions, and inspiring ideas that have made this project possible. We would also like to thank Thomas Wuerthinger (Oracle Labs, Zurich), who came up with the original idea of static profiling for Graal Native Image, and express our gratitude for the support he provided during this project. We also extend our gratitude to the entire GraalVM team and anonymous reviewers for their invaluable feedback.

References

[1] Abien Fred Agarap. 2019. Deep Learning using Rectified Linear Units (ReLU). arXiv:1803.08375 [cs.NE] https://arxiv.org/abs/1803.08375

[2] Thomas Ball and James R. Larus. 1993. Branch prediction for free. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation* (Albuquerque, New Mexico, USA) *(PLDI '93)*. Association for Computing Machinery, New York, NY, USA, 300–313. https://doi.org/10.1145/155090.155119

[3] G. Bebis and M. Georgiopoulos. 1994. Feed-forward neural networks. *IEEE Potentials* 13, 4 (1994), 27–31. https://doi.org/10.1109/45.329294

[4] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo benchmarks: java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications* (Portland, Oregon, USA) *(OOPSLA '06)*. Association for Computing Machinery, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488

[5] Rodrigo Bruno, Vojin Jovanovic, Christian Wimmer, and Gustavo Alonso. 2021. Compiler-assisted object inlining with value fields. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 128–141. https://doi.org/10.1145/3453483.3454034

[6] Tianle Cai, Shengjie Luo, Keyulu Xu, Di He, Tie-Yan Liu, and Liwei Wang. 2021. GraphNorm: A Principled Approach to Accelerating Graph Neural Network Training. arXiv:2009.03294 [cs.LG] https://arxiv.org/abs/2009.03294

[7] Brad Calder, Dirk Grunwald, Michael Jones, Donald Lindsay, James Martin, Michael Mozer, and Benjamin Zorn. 1997. Evidence-based static branch prediction using machine learning. *ACM Trans. Program. Lang. Syst.* 19, 1 (jan 1997), 188–222. https://doi.org/10.1145/239912.239923

[8] Dehao Chen, Neil Vachharajani, Robert Hundt, Shih-wei Liao, Vinodha Ramasamy, Paul Yuan, Wenguang Chen, and Weimin Zheng. 2010. Taming hardware event samples for FDO compilation. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Toronto, Ontario, Canada) *(CGO '10)*. Association for Computing Machinery, New York, NY, USA, 42–52. https://doi.org/10.1145/1772954.1772963

[9] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '16)*. ACM. https://doi.org/10.1145/2939672.2939785

[10] Tianqi Chen and Carlos Guestrin. 2016. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. 785–794.

[11] Cliff Click and Michael Paleczny. 1995. A simple graph-based intermediate representation. In *Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations* (San Francisco, California, USA) *(IR '95)*. Association for Computing Machinery, New York, NY, USA, 35–49. https://doi.org/10.1145/202529.202534

[12] The Micronaut Framework Community. 2024. Micronaut Framework. https://micronaut.io/. Accessed: 2024-12-24.

[13] The Quarkus Community. 2024. Quarkus. https://quarkus.io/. Accessed: 2024-12-24.

[14] The Spring Petclinic Community. 2024. Spring Petclinic. https://spring-petclinic.github.io/. Accessed: 2024-12-24.

[15] Keith D. Cooper and Linda Torczon. 2023. Chapter 4 - Intermediate Representations. In *Engineering a Compiler (Third Edition)* (third edition ed.), Keith D. Cooper and Linda Torczon (Eds.). Morgan Kaufmann, Philadelphia, 159–207. https://doi.org/10.1016/B978-0-12-815412-0.00010-3

[16] Microsoft Corporation. 2024. ONNX Runtime. https://onnxruntime.ai/

[17] Oracle Corporation. 2024. GraalVM Reachability Metadata. https://github.com/oracle/graalvm-reachability-metadata GitHub repository.

[18] Chris Cummins, Zacharias V. Fisches, Tal Ben-Nun, Torsten Hoefler, Michael F P O'Boyle, and Hugh Leather. 2021. ProGraML: A Graph-based Program Representation for Data Flow Analysis and Compiler Optimizations. In *Proceedings of the 38th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 139)*, Marina Meila and Tong Zhang (Eds.). PMLR, 2244–2253. https://proceedings.mlr.press/v139/cummins21a.html

[19] Veerle Desmet, Lieven Eeckhout, and Koen De Bosschere. 2005. Using Decision Trees to Improve Program-Based and Profile-Based Static Branch Prediction. , 336-352 pages. https://doi.org/10.1007/11572961_27

[20] Paul DuBois. 2013. *MySQL*. Addison-Wesley Professional.

[21] Zachary Eichenberger, Shin Lee, Harvin Mumick, and Shivan Prasad. [n. d.]. Simulating Profile Guided Optimization Using Learned Branch Probabilitiy Predictors. ([n. d.]).

[22] Matthias Fey and Jan E. Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*.

[23] Philip J. Fleming and John J. Wallace. 1986. How not to lie with statistics: the correct way to summarize benchmark results. *Commun. ACM* 29, 3 (mar 1986), 218–221. https://doi.org/10.1145/5666.5673

[24] Matt Friehauf, Mikaela Hertel, Juan Liu, and Stacey Luong. 2013. On Compass and Straightedge Constructions: Means. PDF.

[25] Ian Goodfellow. 2018. Deep learning.

[26] Shivani Gupta and Atul Gupta. 2019. Dealing with Noise Problem in Machine Learning Data-sets: A Systematic Review. *Procedia Computer Science* 161 (2019), 466–474. https://doi.org/10.1016/j.procs.2019.11.146 The Fifth Information Systems International Conference, 23-24 July 2019, Surabaya, Indonesia.

[27] William L. Hamilton, Rex Ying, and Jure Leskovec. 2018. Inductive Representation Learning on Large Graphs. arXiv:1706.02216 [cs.SI] https://arxiv.org/abs/1706.02216

[28] John T Hancock and Taghi M Khoshgoftaar. 2020. Survey on categorical data for neural networks. *J. Big Data* 7, 1 (Dec. 2020). https://doi.org/10.1186/s40537-020-00305-w

[29] Dan Hendrycks and Kevin Gimpel. 2023. Gaussian Error Linear Units (GELUs). arXiv:1606.08415 [cs.LG] https://arxiv.org/abs/1606.08415

[30] Urs Hölzle and David Ungar. 1994. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation* (Orlando, Florida, USA) *(PLDI '94)*. Association for Computing Machinery, New York, NY, USA, 326–336. https://doi.org/10.1145/178243.178478

[31] Azme Khamis, Zuhaimy Ismail, Haron Khalid, and Ahmad Mohammed. 2005. The Effects of Outliers Data on Neural Network Performance. *Journal of Applied Sciences* 5 (01 2005).

[32] Chris Lattner. 2008. LLVM and Clang: Next generation compiler technology. In *The BSD conference*, Vol. 5. 1–20.

[33] David Leopoldseder, Lukas Stadler, Manuel Rigger, Thomas Würthinger, and Hanspeter Mössenböck. 2018. A cost model for a graph-based intermediate-representation in a dynamic compiler. In *Proceedings of the 10th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages* (Boston, MA, USA) *(VMIL 2018)*. Association for Computing Machinery, New York, NY, USA, 26–35. https://doi.org/10.1145/3281287.3281290

[34] David Leopoldseder, Lukas Stadler, Thomas Würthinger, Josef Eisl, Doug Simon, and Hanspeter Mössenböck. 2018. Dominance-based duplication simulation (DBDS): code duplication to enable compiler optimizations. In *Proceedings of the 2018 International Symposium on*

Code Generation and Optimization (Vienna, Austria) *(CGO 2018)*. Association for Computing Machinery, New York, NY, USA, 126–137. https://doi.org/10.1145/3168811

[35] Boyi Li, Felix Wu, Ser-Nam Lim, Serge Belongie, and Kilian Q. Weinberger. 2021. On Feature Normalization and Data Augmentation. arXiv:2002.11102 [cs.LG] https://arxiv.org/abs/2002.11102

[36] Pan Li and Jure Leskovec. 2022. The Expressive Power of Graph Neural Networks. In *Graph Neural Networks: Foundations, Frontiers, and Applications*, Lingfei Wu, Peng Cui, Jian Pei, and Liang Zhao (Eds.). Springer Singapore, Singapore, 63–98.

[37] Ilya Loshchilov and Frank Hutter. 2019. Decoupled Weight Decay Regularization. arXiv:1711.05101 [cs.LG] https://arxiv.org/abs/1711.05101

[38] Anqi Mao, Mehryar Mohri, and Yutao Zhong. 2023. Cross-Entropy Loss Functions: Theoretical Analysis and Applications. arXiv:2304.07288 [cs.LG] https://arxiv.org/abs/2304.07288

[39] Angélica Aparecida Moreira, Guilherme Ottoni, and Fernando Magno Quintão Pereira. 2021. VESPA: static profiling for binary optimization. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 144 (oct 2021), 28 pages. https://doi.org/10.1145/3485521

[40] Robert Morgan. 1998. *Building an optimizing compiler.* Digital Press, USA.

[41] Guilherme Ottoni and Bertrand Maher. 2017. Optimizing function placement for large-scale data-center applications. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization* (Austin, USA) *(CGO '17)*. IEEE Press, 233–244.

[42] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. 2018. BOLT: A Practical Binary Optimizer for Data Centers and Beyond. arXiv:1807.06735 [cs.PL] https://arxiv.org/abs/1807.06735

[43] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. arXiv:1912.01703 [cs.LG] https://arxiv.org/abs/1912.01703

[44] Aleksandar Prokopec, Gilles Duboscq, David Leopoldseder, and Thomas Wïrthinger. 2019. An Optimization-Driven Incremental Inline Substitution Algorithm for Just-in-Time Compilers. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 164–179. https://doi.org/10.1109/CGO.2019.8661171

[45] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. 2019. Renaissance: benchmarking suite for parallel applications on the JVM. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 31–47. https://doi.org/10.1145/3314221.3314637

[46] Easwaran Raman and Xinliang David Li. 2022. Learning branch probabilities in compiler from datacenter workloads. *arXiv preprint arXiv:2202.06728* (2022).

[47] Nadav Rotem and Chris Cummins. 2022. Profile Guided Optimization without Profiles: A Machine Learning Approach. arXiv:2112.14679 [cs.PL] https://arxiv.org/abs/2112.14679

[48] Arun Saseendran, Lovish Setia, Viren Chhabria, Debrup Chakraborty, and Aneek Barman Roy. 2019. Impact of Noise in Dataset on Machine Learning Algorithms. https://doi.org/10.13140/RG.2.2.25669.91369

[49] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2009. The Graph Neural Network Model. *IEEE Transactions on Neural Networks* 20, 1 (2009), 61–80. https://doi.org/10.1109/TNN.2008.2005605

[50] Andreas Sewe, Mira Mezini, Aibek Sarimbekov, and Walter Binder. 2011. Da capo con scala: design and analysis of a scala benchmark suite for the java virtual machine. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Portland, Oregon, USA) *(OOPSLA '11)*. Association for Computing Machinery, New York, NY, USA, 657–676. https://doi.org/10.1145/2048066.2048118

[51] Baptiste Wicht, Roberto A. Vitillo, Dehao Chen, and David Levinthal. 2014. Hardware Counted Profile-Guided Optimization. arXiv:1411.6361 [cs.PL] https://arxiv.org/abs/1411.6361

[52] Christian Wimmer, Codrut Stancu, Peter Hofer, Vojin Jovanovic, Paul Wögerer, Peter B. Kessler, Oleg Pliss, and Thomas Würthinger. 2019. Initialize once, start fast: application initialization at build time. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 184 (oct 2019), 29 pages. https://doi.org/10.1145/3360610

[53] Youfeng Wu and James R. Larus. 1994. Static branch frequency and program profile analysis. *Proceedings of MICRO-27. The 27th Annual IEEE/ACM International Symposium on Microarchitecture* (1994), 1–11. https://api.semanticscholar.org/CorpusID:11222408

[54] Milan Čugurović, Milena Vujošević Janičić, Vojin Jovanović, and Thomas Würthinger. 2024. GraalSP: Polyglot, efficient, and robust machine learning-based static profiler. *Journal of Systems and Software* 213 (2024), 112058. https://doi.org/10.1016/j.jss.2024.112058

A Impact of Graph Size on GNN Inference Time

We evaluated the scaling of CPU inference time with graph size by running a GNN model with 8 `SAGEConv` layers and a hidden size of 64 on training CFG graphs, using an *Intel i7 - 1370P* with 18 cores. The average inference time is 4.93ms.

Figure 9 shows inference time increasing with graph size, with a significant jump for graphs exceeding 1000 vertices. This highlights the importance of using compact graph abstractions, like CFGs, to minimize compile-time overhead.

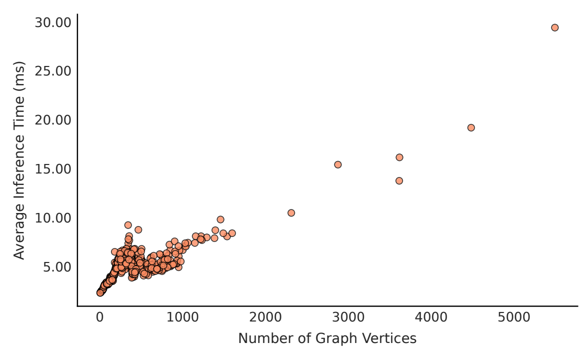

Figure 9. GNN inference time increases with graph size.

B Detailed Test Benchmark Results

We compare the performance of our best GNN-based static profiling configurations against GraalSP across test benchmarks. Figure 10 illustrates the runtime speedup, Figure 11 depicts the increase in binary size, and Figure 12 highlights the compile time increase for each benchmark, all relative to the default configuration that does not have access to static or dynamic profiles.

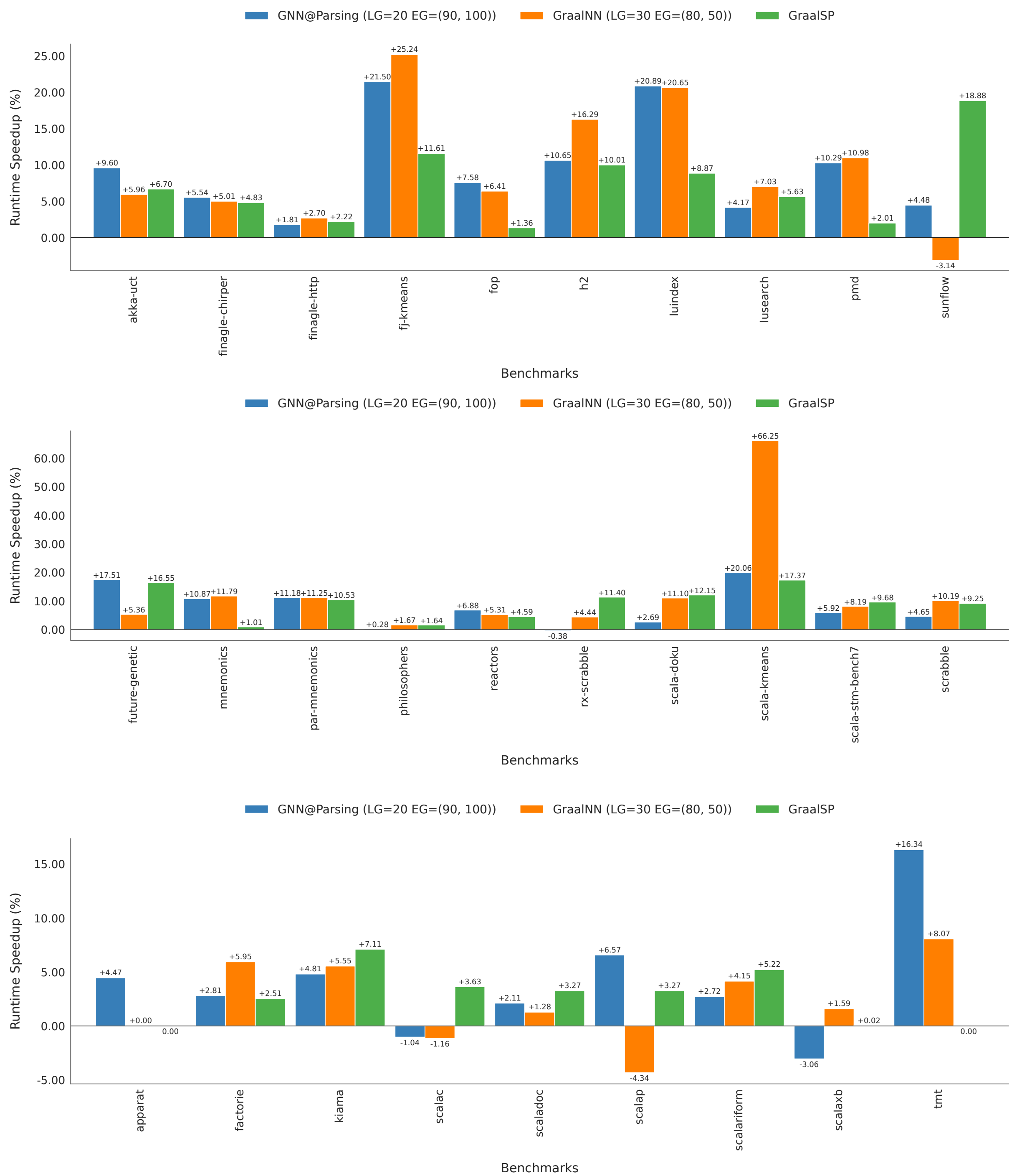

Figure 10. Runtime speedup of GNN@Parsing (LG=20 EG=(90, 100)), GraalNN (LG=30 EG=(80, 50)), and GraalSP over test benchmarks (note: Apparat and TMT benchmarks could not be evaluated for GraalSP).

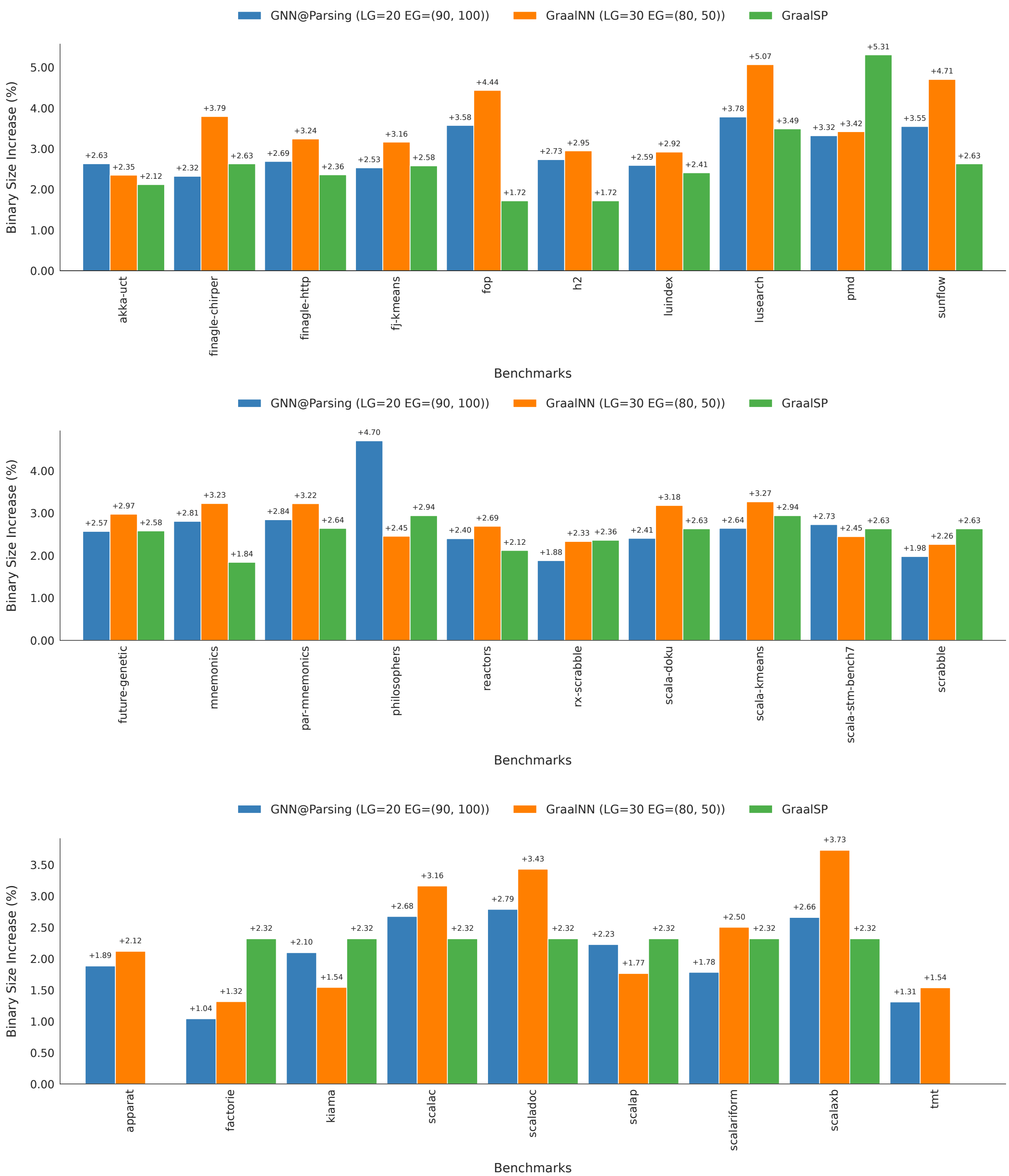

Figure 11. Binary size increase of GNN@Parsing (LG=20 EG=(90, 100)), GraalNN (LG=30 EG=(80, 50)), and GraalSP over test benchmarks (note: Apparat and TMT benchmarks could not be evaluated for GraalSP).

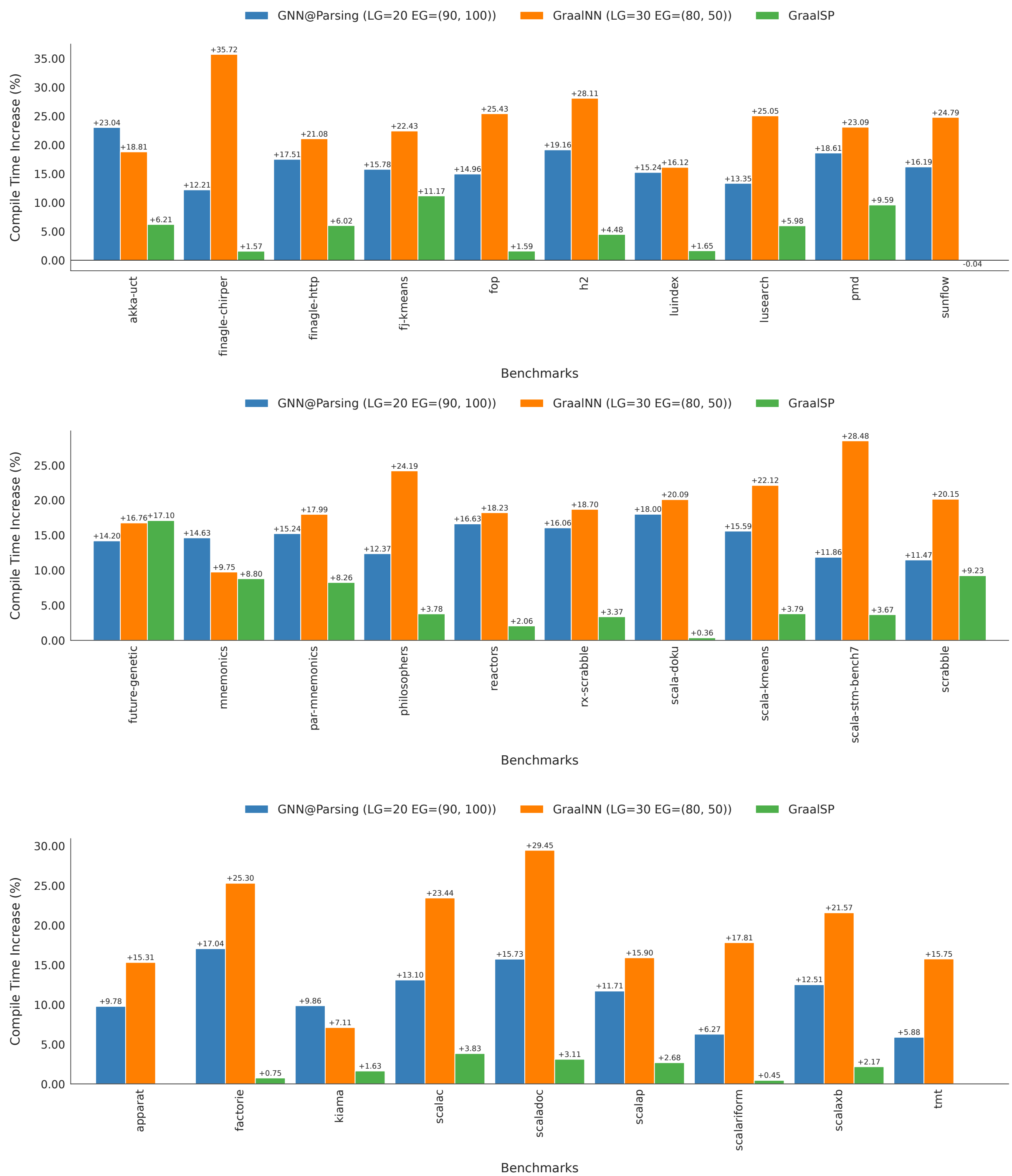

Figure 12. Compile time increase of GNN@Parsing (LG=20 EG=(90, 100)), GraalNN (LG=30 EG=(80, 50)), and GraalSP over test benchmarks (note: Apparat and TMT benchmarks could not be evaluated for GraalSP).