The Journal of Systems and Software 213 (2024) 112058

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

GraalSP: Polyglot, efficient, and robust machine learning-based static profiler☆

Milan Čugurović [a,b,*], Milena Vujošević Janičić [a,b], Vojin Jovanović [c], Thomas Würthinger [c]

[a] Faculty of Mathematics, University of Belgrade, Belgrade, Serbia
[b] Oracle Labs, Belgrade, Serbia
[c] Oracle Labs, Zürich, Switzerland

ARTICLE INFO

Keywords:
Compilers
GraalVM
Static profiler
Machine learning
Regression
XGBoost ensemble

ABSTRACT

Compilers use profiles to apply profile-guided optimizations and produce efficient programs. Dynamic profilers collect high-quality profiles but require identifying suitable profile collection workloads, introduce additional complexity to the application build pipeline, and cause significant time and memory overheads. Modern static profilers use machine learning (ML) models to predict profiles and mitigate these issues. However, state-of-the-art ML-based static profilers handcraft features, which are platform-specific and challenging to adapt to other architectures and programming languages. They use computationally expensive deep neural network models, thus increasing application compile time. Furthermore, they can introduce performance degradation in the compiled programs due to inaccurate profile predictions.

We present GraalSP, a portable, polyglot, efficient, and robust ML-based static profiler. GraalSP is portable as it defines features on a high-level, graph-based intermediate representation and semi-automates the definition of features. For the same reason, it is also polyglot and can operate on any language that compiles to Java bytecode (such as Java, Scala, and Kotlin). GraalSP is efficient as it uses an XGBoost model based on lightweight decision tree models and robust as it uses branch probability prediction heuristics to ensure the high performance of compiled programs. We integrated GraalSP into the Graal compiler and achieved an execution time speedup of 7.46% geometric mean compared to a default configuration of the Graal compiler.

1. Introduction

Profile-guided optimization (PGO), also known as feedback-driven optimization, is a widely used compiler technique that leverages profiles to improve the performance of compiled programs. Profiles refer to data that characterize a program's execution, such as the number of times each branch was executed, the number of function calls, and the caller-callee relationship between functions. PGO uses profiles to make informed decisions during the compilation process (Morgan, 1998; Ottoni and Maher, 2017; Panchenko et al., 2019). For example, optimizations are more aggressively performed in frequently called functions, resulting in more efficient programs. PGO improves various applications, such as compilers (Wicht et al., 2014; Lattner, 2020), warehouse-scale applications (Chen et al., 2016), web browsers (Mozilla Foundation, 2023; Google Corporation, 2023), and game engines (Epic Games Inc., 2022).

The execution time speedup that PGO achieves is significant but can vary up to 52.10% (Panchenko et al., 2019). To get the best from PGO, it is necessary to collect high-quality profiles (Wicht et al., 2014; Chen et al., 2010). Dynamic profiling gives profiles of the best quality (He et al., 2022) but at a high cost, as it increases:

- **Application build pipeline complexity, including time and memory footprint** as instead of one compilation process, the pipeline contains two compilations and one profile-collection run which often takes a considerable amount of time and memory (Chen et al., 2010; He et al., 2022; Nethercote and Seward, 2007; Cheng et al., 2006).
- **Developer's inconveniences,** as it is challenging to find comprehensive inputs that mimic the application's common use cases, hit all of the important code parts, and provide well-distributed profiles (Chen et al., 2010). Furthermore, it is challenging to keep the inputs updated as the application evolves.

Profiling overheads are especially important in the context of continuous integration where they accumulate over time. Despite all the

☆ Editor: Dr. Lingxiao Jiang.

* Corresponding author at: Faculty of Mathematics, University of Belgrade, Belgrade, Serbia.
E-mail address: milan.cugurovic@matf.bg.ac.rs (M. Čugurović).

https://doi.org/10.1016/j.jss.2024.112058
Received 23 August 2023; Received in revised form 29 March 2024; Accepted 6 April 2024
Available online 16 April 2024

benefits that PGO offers, there are large projects, such as MySQL, that still do not put it in production (DuBois, 2013).

Static profilers (Wu and Larus, 1994; Rotem and Cummins, 2021; Moreira et al., 2021) are an essential alternative to dynamic profilers. Instead of profile collection, they use static program features to predict branch execution probabilities. This way, they simplify the complexity of the application build pipeline, reduce the memory and time overheads, and free developers from the inconveniences of finding comprehensive inputs for profile collection (He et al., 2022). However, the accuracy of static profilers is critical as incorrect predictions can lead to wrong optimization decisions and cause a performance decrease (Choi et al., 2012).

State-of-the-art static profilers take advantage of handcrafted heuristics (Wu and Larus, 1994; Ball and Larus, 1993), classification (Kotsiantis et al., 2007; Osisanwo et al., 2017), and regression (Caruana and Niculescu-Mizil, 2006; Alpaydin, 2020) machine learning (ML) models. Handcrafted heuristics are simple and effective, defined based on developers' intuition and practical experience. ML models can predict significantly more fine-grained profiles than heuristics (Moreira et al., 2021). However, ML models are vulnerable to outliers, which can emerge in any data distribution and cannot be avoided (Huang et al., 2011; Kurakin et al., 2016). This means that static profilers that are only ML-based can have a negative impact on performance (Rotem and Cummins, 2021).

In this paper, we present GraalSP, a robust ML-based static profiler that is a fusion of accurate predictions from an ML model and protection of handcrafted heuristics against outliers. Our main contributions are:

An expressive, language-independent, novel feature set defined on a high-level, graph-based compiler intermediate representation (IR), Graal IR (Leopoldseder et al., 2015).

Detailed analysis of a dataset built from real-world Java applications. The analysis provides insights for training an ML model that should drive a static profiler.

A lightweight and accurate XGBoost regression ML model that drives GraalSP.

Branch probability prediction heuristics that ensure high performance in binaries optimized with profiles generated by GraalSP, even in the presence of outliers.

GraalSP: polyglot, efficient, and robust static profiler that is built on top of the regression XGBoost ML model and branch probability prediction heuristics and operates on any application written in languages that compile to the Java bytecode, such as Java, Scala, and Kotlin.

An efficient end-to-end solution that is integrated into the Graal compiler (Wimmer et al., 2019) and officially deployed in June 2023. Based on a comprehensive evaluation performed on 28 benchmarks from suites used in the Graal compiler development (Prokopec et al., 2019b; Blackburn et al., 2006; Sewe et al., 2011), GraalSP achieves execution time savings of geometric mean 7.46%.

Overview of the paper. Section 2 introduces the dynamic and static profilers, background on machine learning, and the basics of the Graal compiler. Section 3 explains the proposed static profiler, GraalSP, while Section 4 discusses the implementation and integration of GraalSP into the Graal compiler. Section 5 evaluates the proposed approach, while Section 6 presents a case study. Section 7 summarizes the related work, and Section 8 discusses GraalSP in the context of relevant static profilers and presents threats to validity. Section 9 concludes the paper and gives directions for future work.

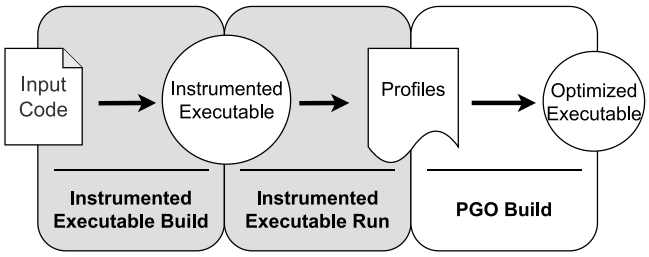


Fig. 1. Instrumentation-based profile collection (in gray) and PGO build.

2. Background

This section briefly overviews the application build pipeline that employs the PGO, dynamic and static profilers, supervised learning, the Graal compiler, and its intermediate representation, Graal IR.

2.1. Profiling and PGO

Compilers use profiles to perform PGO. Profiles can be collected by dynamic profilers or predicted by static profilers. Dynamic profilers can perform instrumentation-based profiling or sampling. On the other hand, static profilers predict the profiles based on heuristics or ML models. Dynamic profilers are also used when creating a labeled dataset, which serves as training data for ML models.

Fig. 1 illustrates the build pipeline that collects profiles using the instrumentation-based profiler and performs the PGO based on the collected profiles (Wimmer et al., 2019; Stallman et al., 2003). This pipeline consists of two compilations and one profile-collection run. The instrumentation-based profiler modifies the program's source code by adding instrumentation — additional code able to record the program runtime information. The instrumented source code is then compiled into an instrumented executable that collects profiles during the profile-collection run. In the subsequent compilation, the compiler optimizes the program with PGO based on the collected profiles. The instrumented program is larger and less efficient than the optimized program, i.e., the profiling process introduces a significant run-time and memory usage overhead. Several different approaches address a trade-off between information quality and high overhead, usually by combining instrumented and normal execution (Arnold and Ryder, 2001; Moseley et al., 2007; Cho et al., 2013).

Besides the program's source code, profilers can instrument a program's binary, like, for example, Valgrind (Nethercote and Seward, 2007). Valgrind avoids separated instrumented build and simultaneously instruments and executes code. This way, the profiling costs are higher than with profilers that modify the program's source code (He et al., 2022), but this approach has the advantage that it can be used even when the source code is not available.

Sampling-based profilers (Luk et al., 2004; Chen et al., 2016; Ottoni and Maher, 2017; Chen et al., 2011; Graham et al., 1982) have a build pipeline similar to the pipeline illustrated in Fig. 1. Instead of compiling the instrumented program, the first step involves building a non-instrumented executable. The second step involves a sampling run, which refers to periodically tracing the program's execution and collecting profiles. Sampling-based profilers usually rely on hardware counters and introduce significantly less overhead than their instrumentation-based counterpart (Whaley, 2000; Novillo, 2014). The main drawback of sampling-based profilers is that profiles collected via sampling are less precise than those collected via instrumentation (He et al., 2022). As collected at the binary level, compilers lose the accuracy of profiles collected by sampling if profiles are retrofitted at the earlier stages of compilation (Chen et al., 2011).

Trained ML models predict profiles, avoid costly profile collection, and mitigate the overheads of dynamic profiling (Moreira et al., 2021;

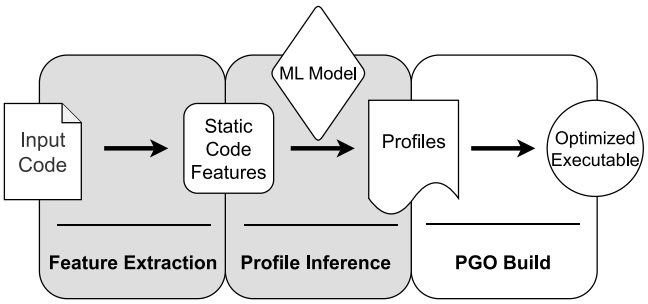


Fig. 2. Static profile prediction (in gray) and PGO build.

Rotem and Cummins, 2021). As illustrated in Fig. 2, a PGO build supported by a static profiler commences with extracting features that characterize code segments. The static profiler extracts features from a compiler's IR (Rotem and Cummins, 2021), or a binary of an input program (Moreira et al., 2021), depending on the approach used. Subsequently, the static profiler invokes a trained ML model to predict profiles based on the extracted features. PGO then uses these predicted profiles to generate optimized programs.

2.2. Supervised machine learning

Supervised learning (Mitchell, 2007; Shalev-Shwartz and Ben-David, 2014) uses a labeled dataset to train the ML model to learn a mapping between features and corresponding labels of instances from the training dataset. In a labeled dataset, every instance is characterized by a set of features and associated with the correct output, i.e., target value or label. Larger, diverse, and representative datasets generally lead to better generalization and more accurate models (Gong et al., 2023; Jain et al., 2020).

2.2.1. Features

Features can be numerical or categorical. *Numerical features* represent quantitative data as integers or real numbers. *Categorical features* can take a limited number of possible values, which are usually represented as text categories. It is necessary to convert categorical features into a suitable format for utilization in ML models. This is typically done by one-hot encoding (Murphy, 2022) which works well for categorical features with a few categories (Rodríguez et al., 2018).

Variance-based feature selection (Li et al., 2017) and principal component analysis (PCA) (Abdi and Williams, 2010; Wold et al., 1987) are the two most commonly used techniques that reduce the number of input features while preserving the essential information and structure of the data. Dimensionality reduction is used to improve the ML model performance. Also, it reduces overfitting (Dietterich, 1995; Hawkins, 2004), an excessive model adaptation to the training data, learning small sample specifics of the data which are not relevant for the whole data distribution.

Variance-based feature selection involves ranking the input features based on their variance and selecting only the most varying features Kumar and Minz (2014), Chandrashekar and Sahin (2014), Li et al. (2017). If the feature variance is low, then the feature is practically constant. As it does not vary from instance to instance, it cannot explain the variation of the target variable.

Principal component analysis transforms a high-dimensional dataset into a lower-dimensional space by identifying principal components, i.e., linear combinations of the original features, mutually orthogonal Wold et al. (1987), Kurita (2019), Maćkiewicz and Ratajczak (1993), Abdi and Williams (2010). PCA captures the data's directions of maximum variance. It ranks components by explaining variance, with the first capturing the most, followed by orthogonal components explaining the remaining variance. To reduce the dimension, n most important features are selected, where n corresponds to the desired dimension of the space.

2.2.2. ML models

The ML model contains parameters, and the learning consists of tuning the model parameters to minimize the prediction error. Besides parameters, ML models have tunable hyperparameters that configure the model (e.g., the depth of a decision tree) (Feurer and Hutter, 2019; Yang and Shami, 2020; Shahhosseini et al., 2022). During the training of ML models that involve tunable hyperparameters, a portion of the training dataset is allocated for a validation set to evaluate the hyperparameter selection. Cross-validation (Stone, 1978; Hastie et al., 2009; Berrar, 2019) is a technique where the dataset is divided into subsets to train and validate the model multiple times, providing a more reliable estimate of a model's performance by leveraging a larger validation set. Training attempts to produce models that generalize well, i.e., accurately predict labels on unknown instances.

In the case of regression ML models, the model prediction error is usually measured using the mean squared error (MSE) and root mean square error (RMSE) (Wang and Bovik, 2009; Tuchler et al., 2002). MSE computes average squared differences between predicted and actual values. RMSE computes the square root of the MSE, offering the advantage of being measured on the same scale as the original data. Another important metric for assessing the quality of the regression ML model is the coefficient of determination. The coefficient of determination (known as R^2 score) (Alpaydin, 2020; Nagelkerke, 1991; Hahn, 1973) is computed as $1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \mu)^2}$, where y_i refers to the target labels, $\hat{y}_i$ refers to the ML model predictions, and μ denotes the mean of the target labels y_i. R^2 score represents the proportion of the variance of the target variable that is "explained" by the regression ML model.

In the ML model training process, instance weights can prioritize the individual instances, focusing the ML model to more accurately predict instances with higher weights. This way, the ML model can prioritize critical samples and improve performance in particular areas of interest. Using instance weights alters the interpretation of MSE and RMSE. They no longer uniformly measure the average deviation of predictions from actual values across all the instances. Instead, they assign greater importance to instances with larger weights. The weighted MSE and RMSE provide a more specific evaluation of model quality, considering the varying importance of different instances as reflected by respective weights.

We discuss the three most commonly used ML models in the context of static profilers: decision tree (DT) (Rokach and Maimon, 2007), gradient boosting (Bentéjac et al., 2021), and neural network (NN) (LeCun et al., 2015).

Decision tree model uses a tree-like structure to model the data. The tree-like structure consists of nodes, branches, and leaves. Each DT node examines one of the features and, based on its value, propagates decision-making down the tree. Leaf nodes predict outcome values. The main benefit of using a DT is that they are interpretable (Breiman et al., 1984; Friedman et al., 2001). Another advantage of DT is that they are fast and easy to train and use (Brijain et al., 2014).

Fig. 3 (left) illustrates one shallow DT that predicts execution probabilities of code segments and examines only three features that characterize the code segments: loop depth, assembly size, and the number of CPU cycles. For example, suppose there is one code segment at the top of a function (i.e., at the loop depth of 0) with an assembly size of 5 bytes. In that case, the DT from Fig. 3 will predict its execution probability to be 0.16.

Overfitting can occur if the tree is too complex and fits the training data too closely, leading to poor performance on new, unseen data. Pruning (Breiman, 2017) helps simplify a model and involves removing branches after the tree has been built. Post-pruning iteratively removes the subtrees with the most negligible impact on the model performance. The minimal cost complexity post-pruning is controlled by the pruning parameter cpp_α. This parameter represents the trade-off between the DT's accuracy on the training data and its simplicity, with larger cpp_α values resulting in more aggressive pruning and simpler trees.

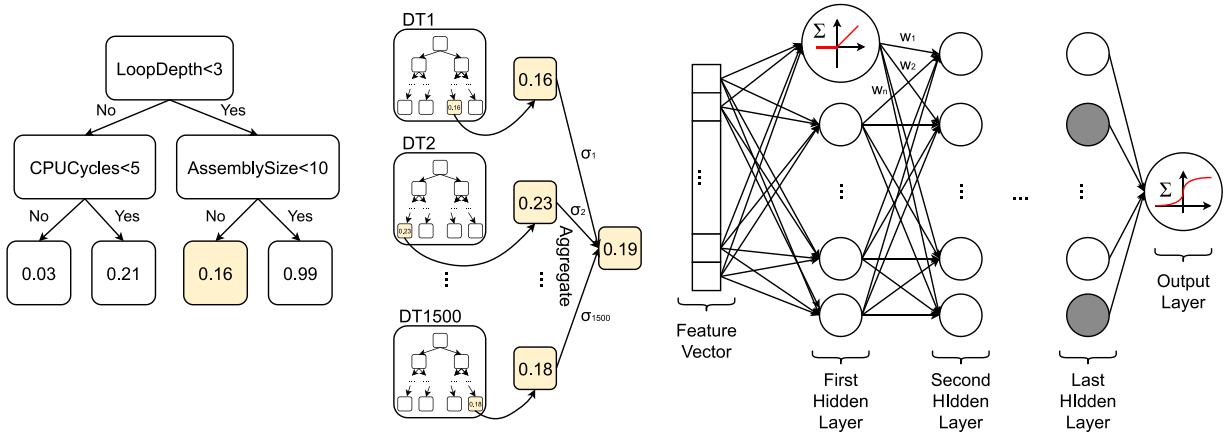


Fig. 3. DT (left), XGBoost ensemble (middle), and DNN (right) model architectures.

DT can be used to rank the input features with respect to their predictive power (Aggarwal, 2014). Gini importance (Lerman and Yitzhaki, 1984) of the feature is the (normalized) overall impurity reduction achieved when the DT model employs a splitting criterion based on that feature.

The Gradient boosting model is an ML ensemble model. Ensembles are groups of multiple ML models that collectively make decisions, by aggregating (possibly with weights) the predictions of the individual models from the ensemble. Ensembles are usually among the best models as model aggregation reduces variance, leading to more stable and reliable predictions (Sagi and Rokach, 2018; Barbez et al., 2020). Gradient boosting iteratively adds new models to the ensemble, fitting each new model to the residuals (i.e., the errors) of the previous models. This way, the ensemble gradually improves its performance by focusing on the samples that are hardest to predict. XGBoost (Chen et al., 2015) is a scalable and flexible implementation of the gradient boosting algorithm. In the case of XGBoost, the ensemble is built of decision trees (Fig. 3, middle).

Neural network model consists of multiple layers of neurons that transform an input vector into an output vector. All network layers except the output layer are named *hidden*. The NN with more than one hidden layer is called a *deep neural network* (DNN)[1] (Hassoun et al., 1995). The last layer in the network is called *the output layer*.

Each layer in a network comprises multiple neurons connected to all the neurons in the previous layer. Each neuron from the hidden and output layers computes a weighted sum of its inputs, which is then passed through an activation function to produce output. The activation function is a non-linear function that introduces non-linearity to the model, allowing it to learn complex patterns and relationships in the data. Most commonly used activation functions include the Rectified Linear Unit (ReLU) activation function ($relu(x) = max(0, x)$) and the sigmoid activation function ($\sigma(x) = \frac{1}{1+e^{-x}}$, which fits the model prediction in the range of $0-1$) (LeCun et al., 2015). Weights associated with a neuron determine the strength of its connections to other neurons. Optimization algorithms tune the model weights during the model training (Bottou et al., 2018; Amari, 1993). Adaptive moment estimation optimizer (Adam) (Kingma and Ba, 2014) is among the most commonly used optimization algorithms. It provides both adaptive learning rates (Dauphin et al., 2015) and momentum (Liu et al., 2020, 2019) for efficient and faster convergence during the training of DNN models.

Dropout (Srivastava et al., 2014) is a regularization technique in which randomly selected neurons are temporarily ignored or "dropped out" during DNN training. Dropout prevents overfitting and improves model generalization performance.

Fig. 3 (right) illustrates a regression DNN model that predicts a single real number. Its output layer consists of only one neuron. The neurons from hidden layers use the ReLU activation function, while the neuron from the output layer uses the sigmoid activation function.[2] We illustrate dropped-out neurons as gray-colored neurons.

2.3. Graal compiler

The Graal compiler (Wimmer et al., 2019) is a modern ahead-of-time compiler written in Java that compiles Java bytecode into standalone executables, which can run directly on an operating system without requiring a Java runtime. The Graal compiler can compile and optimize applications written in Java-bytecode languages such as Java, Scala, Kotlin, Groovy, etc. Graal compiler uses a high-level, graph-based intermediate representation named Graal IR.

PGO in the Graal compiler enables creation of highly optimized binaries. The Graal compiler extensively employs PGO across its more than a hundred compiler phases (Mosaner et al., 2022). The Graal compiler implements profile-driven method inlining (Scheifler, 1977; Ayers et al., 1997; Detlefs and Agesen, 1999) that incrementally explores the call graph of a program and alternates between the inlining and other optimizations (Wimmer et al., 2019; Prokopec et al., 2019a). The Graal compiler uses profiles to perform the dominance-based duplication simulation (Leopoldseder et al., 2018b; Leopoldseder, 2017), addressing the challenge of identifying optimization opportunities before performing any code transformations. This way, the Graal compiler benefits from effectively employing conditional elimination (Stadler et al., 2013), partial escape analysis and scalar replacement optimizations (Stadler et al., 2014), constant folding, and read elimination (Aho et al., 2007). In addition to the aforementioned optimizations, the Graal compiler also uses profiles in standard optimizations (Kennedy and Allen, 2001; Aho et al., 2007) such as vectorization, loop peeling, loop unrolling, and code layout (Pettis and Hansen, 1990).

Graal IR (Duboscq et al., 2013; Leopoldseder et al., 2015) is a sea-of-nodes-based intermediate representation (Demange et al., 2018) that uses static single-assignment form (Cytron et al., 1991; Masud and Ciccozzi, 2020). Graal IR is structured as a directed graph and therefore simplifies implementing and debugging standard and aggressive compiler optimizations (Duboscq et al., 2013). Also, graphs remove redundancy and show improved information locality when contrasted with syntax trees (Bračevac et al., 2023). Nodes in Graal IR can be

[1] Today, there are neural networks with more than 100 layers (He et al., 2016; Wu et al., 2019). Therefore, the definition of a *deep* neural network has become relative.

[2] Although the sigmoid activation function is usually used in the context of classification ML models, it can be used in regression tasks.

floating or *fixed*. The floating nodes represent the data flow (such as addition, subtraction, multiplication, division, bitwise operations, type conversions, and more) and are responsible for propagating values between instructions. Floating nodes can take inputs from other floating nodes or constants embedded in the program and produce output values used by subsequent instructions. Fixed nodes represent function calls, memory loads and stores, bytecode exceptions, control flow constructs, etc. *Merge* nodes are fixed nodes that merge control flow paths within the structured graph. They represent a point in the program where multiple control flow paths converge into a single path. *Phi* nodes merge floating nodes and represent a value that can come from multiple possible sources. *Pi* nodes represent a value that has been narrowed down to a more specific type. As Graal IR is a structured graph, loops can only be entered at their beginning, marked by a fixed *LoopBegin* node. Similarly, the Graal compiler marks the exit from the loops with the *LoopExit* nodes.

The Graal compiler parses the branching statements in the program (for example, `If` statements) to the *control-flow split nodes* (*cfs*-nodes) in the Graal IR graph. *Cfs*-nodes are nodes whose out-degree is greater or equal to two. The edges that exit these nodes will be referred to as branches. A *cfs*-node from the Graal IR corresponds to the conditional branch instruction in the three-address code IR (for example, in LLVM IR (Lattner, 2008; LLVM Project, 2024)).

To create the control-flow graph (CFG), the Graal compiler does the scheduling of the floating nodes around the fixed nodes and assigns them to the CFG blocks. Branches of the *cfs*-nodes in the Graal IR correspond to the edges between blocks in the CFG.

3. The design of GraalSP

GraalSP predicts execution probabilities for branches of binary *cfs*-nodes, simplifying the prediction task to the prediction of the execution probability p_1 of the first branch. The terms "the first branch" and "the second branch" refer to the order in which the branches appear after parsing the code. GraalSP calculates the execution probability of the second branch as $p_2 = 1-p_1$. GraalSP uses uniform distribution to model the execution probabilities of branches corresponding to non-binary *cfs*-nodes. This simplifies the ML model training and has also been done in previous research (Ball and Larus, 1993; Calder et al., 1997; Moreira et al., 2021; Rotem and Cummins, 2021). This section provides a detailed overview of GraalSP and includes a dataset creation process, specifics of ML models, and a description of static branch probability prediction heuristics.

3.1. Dataset creation

The Graal compiler parses Java bytecode to create the Graal IR and a corresponding CFG of a program. Fig. 4 shows the process of dataset creation. In the first stage of dataset creation, we extract features that characterize the branches from the Graal IR. Also, we collect profiles via instrumentation-based dynamic profiling (Rotem and Cummins, 2021; Calder et al., 1997) as this approach gives the best-quality profiles. In the second stage of the dataset creation process, we use collected profiles to label extracted features. This way, we create the dataset for the training of the ML model that will drive the static profiler.

For each binary *cfs*-node, we characterize its first branch by features that capture the specifics of (i) the *cfs*-node, (ii) the block B that hosts the *cfs*-node, (iii) the predecessors' blocks of the block B, (iv) the blocks dominated by the CFG blocks on which CFG edges from the block B point to.

3.1.1. Static features

We propose a feature set that consists of 11 numeric features and 5 composite features, which we encode to vectors of integers. We list all the features in Table 1. Regarding the composite features, three features are categorical and two features are in the form of a map where keys are Graal IR node types and values are integers.

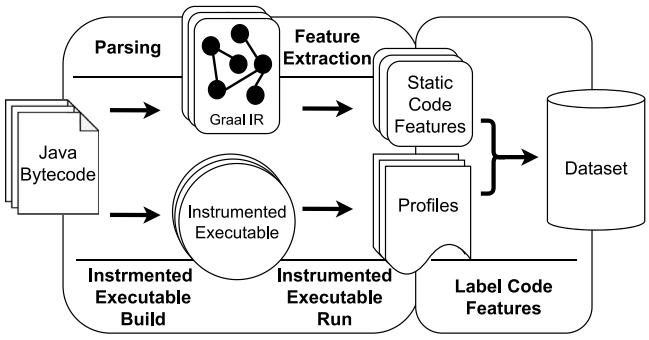


Fig. 4. Dataset creation process.

Standard features. The features 1–10 from Table 1 are adopted (Calder et al., 1997; Moreira et al., 2021; Desmet et al., 2005; Rotem and Cummins, 2021). Most of them are numerical. We use feature 4 to characterize the size of the branch, and features 5–10 to characterize the control flow of the program.

Features 1–3 from Table 1 are categorical. The `CSType` feature characterizes the type of the *cfs*-node and can be `IfNode` or `SwithcNode` as those nodes can indicate the branching in the Graal IR. We use the `CSBranchType` feature to characterize the logic operation that indicates the branching, e.g., checking if a number is negative (< 0) or non-positive (≤ 0). There are 14 different such operations. For each branch in the Graal IR, the `Header` feature characterizes the type of the branch: branch can start with the `LoopExit` node or `Begin` node.

Adapted features. CrystalBall (Zekany et al., 2016) parses features from the LLVM IR blocks. They count the different LLVM instruction types within the CFG blocks. As we rely on the Graal IR, which is graph-based, we adapt this idea by counting different kinds of nodes within the CFG blocks (Table 1, features 11 and 12). We use the `NodeCountMap` feature to track the fixed nodes and capture program control flow, and the `FNodeCountMap` feature to track the floating nodes and capture the data flow within the program. This way, we avoid crafting meaningful features manually. Widely used features such as the number of *load* and *store* instructions (Rotem and Cummins, 2021; Moreira et al., 2021) are covered by the values of the `Load` and `Store` nodes that are stored within the `NodeCountMap` feature. Similarly, the number of function calls (Rotem and Cummins, 2021; Moreira et al., 2021) and the number of `Phi` instructions in a block (Rotem and Cummins, 2021) are covered by the corresponding entries in the `NodeCountMap` and `FNodeCountMap` map features.

Novel features. We define new features that, based on the high-level Graal IR precisely approximate low-level program characteristics (Table 1, features 13–16). Leopoldseder et al. (2018a) propose a static cost model that estimates the number of CPU cycles required for the execution of a node and the node's assembly size. Relying on this cost model, we define an approximation of the required CPU cycles and assembly size of the block hosting the *cfs*-node and the dominated blocks that characterize the CFG edges. Furthermore, we enhanced the feature set by counting the number of nodes that are estimated as CPU cheap (take 0 or 1 CPU cycles) or CPU expensive (take more than 64 CPU cycles).

3.1.2. Features encoding

We employ one-hot encoding of categorical features as our categorical features take a small number of categories. We encode `NodeCountMap` and `FNodeCountMap` features as vectors of integers. The length of a vector corresponding to `NodeCountMap` is 165 (the number of different node types in the Graal IR). The length of a vector corresponding to the `FNodeCountMap` feature is 56 (the number of different floating node types within the Graal IR). We preserve node

Table 1
Static features that characterize a branch. For categorical features, the number of different categories is given in brackets. For map-based features, the maximal number of map entries is given in brackets. All numeric features are integers.

#	Feature name	Kind	Type	Target	Description
1.	`CSType`	Standard	Categorical (2)	*cfs*-node	Type of a *cfs*-node
2.	`CSBranchType`	Standard	Categorical (14)	*cfs*-node	Type of the node that determines the branching direction
3.	`Header`	Standard	Categorical (2)	Dominated blocks	Type of the first node in the dominated blocks sequence
4.	`BlocksCount`	Standard	Numeric	Dominated blocks	Number of dominated blocks
5.	`CSDepth`	Standard	Numeric	*cfs*-node	The number of blocks containing *cfs*-nodes that have a successor node that dominates the block containing this *cfs*-node
6.	`MaxCSDepth`	Standard	Numeric	Dominated blocks	Maximum `CSDepth` of the *cfs*-nodes within the dominated blocks
7.	`LoopDepth`	Standard	Numeric	Block, Dominated blocks	The number of nested loops that enclose the given block/dominated blocks
8.	`MaxLoopDepth`	Standard	Numeric	Dominated blocks	Maximum `LoopDepth` within the dominated blocks
9.	`DominatorDepth`	Standard	Numeric	Block	Depth of a block in the dominator tree
10.	`PCount`	Standard	Numeric	Block	The number of predecessors
11.	`NodeCountMap`	Adapted	Map (165)	Dominated blocks	Number of occurrences of each node
12.	`FNodeCountMap`	Adapted	Map (56)	Block, Predecessors blocks	Number of occurrences of each floating node
13.	`AssemblySize`	Novel	Numeric	Block, Dominated blocks	Estimated assembly size
14.	`CPUCycles`	Novel	Numeric	Block, Dominated blocks	Estimated number of CPU cycles
15.	`CPUCheap`	Novel	Numeric	Block, Dominated blocks	Number of CPU cheap nodes
16.	`CPUExpensive`	Novel	Numeric	Block, Dominated blocks	Number of CPU expensive nodes

order by sorting the nodes in each vector lexicographically. This way, each node type has a unique position in the encoding vector.

The usage of composite features yields high-dimensional feature vectors: once the categorical features are one-hot encoded and the maps are transformed into vectors, we concatenate these vectors with the numerical features and obtain a feature vector of length 468. To improve ML model training, we have to preprocess feature vector because of its high dimensionality (Dhal and Azad, 2022; Kotsiantis et al., 2006; Moreira et al., 2021).

The usage of composite features confers the advantage of ensuring the sustainability of the ML pipeline. The Graal compiler, as a commercial product, keeps evolving, and improvements of the compiler are reflected in adjustments to the IR. Adjustments of the IR involve adding new nodes, deleting existing nodes, and changing the roles of existing nodes (e.g., a change in optimization can alter the way optimization utilizes IR nodes). By using composite features, the ML model can undergo regular training intervals during which feature selection consistently identifies the most important features. This approach imparts resilience to the ML pipeline against changes in the IR, obviating the necessity for manual crafting of features whenever the IR undergoes modifications.

3.1.3. Data labeling

To label data, we calculate branch execution frequencies using the branches' execution counts collected through instrumentation-based profiling. If the first branch is executed n_1 times and the second branch is executed n_2 times, then the execution frequency of the first branch is calculated as $\frac{n_1}{n_1+n_2}$.

3.2. ML models for branch probability prediction

The ML model that guides the static profiler should correctly predict probabilities of branches that lead to frequent paths, as they highly influence the performance of the optimized executable. To focus the models on more frequent branches, we use *normalized branch weights*. We calculate a normalized branch weight as the ratio of the *cfs*-node execution count and the execution count of the most frequently executed *cfs*-node in the benchmark. We calculate the execution count of a *cfs*-node by summing the number of times its branches are executed.

We achieved the best results using DT, DNN, and XGBoost ML models. Besides these three models, we also experimented with other ML models, such as support vector machines and the k nearest neighbors algorithm.

Reducing the number of features. To reduce the number of features, for the DT and XGBoost model training, we employ the variance-based feature selection. Our pipeline requires dimensionality reduction as, with composite features, we end with a high-dimensional feature vector. We opt for feature selection to preserve the original feature space as done in previous works that use the DT-based models (Calder et al., 1997; Rotem and Cummins, 2021). We remove features with a variance of less than 0.5. We choose the threshold value empirically via cross-validation of the DT and XGBoost models. This way, we got the feature vector of dimension 78. We use the PCA algorithm to select the 78 most informative data components for DNN training. We choose to use the 78 PCA components to follow the number of features from the first two models.

Decision tree model. We do the tree post-pruning, with the pruning parameter $cpp\_\alpha = 10^{-6}$, to avoid overfitting. We experimentally determined the concrete parameter value based on the model's performance on the validation set.

Gradient boosting model. We train the XGBoost ML model that predicts branch probability by averaging the predictions of 1500 DTs. We experimentally choose to have 1500 DTs in the ensemble, each with a maximum depth of 10. An increase in depth results in overfitting, whereas constraining the maximum depth to 9 or below does not adequately fit our training set.

Deep neural network. We use DNN that comprises six hidden layers with 512, 512, 512, 256, 256, and 256 neurons, respectively. The neurons in the hidden layers employ the ReLU activation function. The output layer consists of a single neuron using the sigmoid activation function. We use PCA to preprocess the features and we train the DNN with the 78 input features. The proposed DNN has 832 513 parameters. We trained the DNN model using the Adam optimizer. To prevent overfitting, we employed dropout on the last two hidden layers, randomly turning off 5% of the neurons from those layers. We trained the model using the MSE loss. We also tried the binary cross-entropy loss functions but empirically chose to use the MSE.

3.3. Branch probability prediction heuristics

The fundamental premise of machine learning is that the user input data will adhere to the same distribution as the data used for model

training. However, outliers can always occur (Ghosh and Vogt, 2012). This becomes particularly important when an ML model is used in a commercial product. To make GraalSP robust, we enhance it with two branch probability prediction heuristics, which handle the situations when the ML model faces outliers. We chose experimentally the exact values used in the heuristics.

The loop heuristic ensures that a branch leading to a loop body always has a predicted probability higher or equal to 0.20. If the ML model predicts the probability to be lower than 0.20 the loop heuristic will correct the prediction and set it to 0.20. This heuristic ensures that GraalSP will not inadvertently cut optimizations from the hot code segments. The hot code segments refer to code segments that will be executed frequently when the program runs.

The dead-end heuristic ensures that every branch that leads to blocks that terminate the program has a predicted probability lower or equal to 0.80. This heuristic applies to a branch only when the opposite branch leads to blocks with an estimated assembly size of at least 50 bytes. The goal of this heuristic is to avoid favoring a branch that leads to program termination when an alternative branch continues with program execution.

Note that these heuristics represent a trade-off between a compiled program's size and performance. For example, ensuring that a loop body probability is at least 0.20 avoids making mistakes by predicting a low execution probability to a frequent loop. On the other hand, this leads to more aggressive code duplication within each loop body (including the bodies corresponding to cold code segments), resulting in a larger binary size. Therefore, these heuristics bring performance gain but at a cost of a larger binary size.

4. Implementation

We have developed GraalSP and integrated it into the Graal compiler 23.0.

4.1. Training and validation datasets

We created the training dataset out of 44 real-world Java applications from the open-source GraalVM Reachability Metadata Repository, provided by Oracle (Oracle Corporation, 2023).[3] GraalVM Reachability Metadata Repository includes various applications, for example, database libraries, data compression libraries, logging libraries, networking libraries, web servers, HTTP protocols, and emailing support. This way, we avoid the problem of training datasets that often rely on benchmark data rather than real-world projects, which leads to lower-quality static profilers (Moreira et al., 2021).

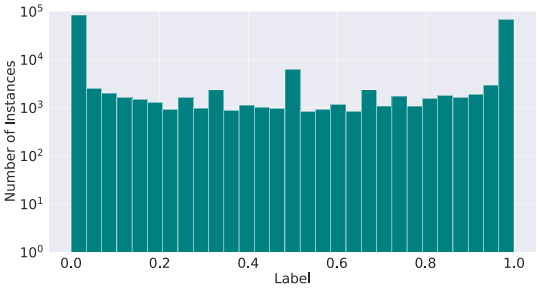
GraalVM Reachability Metadata Repository contains multiple versions of the same libraries. For example, there are two different versions of the `com.mysql` library (8.0.29 and 8.0.31). As these two library versions share the majority of the code, we do not use the older benchmark version in dataset creation. Using multiple versions of the same library in dataset creation can be interpreted as data augmentation (Van Dyk and Meng, 2001; Yu et al., 2022) that would focus the model on the replicated instances. Therefore, we ensure that each library is represented only by its newest version. We list the used libraries in Table 10 B.

We create the training set that set consists of 323 350 instances. We label only the first branch corresponding to binary *cfs*-nodes. Also, we discard the branches of binary *cfs*-nodes corresponding to the bytecode exceptions as these branches have very low execution probabilities, which are hardcoded in the Graal compiler.

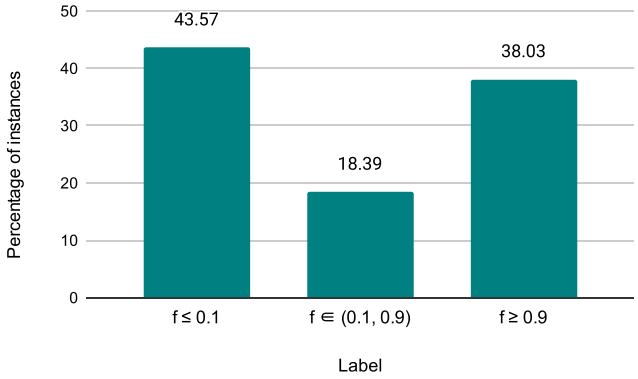
Table 2 presents the main characteristics of the distributions of the labels in our training dataset. That is the data distribution we intend to learn. The distribution exhibits strong symmetry, indicated by a skewness[4] of approximately 0.1, and close alignment between the mean and median values. Fig. 5(a) visualizes the distribution of training labels. The distribution confirms expectations that during execution, in most cases, only one of the *cfs*-node's branches is taken when the program runs (Ball and Larus, 1993). However, as shown in Fig. 5(b), 18.39% of instances are labeled with execution frequency that is not close to 0 nor 1. This justifies our decision to use regression rather than classification.

Table 2
Statistics of the distribution of labels from the training dataset.

Statistic Name	Value
Mean	4.7315×10^{-1}
Standard deviation	4.5774×10^{-1}
Skewness	1.0381×10^{-1}
Minimum	0.0
25th Percentile	$1.000000000029 \times 10^{-6}$
50th Percentile (Median)	$4.222222222222 \times 10^{-1}$
75th Percentile	$9.999990000000 \times 10^{-1}$
Maximum	1.0



(a) Histogram of the labels from the training set (logarithmic scale, 50 bins).



(b) Classification of the labels into three categories depending on the profiled branch frequency f.

Fig. 5. Visualizations of the labels in the training dataset.

Validation dataset. We employed cross-validation to fine-tune hyperparameters when training the DT and XGBoost ML models. Due to the computational complexity of the DNN model training, cross-validation is not effective. Therefore, we randomly selected 10% of the training dataset and employed it as the validation set when tuning the hyperparameters of the DNN model.

[3] The test set is made out of the standard Graal compiler benchmarks and is entirely separate from the training set (Section 5.1.2).

[4] Skewness is a statistical measure that describes the asymmetry or lack of symmetry in a probability distribution. It indicates the degree to which a distribution deviates from being perfectly symmetrical (for example, the normal distribution $\mathcal{N}(\mu, \sigma)$ is symmetric with zero skewness).

Table 3
Statistics of the instance weights from the training dataset.

Statistic Name	Value
Mean	1.3515×10^{-4}
Standard deviation	1.6963×10^{-3}
Skewness	$2.2889 \times 10^{+1}$
Minimum	1.6038×10^{-10}
25th Percentile	1.4748×10^{-8}
50th Percentile	6.0001×10^{-8}
75th Percentile	4.0563×10^{-7}
90th Percentile	4.4665×10^{-6}
95th Percentile	2.5704×10^{-5}
Maximum	1.1431×10^{-1}

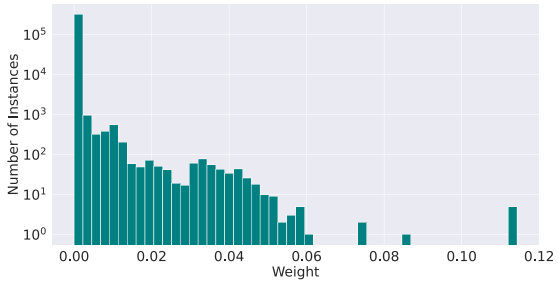


Fig. 6. Histogram of the normalized instances' weights from the training dataset (logarithmic scale, 50 bins).

4.2. Instances' weights

Table 3 presents the main characteristics of the distribution of the weights in the training dataset while Fig. 6 visualizes this distribution. The distribution is highly positively skewed, with a long tail on the right side. The leftmost histogram bin from Fig. 6 counts weights smaller than 0.002 and contains 320212 instances, making up 99% of the training dataset. The significant difference between the 95th percentile and the maximum value indicates a few extremely high values. Instances corresponding to those weights are not outliers; on the contrary, those are the instances we want to predict as accurately as possible. For example, the k-means (Lloyd, 1982; MacQueen et al., 1967) algorithm is an iterative algorithm whose performance is notably affected by predictions of execution probabilities for its main loops. Branches corresponding to those loops are frequently executed and therefore have assigned high weights.

4.3. ML models implementation

We use Python and the following implementations:

- SciKit Learn Library, version 1.1.2 (Pedregosa et al., 2011) for feature preprocessing and the DT model,
- DLMC XGBoost Library, version 1.6.1 (Chen and Guestrin, 2016) for the XGBoost model,
- PyTorch ML framework, version 1.12.1 (Paszke et al., 2019) for the DNN model.

We trained the models using a laptop with 6-core Intel(R) Core(TM) i7-9850H CPU@2.60 GHz, 32 GB DDR4 RAM.

The automatization process of ML model training is important (Steidl et al., 2023). We use the GraalVM continuous integration infrastructure to automatize dataset creation and ML model training. The process is executed regularly (once a month) to keep the ML model up-to-date.

Using composite features while regularly recreating the dataset and retraining the ML model makes the ML pipeline resistant to changes in the IR (Section 3.1.2). Java community constantly updates the GraalVM Reachability Metadata Repository with new programs and the latest versions of libraries. Therefore, using programs from this repository as the training programs in combination with the automatization of the dataset creation and ML model training enables improving the commercial usability of GraalSP over time and its performance on fresh programs. To label the instances from the dataset, we take advantage of the instrumentation-based profiler integrated into the Graal compiler.

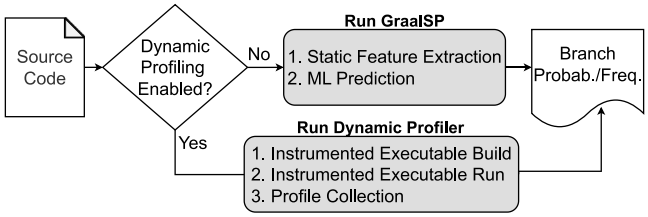


Fig. 7. Integration of the dynamic profiler and GraalSP in the Graal compiler.

4.4. ML models deployment

To include the trained ML model in the Graal compiler, we use the ONNX Java Runtime, version 1.12.0 (Microsoft Corporation, 2021) supporting the Windows, Linux, and Darwin OS distributions. We implemented feature extraction and branch probability prediction as phases in the Graal compiler compilation pipeline.

We do the feature extraction and ML model predictions after the code parsing phase and the creation of the Graal IR graph. This way, we enable as many phases as possible to perform PGO. It is also possible to clone feature extraction and branch probability prediction phases and inject them at different stages of compilation. As the IR changes during compilation, this leads to more accurate profiles at later compilation phases. However, the experimental evaluation showed just a slight increase in efficiency while the compile-time overheads increased.

We simultaneously predict probabilities for all branches within a single method, which reduces the number of ML model inference calls and overall compile time. This choice of predicting probabilities at the method level aligns with the Graal compiler's concurrent image-building process, where methods serve as a concurrency unit.

Fig. 7 shows an integration of GraalSP with dynamic profiling. If dynamic profiling is enabled, we instrument the code and run the instrumented image to collect profiles. The Graal compiler will use the profiled branch frequencies if the application build includes dynamically obtained profiles. Otherwise, the compiler will do optimizations based on predicted probabilities.

4.5. GraalSP-PLog for analysis of predicted profiles

Debugging the predictions of a static profiler is challenging: even when running a simple *HelloWorld* program, the execution visits 1689 *cfs*-nodes. The predicted execution probability of each branch within these control splits influences the program's performance. To help developers inspect GraalSP predictions and isolate particular examples of correct and incorrect predictions, we have developed *GraalSP Profiles Logger* (GraalSP-PLog). The GraalSP-PLog compiles the input program, runs GraalSP, saves predictions of the ML model and corrections from static heuristics, performs instrumentation profiling, collects profiles, and generates the report.

For each *cfs*-node in the input program, the GraalSP-PLog reports its position in the source code, the method in which it is located, the declaring class of that method, the predicted probability of its first branch, the profiled execution frequency of its first branch, and its execution count. As the execution count can vary significantly depending on the number of loops in an application, to facilitate interpretation, the tool normalizes execution counts, ensuring that the normalized execution count of all nodes in the application sums up to 1. The

tool arranges nodes in the report in decreasing order according to their execution count. As nodes with higher execution counts have a higher impact on the overall performance of the program, this way the logging enables easy identification of the performance-critical correct and incorrect predictions.

To track the impact of branch execution probabilities on the decisions in optimization phases, we have integrated Ideal Graph Visualizer (IGV) (Würthinger, 2011; Oracle Corporation, 2024) in GraalSP-PLog. IGV visually represents changes in the IR graphs throughout the compilation phases, making it easier to understand the impact of optimization phases on method compilation. The users of GraalSP-PLog can specify a regular expression, and the tool will plot IR graphs of all functions whose names match the specified regular expression. This visualization feature enhances the human evaluation of GraalSP.

5. Evaluation

We evaluate the proposed feature set, the trained ML models, and the impact of integrating GraalSP into the Graal compiler. The impact of PGO on execution time speedup is the most important metric of the profiles' quality. In addition to execution time speedup, the binary size and compile-time are also important metrics, as it is not difficult to gain speedup if these are not considered.[5] Therefore, our evaluation addresses the program's runtime, binary size, and compile time.

5.1. Evaluation setup

We conducted all experiments using the same evaluation setup which we describe in this section.

5.1.1. Software versions and hardware configuration

We use the GraalVM Enterprise Edition 23.0 based on Java 17 (*labsjdk* enterprise edition 17 *jvmci* 23.0) running on the Oracle Linux Server 7.4, kernel version Linux *4.1.12-112.14.13.el7uek.x86_64*. We perform the experiments on the machine cluster *X5* node equipped with two Intel *E5-2699 CPUs @ 2.30 GHz* 18-core processors, 64K *L1* cache, 256K *L2* cache, 46080K *L3* cache, 512 GB *DDR4-1600* memory, *LSI MegaRAID SAS-3 3108* and *Mellanox Connect-X Infiniband* cards.[6]

We disable the turbo-boost, bind the CPU to the first core, and use the memory of that core to reduce the instability of measurements due to the varying CPU frequency and the non-uniform memory access (Kleen, 2005; Beyer et al., 2019). We run the benchmarks with the hyper-threading enabled and mount the RAM disk with a size of 40 GB (to avoid the IO nose). We set the CPU C-states to 0 to avoid CPU power-saving modes. We use Serial Garbage Collector from the Graal compiler.

5.1.2. Test benchmarks and test dataset

To evaluate the ML models and GraalSP, we use the standard Graal compiler benchmark suites:

- The Renaissance suite (Prokopec et al., 2019b) is a modern and diverse benchmark collection that contains numerous authentic workloads from different programming paradigms. These include concurrent, functional, object-oriented, big-data processing, message-passing, stream processing, and ML. It consists of various benchmarks, for example, optimizing genetic algorithms through Jenetics (Wilhelmstötter, 2021) or emulating intense server loads using Twitter Finagle (Twiter Inc, 2023).
- The DaCapo suite (Blackburn et al., 2006) consists of open-source, real-world applications with non-trivial memory loads. For example, the simulation of processes on a grid of AVR microcontrollers (Kunikowski et al., 2015), creating PDF files, indexing and searching large sets of documents, and tasks related to source code analysis.
- DaCapo con Scala suite (Sewe et al., 2011) is an expansion of the DaCapo suite, focusing on assessing the efficiency of Scala programs on JVMs. It contains libraries for language processing, Scala compilers and class file decoders, a code formatter, and XML data-binding tools.

Table 11 B lists all the test benchmarks. As can be seen from the table, the training data and the test data do not intersect.

To evaluate the quality of ML models, we created a test dataset consisting of 198 157 instances. In addition to evaluating the ML models on the test dataset, we also evaluate the impact of a static profiler on compiled and optimized programs from the test benchmarks.

When aggregating the results over a benchmark suite, we scale the results to the default configuration of the Graal compiler and aggregate scaled values using the geometric mean (Manikandan, 2011). We use the geometric mean since it is the preferred method for aggregating scaled results (Fleming and Wallace, 1986).[7]

Regarding the evaluation of runtime performance impact, we ran the benchmarks in isolation and averaged multiple benchmark runs to improve the accuracy of the results. We use the default number of runs of each benchmark established in the CI of the Graal compiler and prior research (Bruno et al., 2021). The number of runs is tuned so that the benchmark performs more runs if it is unstable (due to non-determinism, concurrent execution, etc.). The number of runs and the execution time variance for each benchmark is given in Table 11 B.

5.2. Crafting GraalSP

We evaluate the features and the quality of trained ML models to create the best version of GraalSP.

5.2.1. Most relevant features

We used the Gini importance and the DT model to measure the features' importance. Fig. 8 shows the 15 most valuable features.

Our evaluation shows that the proposed static cost model features are valuable in static branch probability prediction. In particular, the estimated assembly size of the block that hosts the *cfs*-node is ranked as the most valuable feature, while the estimated number of CPU cheap instructions and CPU cycles in the block that hosts the *cfs*-node are ranked as the 3rd and the 6th most valuable features.

The feature importance analysis emphasizes valuable features from the `NodeCountMap` and `FNodeCountMap` maps. For example, DT ranks as the 2nd most valuable feature the number of the `ParameterNode` nodes in the predecessors' blocks of the block that hosts the *cfs*-node and as the 4th most valuable feature the number of these nodes in the block that hosts the *cfs*-node. The `ParameterNode` node represents a method's parameter and is scheduled to the first block of the function's CFG. Therefore, the importance of these features shows that it is essential to distinguish *cfs*-nodes from the top of a function.

In addition to the number of the `ParameterNode` nodes, the `NodeCountMap` and `FNodeCountMap` maps suggest other valuable features. The number of the `LoopExitNode` nodes within CFG blocks corresponding to the first branch of a *cfs*-node is ranked as the 8th most valuable feature. Similarly, the number of the `BeginNode` nodes and the number of the `EndNode` nodes within the CFG blocks corresponding to the second branch of a *cfs*-node are ranked as the 7th

[5] For example, a simple heuristic that inlines each function call speeds up the program's execution time, but it comes at the cost of a notable increase in the program's size and consequently in increased compile time.

[6] Our experiments can also be conducted on machines with much lower performances, including laptops.

[7] Arithmetic mean, in this context, can lead to wrong conclusions (Fleming and Wallace, 1986).

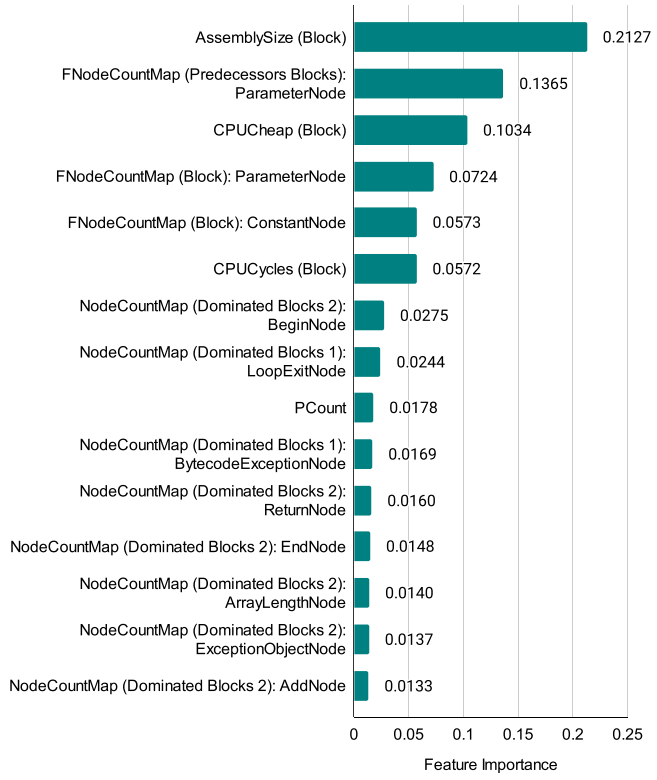


Fig. 8. Feature importance extracted from the DT model. A feature name is given with its target in parentheses. For the `NodeCountMap` and `FNodeCountMap` features, we also specify the map's key that the feature refers to.

and 12th most valuable features. The DT model uses these features to characterize a program's control flow.

Additionally, the number of nodes indicating program terminations, such as the number of the `BytecodeExceptionNode` nodes within the blocks that correspond to the first branch of a *cfs*-node and the number of the `ReturnNode` nodes and `ExceptionObjectsNode` nodes in blocks corresponding to the second branch of a *cfs*-node, are also valuable features. This aligns with other studies, which have manually extracted similar features (Rotem and Cummins, 2021; Moreira et al., 2021). Thus, besides discovering new features, the feature maps can confirm the relevance of previously proposed features.

5.2.2. Evaluation of the ML models

We evaluate the ML models to determine the best model that will drive GraalSP. Table 4 lists the weighted RMSE and R^2 score metrics values that ML models achieve on the test set. Furthermore, the table shows size, training time, and inference time[8] of the ML models.

The XGBoost ML model outperforms the DT and DNN models, with the lowest RMSE error and highest R^2 score on the test set. In addition, it is 10 times smaller than the DNN model. The model training and inference times also vary a lot. As anticipated, the DT ML model is the fastest to train, with a time of slightly over 4 min. In contrast, the DNN model takes nearly two hours to train, owing to its network of over 800 000 parameters. The XGBoost ensemble, comprising 1500 trees, requires approximately one hour of training. The prediction time of the XGBoost ML model is around 6 times shorter than the prediction time of the DNN model.

[8] We measure the inference time using a development laptop (same setup as for model training) to estimate realistic overheads users will have on their machines.

Table 4
Comparison of different ML models considering RMSE and R^2 score on the test set, model size, training, and inference time.

Model	RMSE	R^2 score	Size (KB)	Train time (s)	Infer time (s)
DT	0.3510	0.3812	19.9	243	0.1
DNN	0.3230	0.4758	3117.3	6459	3.0
XGBoost	0.2989	0.5512	289.5	3264	0.5

Table 5
Comparison of the static profilers considering their impact on runtime, binary size, and compile time. Values in the table are percentages compared to the baseline configuration.

Model	Runtime speedup	Binary size increase	Compile-time increase
DT	4.91	2.44	4.04
DNN	5.16	2.44	3.20
XGBoost	5.22	2.44	2.17

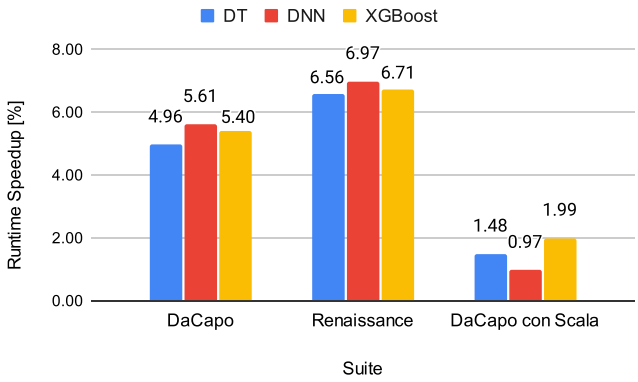


Fig. 9. Runtime speedup of profilers driven by the DT, DNN, and XGBoost ML models.

To evaluate the impact of incorporating an ML-based static profiler into the Graal compiler, we compare:

- as a baseline, the default version of the Graal compiler, which uniformly distributes execution probabilities over the branches,
- the Graal compiler which uses a static profiler driven by the DT model,
- the Graal compiler which uses a static profiler driven by the DNN model, and
- the Graal compiler which uses a static profiler driven by the XGBoost ML model.

Table 5 summarizes the results.

Runtime performance. We obtain the best improvement in the runtime performance of 5.22% with a static profiler based on the XGBoost model (Table 5). Fig. 9 summarizes the performance impact of the static profilers on test benchmark suites. Table 12 C presents the geometric standard deviation of the aggregated runtime speedup across benchmark suites. Note that all models perform much worse on the DaCapo con Scala benchmark suite, as that code is considerably older and diverges from the code on which the models were trained (real-world, up-to-date Java code). XGBoost generalizes better than the DNN model,[9] as, on the DaCapo con Scala suite, the XGBoost model outperforms the DNN model by approximately two times.

Considering data distribution in our training dataset, the *scala-doku*, *scala-kmeans*, *scalap*, and *scalaxb* benchmarks are outliers. Our ML models introduce slowdowns on these benchmarks ranging from

[9] In the context of ML, generalization refers to the model's ability to predict accurate labels of new, previously unseen data.

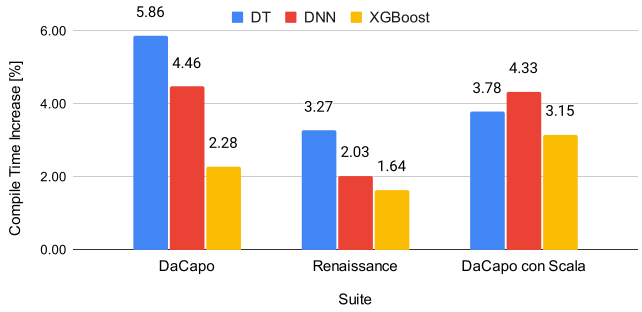


Fig. 10. Compile-time increase of profilers driven by the DT, DNN, and XGBoost ML models.

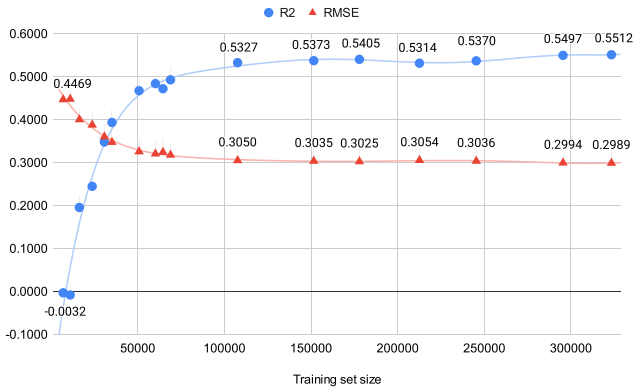


Fig. 11. Effect of the training set size on RMSE and R^2 score metrics in the test set.

0.27% to 30.48%. Although all outliers are Scala benchmarks, it is important to note that our models can perform well on Scala code. For example, the static profiler that uses the XGBoost ML model accelerates the DaCapo con Scala Kiama benchmark, a Scala Library for Language Processing (Sloane, 2009), by 13.54%.

Binary size. Concerning the impact on the binary size of compiled programs, all the ML models exhibit similar behavior, introducing an overall program size increase of 2.44% (Table 5). Trained ML models become part of the Graal compiler and not part of the optimized binaries. Therefore, the ML model size impacts the compiler's size and does not affect the size of the optimized binaries. The static profiler affects the size of an optimized binary as optimizations (e.g., duplication) are performed based on predicted branch probabilities. In the baseline compilation scenario, the Graal compiler assumes uniform probability distribution over the branches of the *cfs*-nodes, which results in less aggressive code duplication (Leopoldseder et al., 2018b) and smaller binary sizes. The disparity in the size of the resulting binaries for different ML models is negligible, lower than 0.01%. The observed increase in program size is 2.70% for the DaCapo suite, 2.59% for the Renaissance suite, and 1.88% for the DaCapo con Scala suite. While the ML models exhibit the lowest runtime performance increase on the DaCapo con Scala suite, they also contribute to the lowest increase in binary size on this suite.

Compile time. XGBoost model obtains the smallest increase in compilation time (Table 5). Fig. 10 shows the compile-time impact of the static profilers on the test benchmark suites, while Table 12 C shows the geometric standard deviation of the aggregated results across benchmark suites. The XGBoost model also introduces the smallest compile-time overheads on every suite. In the context of the ML-based static profilers, programs' compile time is affected by feature extraction, ML model inference, and the quality of profiles that trigger PGO. Regarding the compile-time increase, the main drawback of the DNN model is the expensive inference, as the DNN consists of over 800 000 parameters (Table 5). Concerning the DT model, its predictions come at a low cost. Yet, the rise in compile time for this model is magnified by the low quality of its predictions. Less accurate predictions direct PGOs towards different code sections compared to the XGBoost and DNN models.

5.2.3. Discussion

In this section, we discuss details regarding the training and quality of ML models: evaluation of the training set's size on model quality, the impact of the feature set size on model predictions and training time, the utilization of weights in the model training to favor important instances, and quality of the model predictions on different languages compiling to Java bytecode.

Assessing the effects of the training set size on model quality. Fig. 11 shows the effect of gradually adding programs to the training set on RMSE and R^2 score metrics values on the test set. Increasing the number of instances in the training set enables better model generalization and leads to a reduction in RMSE and an increase in R^2 score on the test set. This aligns with similar experiments conducted on classification models, where the classification error on the test set decreased with the enlargement of the training set (Calder et al., 1997).

When the training set consists of fewer than 10 000 instances, the model struggles to generalize, resulting in a notably high RMSE (approximately 0.45) and an R^2 score value near 0. When the number of instances in the training set is between 10 000 and 100 000, new instances significantly improve the model's performance. Once the training set surpasses 100 000 instances, new data still enhances the model's performance, although to a lesser extent.

Assessing the impact of features on model quality. We trained an XGBoost model with different numbers of input features and tracked the quality of model predictions on the test set. Fig. 12 shows the change in RMSE and R^2 score on the test set depending on the number of features. Adding more features improves the model quality but at the cost of increasing the training time (as shown in Fig. 13).[10] Furthermore, a larger increase in the number of features can lead to a slight overfitting of the XGBoost model on the training data and cause a decrease in the quality of model predictions (as observed by the slightly decreased R^2 score and increased RMSE when the model uses more than 200 features).

Therefore, we choose to use all features with a variance greater than 0.5 and to train the model with 78 features. This way, training an XGBoost ML model takes 59 min and 30 s.

Assessing the impact of instance weights on the model training and model quality. We have expected that a highly positively skewed distribution of weights significantly impacts the training time of an ML model and the model itself. To confirm this assumption, we trained the *XGBoost-NoWeights* model using the same configuration as for the training of the *XGBoost* ML model, except that we did not use instance weights in the training process.

The trained *XGBoostNoWeights* model is 6.9MB, almost 24 times larger than the *XGBoost* model. This results from many instances with significantly small weights, which the original *XGBoost* ML model tends to ignore. As the *XGBoostNoWeights* model is much larger than the original model, its inference time is much longer, averaging 1.5 s.

[10] We used variance-based feature selection to select features for training in Figs. 12 and 13. This way, we used the most informative features in each training session.

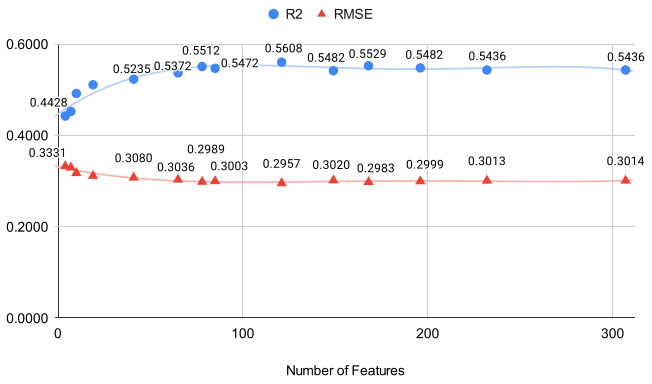


Fig. 12. The impact of the size of the feature set on RMSE and R^2 score metrics in the test set.

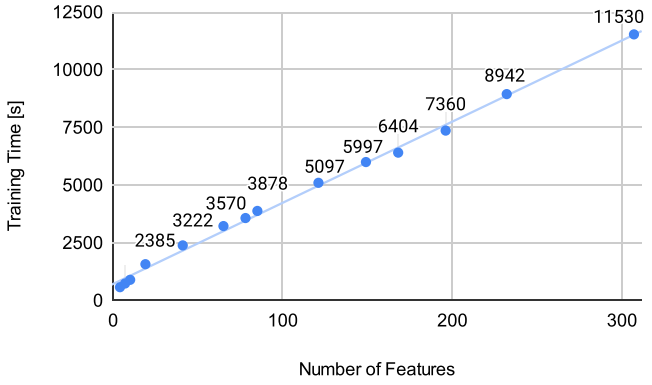


Fig. 13. The impact of the size of the feature set on the duration of model training.

Moreover, the training time for the *XGBoostNoWeights* model is 4503 s. It is 38% longer than the training time of the XGBoost model.

When employing the *XGBoostNoWeights* model in a static profiler, runtime speedup reaches 3.18%, geometric mean. This is 57% lower than the speedup of the static profiler that uses the XGBoost model, which focuses on frequent branches. The *XGBoostNoWeights* model is slightly better than the original model regarding the size of compiled programs. Since the *XGBoostNoWeights* model treats all instances equally, it predicts profiles with equal accuracy for frequent and infrequent code sections, resulting in a 0.89% reduction in the compiled program's size. Conversely, the *XGBoost* ML model with weights tends to ignore instances with lower frequency, leading to their less precise predictions and a slight increase in the compiled program's binary size.

Assessing model prediction quality on programs written in different languages. Our training set consists exclusively of Java applications. To assess the quality of models and features across different languages compiling to Java bytecode, we compared the XGboost model error on the programs from the DaCapo suite (this suite contains programs written only in Java) and programs from the DaCapo con Scala suite (this suite contains programs written in Scala). The RMSE of the XGBoost predictions on programs from the DaCapo suite is 0.2950, while it is 0.3038 for the programs on the DaCapo con Scala suite. On the other hand, the R^2 score of the XGBoost predictions on programs from the DaCapo suite is 0.5402, while it is 0.5664 for the programs on the DaCapo con Scala suite.

Given the close similarity of the measured values for RMSE and R^2 metrics, it is difficult to make general conclusions regarding the quality of XGBoost predictions on Java and Scala programs. This is aligned with the varying runtime performance of Scala programs compared to Java programs (we have examples of both worse and better performance improvement with XGBoost predictions on Scala programs compared to Java programs (Section 5.2.2)). Therefore, we can assume that the quality of a model prediction on a specific program depends on the characteristics of the program, rather than the programming language.

Table 6
Comparison of GraalSP and profilers driven by Wu and Larus's heuristics, XGBoost model and dynamically collected branch profiles considering their impact on runtime, binary size, and compile time.

Profiler	Runtime speedup	Binary size increase	Compile-time increase
Wu and Larus	5.64	31.60	78.51
XGBoost	5.22	2.44	2.17
GraalSP	7.46	3.91	12.01
Dynamicprofiler	33.17	−14.57	−10.98

5.3. Evaluation of GraalSP

According to the presented results, the static profiler that utilizes the XGBoost ML model achieves the best runtime performance speedup, generalizes the best, and injects the lowest compile-time overheads compared to the baseline. Therefore, we choose the XGBoost ML model to predict the branch probabilities in GraalSP. To evaluate incorporating GraalSP into the Graal compiler, we compare:

- as a baseline, the default version of the Graal compiler,
- the Graal compiler that uses branch probability prediction heuristics proposed by Wu and Larus (1994) (we discuss the implementation details in Appendix A),
- the Graal compiler that uses the static profiler driven by the XGBoost ML model,[11]
- the Graal compiler that use GraalSP, and
- the Graal compiler that uses the dynamically collected branch profiles.

Table 6 summarizes this comparison, while Figs. 22, 23, and 24 C show detailed results of the comparison. In the following text, we discuss the obtained results.

5.3.1. Runtime performance

GraalSP achieves an overall runtime improvement of the 7.46% geometric mean compared to the baseline configuration of the Graal compiler (Table 6). This is 2.24% more than the speedup of the static profiler driven by the XGBoost model. This improvement comes from the profile prediction heuristics, representing a tradeoff between a runtime performance and an increase in a binary size and compile time. Fig. 14 presents performance improvements achieved by the profilers on the test suites. Table 12 C presents the geometric standard deviation of the aggregated runtime speedup across test suites. Branch probability prediction heuristics improve the performance of the XGBoost model by 0.47%, 2.70%, and 3.08% on DaCapo, Renaissance, and DaCapo con Scala benchmark suites, respectively.

Fig. 15 summarizes a performance impact of the static profiler that uses the XGBoost ML model and GraalSP across the benchmarks where the XGBoost ML model causes performance slowdowns. GraalSP effectively addresses these outliers. Depending on a program and the influence of mispredicted loops and mispredicted terminating branches on its runtime, GraalSP restores the program's performance to the level of the baseline configuration (as observed in the *scalaxb* benchmark) or improves the benchmark's performance (as observed on the *scala-doku*, *scala-kmeans*, and *scalap* benchmarks).

[11] We use results presented in Section 5.2.2.

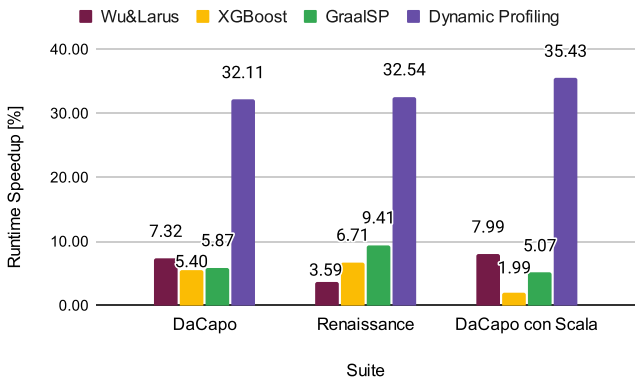


Fig. 14. Runtime speedup of GraalSP and profilers driven by Wu and Larus's heuristics, XGBoost model, and dynamically collected branch profiles.

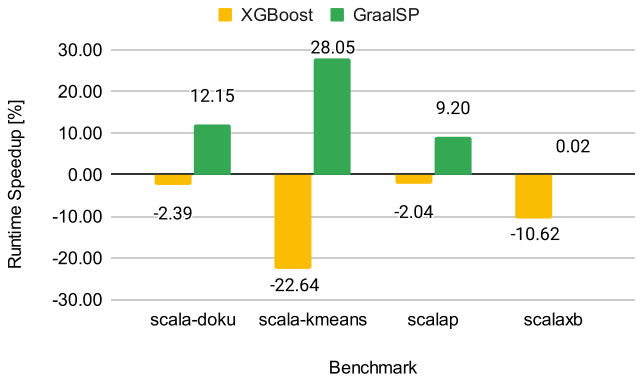


Fig. 15. Runtime speedup of GraalSP and profiler driven by the XGBoost model on the outliers.

The static profiler of Wu and Larus performs 1.82% worse than GraalSP, by improving program execution time by 5.64% (Table 6). Although it has better runtime speedup than GraalSP on the DaCapo and DaCapo con Scala suites (Fig. 14), these results come at a high cost: static profiler driven by Wu and Larus's heuristics leads to the generation of much larger binaries and spends much more time on the compilation (Table 6).

The Graal compiler that uses dynamically collected branches' profiles represents the upper limit regarding performance achievable with a static profiler (an improvement of 33.17%, Table 6). Although this kind of dynamic profiling does not directly include information about executed methods, it implicitly contains such information as the dynamic profiler collects branch profiles only from the executed methods. Consequently, the Graal compiler applies more aggressive optimizations to methods with collected profiles and achieves better results.

5.3.2. Binary size

GraalSP increases the size of the generated binaries by 3.91% (Table 6). An additional increase in the binary size of 1.47% observed between the XGBoost model and GraalSP is the consequence of the probability prediction heuristics. This is an expected behavior: GraalSP's heuristics do not let the loop body probability be less than 0.2, and this way, the Graal compiler performs additional loop body duplications and generates larger programs.

Wu and Larus's heuristics increase the size of the binaries compiled with the Graal compiler by 31.60%. This occurs due to the code duplication phase and the fact that these heuristics aggressively favor

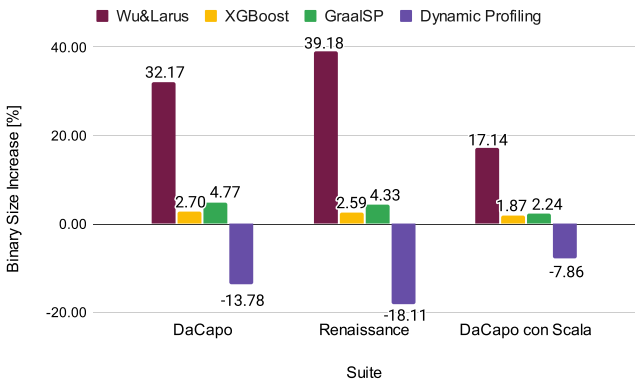


Fig. 16. Binary size increase of GraalSP and profilers driven by Wu and Larus's heuristics, XGBoost ML model, and dynamically collected branches profiles.

loop bodies: three out of nine heuristics by Wu and Larus predict loop body execution with high probabilities (*Loop branch*, *Loop exit*, and *Loop header* heuristics predict loop body execution with probabilities of 0.88, 0.80, and 0.75 respectively). Wu and Larus's heuristics lead to a much larger binary size increase than GraalSP heuristics. This is because Wu and Larus's heuristics assign a much higher execution probability to loop bodies than the GraalSP heuristic.

A binary size of programs compiled with dynamically collected branch profiles decreases by 14.57% compared to the baseline configuration of the Graal compiler. This results from the Graal compiler duplicating only in hot code areas using exact, dynamically collected profiles, which leads to space savings.

Fig. 16 shows the binary size impact of the profilers on the test suites, while Table 12 C shows the geometric standard deviation of the aggregated binary size increase across the test suites. The dynamic profiler outperforms static profilers and, on every test suite, leads to the generation of smaller binaries. In contrast, the static profiler utilizing Wu and Larus's heuristics produces considerably larger binaries than the other profilers (on the Renaissance suite, the increase in binary size is 39.18%).

5.3.3. Compile time

When integrated into the Graal compiler, the static profiler that uses the XGBoost model increases a program's compile time by 2.17%, while GraalSP increases a compile time by 12.01% (Table 6). As already discussed, due to the loop heuristic, GraalSP generates larger binaries than the XGBoost-based static profiler. Generating a larger binary also increases compile time.

Since a static profiler that uses Wu and Larus's heuristics significantly increases a binary size, the compile-time overheads of this profiler are also substantial. The static profiler of Wu and Larus does not perform feature extraction and ML model inference. However, the amount of time spent generating much larger binaries negates this advantage, and the static profiler based on Wu and Larus's heuristics increases compile time much more than GraalSP. The dynamic profiles reduce the overall compile time by 10.98% as this way, the Graal compiler applies time-consuming optimizations only on the hot code segments. When computing the compile time of the dynamic profiler, we do not account for the time needed for program instrumentation and profile collection. If we were to include that as well, the compilation time of the dynamic profiler would be an order of magnitude larger than the compilation time of the static profilers.

Fig. 17 shows the compile-time impact of the profilers on each benchmark suite. Table 12 C reports the geometric standard deviation of the aggregated compile-time increase across benchmark suites. The impact on compile-time on the test suites correlates with the size of generated binary files.

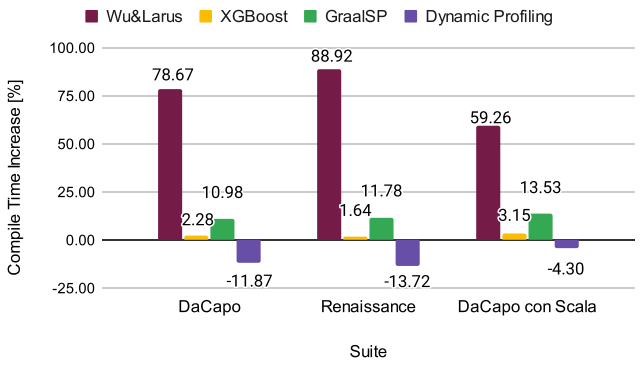


Fig. 17. Compile-time increase of GraalSP and profilers driven by Wu and Larus's heuristics, XGBoost model, and dynamically collected branch profiles.

5.4. Showcases of GraalSP predictions

To get better insights into the predictions of GraalSP, we utilized the GraalSP-PLog to analyze the predictions on particular code samples from test benchmarks. We selected diverse code samples to demonstrate various coding techniques (such as recursion, iteration, and OOP concepts) as well as different programming languages (Java and Scala).

5.4.1. Correct predictions

Listing 1 shows a `for` loop of the `indexDocs` method in the *luindex* benchmark.[12] In this loop, the method indexes all documents from the input directory. The loop on line 4 exemplifies a typical loop structure that invokes a function (alongside performing additional tasks such as object creation, array element access, etc.). GraalSP predicts the execution probability of the `for` loop body to be 0.9873, while during execution, we profiled the execution frequency of the loop body to be 0.9477.

```
1 if (files != null) {
2   System.out.print(file.getName()+" ("+files.length+") ");
3   Arrays.sort(files);
4   for (int i = 0; i < files.length; i++) {
5     indexDocs(writer, new File(file, files[I]), prefix);
6   }
7 }
```

Listing 1: Indexing documents in the *luindex* benchmark.

5.4.2. Incorrect predictions

Listing 2 shows a *for* loop from the `filter` function of the *Jenetics* library that the *feature-genetic* benchmark uses to filter out old and invalid individuals from the population.[13] GraalSP predicts the execution probability of the `for` loop body at 0.7839, while the dynamically profiled execution probability of the loop body is 0.9615. GraalSP predict execution probabilities of the true branches of the first and second `if` statements within the loop at 0.6147 and 0.8284, respectively. The dynamically profiled execution probabilities of the true branches of the `if` statements are 0, as the algorithm did not filter out invalid and old instances from the population (according to our input sample). These two *cfs*-nodes are examples of the GraalSPs incorrect predictions. However, if the input data (or evolution parameters and constraints) were different, the method would execute `if` statements, resulting in smaller prediction errors.

```
1 final MSeq<Phenotype<G, C>> pop = MSeq.of(population);
2 for (int i = 0, n = pop.size(); i < n; ++i) {
3   final Phenotype<G, C> individual = pop.get(i);
4   if (!_constraint.test(individual)) {
5     pop.set(i, _constraint.repair(individual, generation));
6     ++invalidCount;
7   } else if (individual.age(generation) > _evolutionParams
8           .maximalPhenotypeAge {
9     pop.set(i, Phenotype.of(_genotypeFactory.newInstance(),
10          generation));
11    ++killCount;
12  }
13 }
```

Listing 2: Iteration over the population in the *Jenetics* library.

5.4.3. Impact of branch probability prediction heuristics

In addition to facilitating easier analysis of GraalSP predictions on individual code samples, GraalSP-PLog can assist in understanding the impact of branch probability prediction heuristics on GraalSP. In the *scala-kmeans* benchmark, a total of 4030 `If` nodes were executed. Branch probability prediction heuristics refine profiles of 70 nodes: specifically, a loop heuristic is triggered on 28 nodes, while the exit heuristic adjusts probabilities predicted by the ML model on 42 nodes. The activation of the loop heuristic on the sixth and ninth most frequently executed nodes within the benchmark notably enhances the program's runtime performance. In the scala-means benchmark, the loop heuristic was triggered twice for the top 10 most frequent instances: on the 6th and 8th most frequent *cfs*-nodes. On the two incorrectly predicted instances, the XGBoost model predicts execution probabilities of 0.18 and 0.05, where the profiled values are 0.92 and 0.85. Loop heuristic sets these probabilities back to 0.2 and does not let optimizations cut off these loops.

6. Case study

In this section, we examine GraalSP using the example of the Heap Sort algorithm (Listing 3). To sort an array of integers into ascending order, the `heapSort` method uses the `pushDown` method that pushes a specified element down the heap. The static profiler predicts probabilities of executing the body of the `for` loop at line 6 and the body of the `while` loop at line 9.

```
1 /**
2 * Sorts the specified range of the array using
3 * heap sort.
4 */
5 private void heapSort(int[] a, int low, int high){
6   for(int k = (low + high) >> 1; k > low; ){
7     pushDown(a, --k, a[k], low, high);
8   }
9   while(--high > low){
10    int max = a[low];
11    pushDown(a, low, a[high], low, high);
12    a[high] = max;
13  }
14 }
```

Listing 3: Heap Sort Implementation, Java Development Kit 17.

[12] The source code (Java) of this method is available online (GithHub, 2024a).

[13] The source code (Scala) of this method is available online (GithHub, 2024b).

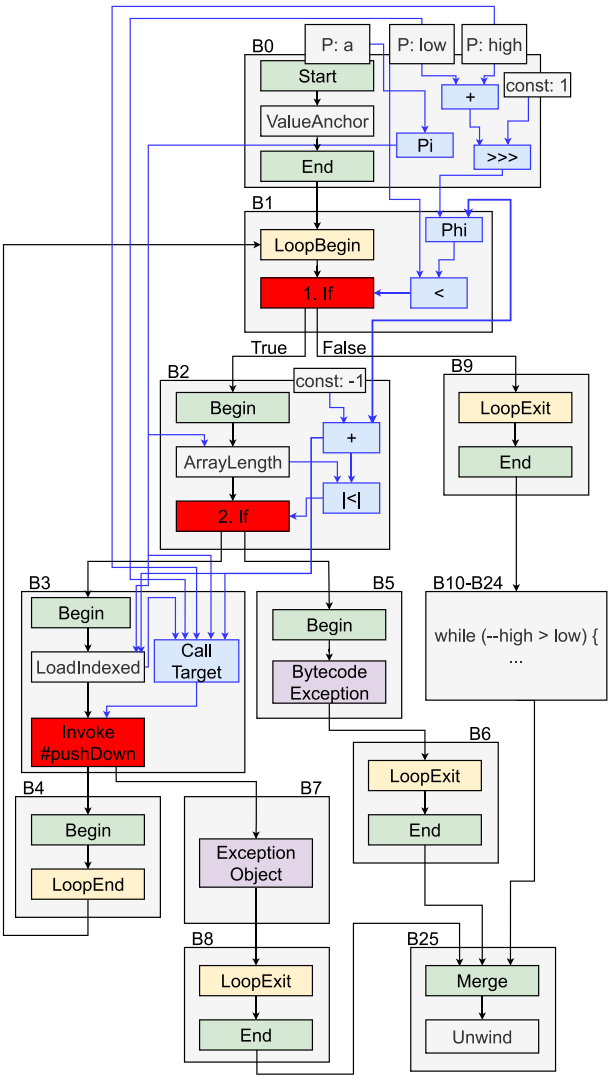


Fig. 18. Graal IR graph and CFG parsed from the `for` loop (the `heapSort` function, Listing 3).

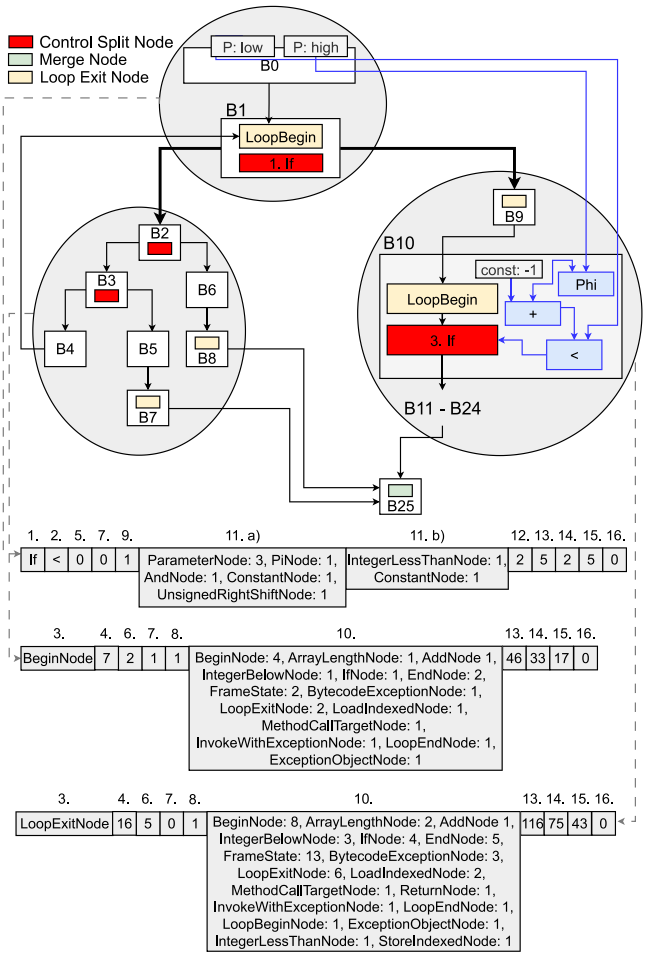


Fig. 19. Static features corresponding to the first branch of the `If.1` node from Fig. 18. Features numbering follows numeration in Table 1.

6.1. Feature extraction

Fig. 18 shows the Graal IR graph and CFG that corresponds to the `for` loop from Listing 3.[14] The *cfs*-node `1.If` from block B1 has two branches: the first one that points to the `Begin` node within block B2 and the second one that points to the `LoopExit` node within block B9. To characterize the first branch of the `1.If` *cfs*-node, we extract features from:

(i) the `1.If` *cfs*-node,
(ii) the block B1 that hosts the `1.If` *cfs*-node,
(iii) the block B0, as a predecessor for the block B1,
(iv) the blocks B2–B8 which are dominated by B2 and from blocks B9–B24, which are dominated by block B9.

Fig. 19 presents all the features extracted for the `1.If` *cfs*-node.

Standard features.. As the `for` loop is on top of the `heapSort` function, its `LoopDepth` and `CSDepth` are 0. As block B0 dominates block B1, the dominator depth of the block B1 is 1. The maximal nesting level in the blocks dominated by block B2 (dominated blocks that corresponds to the CFG edge B1–B2) is 2, as the `Invoke` node from Fig. 18 has the `CSDepth` equal to 2. Also, these dominated blocks have a loop depth equal to 1. The dominated blocks that corresponds to the CFG edge B1–B9 are not within the loop and therefore have a loop depth of 0. The false branch of the `1.If` *cfs*-node starts with the `LoopExit` node while the true branch starts with the `Begin` node. These nodes are values of the `header` feature for the branches of the `1.If` node.

Adapted features.. The true branch of the `1.If` *cfs*-node corresponds to the CFG edge B1–B2. Blocks B2–B8 are dominated by the block B2 and contain a method call (`Invoke` node), two bytecode exceptions (`BytecodeException` and `ExceptionObject` nodes), array element loads (`LoadIndexed` node), array length checks (`ArrayLength` node) and two loop exits (`LoopExit` node). The `NodeCountMap` feature captures this information, as listed in Fig. 19. The `FNodeCountMap` contains only information about floating nodes within the block that host the *cfs*-node (Fig. 19, feature 11a) and within its predecessors (Fig. 19, feature 11b).

Novel features. In the example from Fig. 18, the estimated assembly size of the dominated blocks that belong to the loop body is 46 (calculated by summing the estimated assembly sizes of nodes from blocks B2–B8). The estimated assembly size of the dominated blocks that corresponds to the loop exit (blocks B9–B24) is 116. For example, block B9 contains two nodes: a `LoopExit` node estimated at an assembly size of 4 and an `EndNode` with an estimated assembly size of 0. Blocks

[14] To improve readability, we abstract some details in the IR graph, for example, the numbering of all nodes.

#	Declaring Class	Method Name	Node	Profiled Execution Frequency	Predicted Probability	Guarded Probability	Normalized Execution Count	Execution Count
1	SortExample	pushDown	13|If	0.0603	0.2763		0.2281841392	120636368
2	SortExample	pushDown	22|If	0	0.2084		0.2144282767	113363920
3	SortExample	pushDown	75|If	0.9759	0.8048		0.2144282767	113363920
4	SortExample	pushDown	50|If	0.5081	0.4885		0.214428254	113363908
5	SortExample	generateRanc	24|If	1	0.8363		0.01891503912	10000001
6	SortExample	heapSort	48|If	1	0.7181		0.01891503723	10000000
7	java.util.Random	next	33|If	0	0.563		0.01891503723	10000000
8	SortExample	heapSort	14|If	1	0.642		0.009457520506	5000001
9	java.lang.Integer	stringSize	29|If	0.3289	0.3243		0.002437033257	1288410.5
10	java.lang.Integer	stringSize	22|If	1	1		0.002436988807	1288387
11	java.nio.Buffer	position	29|If	0	0.0002		0.002314271828	1223509
12	java.nio.Buffer	position	5|If	1	1		0.002314042957	1223388
13	java.nio.ByteBuffer	arrayOffset	11|If	1	1		0.002253632111	1191450
14	java.nio.ByteBuffer	arrayOffset	6|If	0	0.0002		0.002253492139	1191376
15	java.nio.CharBuffer	arrayOffset	9|If	1	1		0.002240123937	1184308

Fig. 20. GraalSP-PLog report: nodes from the `heapSort` method are marked in orange, while nodes from the `pushDown` method are marked in yellow. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

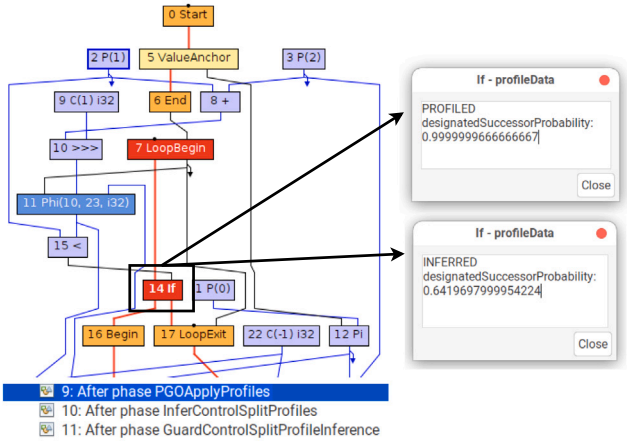


Fig. 21. Graal IR graph in the GraalSP-PLog: Graal IR of the `heapSort` method (upper left), part of a list of the profiling phases that can be inspected (lower left), pop-up windows displaying dynamically profiled execution frequency of the branch `14 If − 16 Begin` (upper right) and its execution probability predicted by GraalSP (lower right).

dominated by the block B2 have 17 cheap CPU instructions, while those dominated by the block B9 contain 43 such instructions.

6.2. GraalSP inference

We concatenate three vectors from Fig. 19, encode categorical and map features, and apply feature selection to create an input feature vector for the XGBoost model. We run the XGBoost model and predict the probability of executing the true branch of the `1.If` node or, specifically, the probability of executing the body of the `for` loop to be 0.64.

Since predicted probability is not below 0.20, we do not apply the loop heuristic. As the branches of the `1.If` node does not lead to program termination, we do not apply the dead-end heuristic to tune predicted probability. Using a similar procedure, GraalSP predicts the execution of the loop body of the *while* loop as 0.72.

6.3. Evaluation of the GraalSP predictions

To evaluate the quality of GraalSP predictions, we can compare them with execution frequencies collected via dynamic profiling or we can opt for human evaluation. However, the most important evaluation of the quality of GraalSP predictions is the evaluation of their impact on the runtime efficiency and binary size of the compiled programs and their impact on the compilation time.

6.3.1. Dynamic profiling

Fig. 20 shows the top 15 instances of the GraalSP-PLog prediction report with the results of dynamic profiling of execution of the `heapSort` method with an input array of 10 million elements. Note that, with different-sized input arrays, profiling would collect different execution frequencies.

The normalized execution counts of nodes from the `heapSort` and `pushDown` methods sum up to 0.8998, meaning that the execution performance of the sorting program will highly depend on the execution performance of these two functions. GraalSP predicts the profiles of these nodes with the RMSE value of 0.2336.

With GraalSP-PLog, we can track the statically predicted branch execution probabilities on the IR graph of a method, as shown in Fig. 21.[15] In addition to the statically predicted branch execution probabilities, we can also track the dynamically profiled execution frequencies.

[15] IR node `14 If` in Fig. 21 corresponds to node `1.If` in Fig. 18.

6.3.2. Human analysis

Human evaluation of the predicted probabilities relies on the semantics and the anticipated distribution of input data. The `for` loop starting at line 6 inserts half of the elements from the input array `a` into the heap. Thus, the body of this loop will be executed $\frac{n}{2}$ times, where n is the size of the array `a`. Additionally, the loop exits only once, when all the elements are inserted into the heap. This leads to an execution frequency of $\frac{\frac{n}{2}}{\frac{n}{2}+1}$. The minimal execution frequency (for an array of only two elements) is 0.5 and the execution frequency increases with the size of the array. Therefore, each prediction larger than 0.5 is sound, while higher values correspond to the sorting of longer arrays.

6.3.3. Performance impact

Accurate predictions of the execution probabilities are crucial for aggressive optimizations, such as function inlining. If a static profiler predicts that the bodies of the `for` and `while` loops will execute frequently, that will suggest the compiler to inline the invocations of the `pushDown` method. Otherwise, the compiler will not inline the `pushDown` method, and the call costs will be incurred during program execution. For example, consider sorting a randomly generated array of 10 million integers. If calls to the `pushDown` method are inlined, the average sorting time is 1.80 s, while if these calls are not inlined, the average sorting time is 2.18 s.[16] Therefore, imprecise branch probabilities slow down sorting by 21.11%.

GraalSP predicts 0.64 as the probability of executing the body of the `for` loop from Listing 3. We can question the quality of this prediction, as higher predictions correspond to execution frequencies of larger arrays, where sorting efficiency becomes more crucial. For example, for all arrays longer than 20 elements, the execution frequency is greater than 0.90, corresponding to a prediction error of at least 0.26. However, concerning the observed performance of the sorting program optimized according to the GraalSP predictions, a prediction of 0.64 is sufficient to trigger function inlining and improve the runtime performance of the sorting. When compiling a sorting program with GraalSP enabled, we observed a 3.12% increase in the binary size of the generated

[16] To assess the real impact on users, we ran the experiment using the enterprise version of the Graal compiler on a development laptop with a 6-core Intel i7 processor. We measured sorting time using the nanosecond precision and averaged ten independent program runs.

executable file (from 6.4 MB to 6.6 MB). Additionally, the compilation time increased by 16.44% (from 22.5 s to 26.2 s).

7. Related work

This section examines the relevant literature regarding static branch prediction and static profilers. Static branch prediction is a problem of predicting which branches will be taken at the program runtime. Contrary to static branch prediction, static profilers predict the branch probabilities and do not classify the branches as taken or not taken.

7.1. Static branch prediction

Employing the hand-crafted heuristics to solve the static branch prediction problem based on a fixed set of features is motivated by the observation that control flow performs similarly over different program executions (Fisher and Freudenberger, 1992). Ball and Larus (1993) established a set of static heuristics for predicting the non-loop branches based on a set of handcrafted, binary-level features. They ordered the heuristics empirically. In cases when multiple heuristics apply to the same branch, they use the first applicable heuristic to predict the taken branch. These heuristics are used by default in the Clang compiler (Lattner, 2008; LLVM Project, 2024; Moreira et al., 2021). Deitrich et al. (1998) expand the Ball and Larus heuristics and integrate them into the IMPACT compiler (Chang et al., 1991). Also, static heuristics can perform branch prediction using the features extracted from the program's source code (Wong, 1999). DT model can be used to optimize the order of application of the hand-crafted heuristics and discover the new branch prediction heuristics (Desmet et al., 2005).

Calder et al. (1997) established using the ML classification models in the static branch prediction. They employ DT and DNN models to predict taken branches. They train the ML models using the features extracted from a low-level IR of a C compiler for MIPS and of a Fortran compiler for DEC.

Buse and Weimer (2009) use the logistic regression (LR) model (Wright, 1995) that, based on the source-level features, successfully predicts hot paths.[17] LR successfully selects the top 5% of paths that account for more than half of a program's total runtime. Zekany et al. (2016) introduce recurrent neural networks (Medsker and Jain, 1999; Greff et al., 2016) that operate on the compiler's IR as hot-path classifiers. This way, they avoid feature extraction from the source code and achieve language independence.

Shih et al. (2021) apply both classical ML methods and deep learning to solve the branch prediction problem. They extracted features from the LLVM IR (Lattner, 2008; LLVM Project, 2024), and trained five classical ML models: DT, the k-nearest neighbors (KNN) (Keller et al., 1985), random forest (Biau and Scornet, 2016), support vector machine (SVM) (Steinwart and Christmann, 2008), and gradient boosting ensemble (Bentéjac et al., 2021). They also utilize a graph neural network (GNN) (Zhou et al., 2020; Wu et al., 2020) model that works on top of the ProGraML features.[18] They show that the GNN model, which does not predefine the code characterizing features, can achieve performance comparable to the classic ML models (Shih et al., 2021).

Besides branch and hot path predictions, ML models can predict other programs' runtime characteristics. The KNN classifier successfully deals with a multi-label classification and predicts loop unroll factor (Stephenson and Amarasinghe, 2004). The SVM classifier trained on ten static code features, such as the number of loops, calls, and load instructions within a method, can predict hot methods with an accuracy of over 60% (Johnson and Valli, 2008; Mahapatra and Patra, 2018). Also, the ML models can predict the relative frequency of the virtual calls based on the static code features (Zhang et al., 2011). Message-passing NN (Li et al., 2016; Gilmer et al., 2017) can successfully utilize ProGraML representation and, with high accuracy, predict advanced tasks such as heterogeneous device mapping and program classification (Cummins et al., 2021).

7.2. Static profilers

State-of-the-art static profilers take advantage of handcrafted heuristics (Wu and Larus, 1994), classification ML models (Kotsiantis et al., 2007; Osisanwo et al., 2017), and regression ML models (Caruana and Niculescu-Mizil, 2006; Alpaydin, 2020) that utilize binary-level features.

Wu and Larus (1994) established a set of static heuristics that predict branch probabilities. Table 9 A lists these heuristics. They approximate the likelihood of a branch being taken by combining the predictions of the nine heuristics that operate on a program binary. Implemented heuristics rely on heuristics initially defined by Ball and Larus (1993). Wu and Larus improve some of the heuristics proposed by Ball and Larus, introduce new ones, and propose using Dempster-Schaefer's theory of evidence (Gordon and Shortliffe, 1984) to overcome the problem of combining multiple manually defined heuristics. Each heuristic h, if applicable, predicts the taken branch b with the predefined probability $h(b)$. When two heuristics (h_1 and h_2) apply to a branch b, they are combined using the formula: $(h_1 \oplus h_2)(b) = \frac{p \cdot q}{p \cdot q + (1-p) \cdot (1-q)}$, where $h_1(b) = p$ and $h_2(b) = q$.

The GCC compiler uses heuristics similar to those proposed by Wu and Larus, with additional heuristics tailored for the C programming language (e.g., predict that the return value of `malloc` function is almost always non-null) (Stallman et al., 2012; Alovisi, 2020). GCC compiler can combine heuristics using Dempster-Schaefer's theory, following the approach suggested by Wu and Larus. The Intel Labs Haskell Research Compiler (Liu et al., 2013) employs a static profile estimation approach based on Wu and Larus's work to perform selective inlining at high-frequency call sites. Boogerd and Moonen (2006) applied the five heuristics of Wu and Larus, along with their methodology of combining multiple heuristics, to develop the *ELAN* tool, a tool that helps users prioritize information gathered from software inspection tools (Fagan, 2011). Sun et al. (2011) employ a static profiler proposed by Wu and Larus to develop the *Lukewarm* tool. This tool provides probabilistic points-to-analysis (Hwang et al., 2003) of Java programs.

Moreira et al. (2021) proposed VESPA, a regression DNN model that predicts branch probabilities based on the binary level features. They improve the work of Calder et al. (1997) by expanding the feature set used to describe the branch instructions in the code and adopting their models to predict branch probabilities. They demonstrate that the ML models can drive binary optimizations by integrating the VESPA in the BOLT binary optimizer (Panchenko et al., 2019). This way, they improved the runtime performance of the binaries optimized with BOLT by an average of 5.47%.

Rotem and Cummins (2021) developed a static profiler that employs an XGBoost ensemble for multiclass classification (Aly, 2005). To train the model, they discretize profiled probabilities into 11 categories (from 0.0 to 1.0, with a uniform step size of 0.1), thus translating the branch probability prediction problem to the multi-class classification problem. The trained ensemble and corresponding static profiler were integrated directly into LLVM, supplementing the existing heuristics.

[17] The hot path refers to the sequence of frequently executed instructions.

[18] *Program graphs for ML* (ProGraML) (Cummins et al., 2021) proposes a novel, compiler-independent graph-based program representation that captures the control flow and the data relations defined by the source code.

Table 7
Quantitative analysis of the state-of-the-art static profilers. Values in the table (except the first two rows) are percentages compared to a baseline configuration. Empty cells correspond to non-reported values.

	Wu and Larus (1994)		VESPA (Moreira et al., 2021)	Rotem and Cummins (2021)	GraalSP
	BOLT (Panchenko et al., 2019; Moreira et al., 2021)	Graal compiler (Wimmer et al., 2019)			
Number of test benchmarks	4	28	4	10	28
Aggregation metric	Average	Geomean	Average	Geomean	Geomean
Runtime speedup (%) Min, Max	2.77 1.11, 4.42	5.64 −49.30 , 20.11	5.47 2.28, 8.84	1.60 −7.00 , 16.00	7.46 −0.88 , 28.05
Binary size increase (%) Min, Max		31.60 12.52, 47.93			3.91 1.46, 6.44
Increase of the time to generate a binary (%) Min, Max	178.72 138.56, 213.24	78.51 31.70, 122.20	1296.07 1089.13, 1629.82		12.01 3.37, 21.01

8. Discussion

We discuss GraalSP in the context of related work, i.e. in the context of the state-of-the-art static profilers: the static profiler based on Wu and Larus's heuristics (Wu and Larus, 1994) and the ML-based static profilers, one presented by Rotem and Cummins (2021) and VESPA (Moreira et al., 2021). We also discuss threats to validity.

8.1. Quantitative analysis

Static profilers target different platforms and programming languages and are evaluated on different benchmarks. Therefore, reported numbers cannot be directly compared, and their comparison must consider all the stated differences.

Table 7 summarizes the impact of the discussed static profilers on the program's runtime, binary size, and time required to generate a binary according to values reported in the corresponding papers and values measured in our evaluation. Values in the table report percentages relative to a baseline configuration of a compiler.[19]

The impact on runtime performance significantly depends on the benchmarks used in the evaluation. We can observe a wider range between minimal and maximal impact when a bigger number of different benchmarks is considered. The relevant studies do not report results regarding the impact of static profilers on the size of the optimized programs. We consider this to be a very important metric. Concerning the time to generate a binary, GraalSP obtains an order of magnitude better results than the other static profilers.[20]

8.2. Qualitative analysis

We conduct a comparison of the static profilers concerning their feature sets, employed ML models, employed branch probability prediction heuristics, and software platforms they target. Table 8 summarizes this comparison.

Feature set. To the best of our knowledge, there is no other static profiler that operates on a graph-based IR. Wu and Larus (1994) introduce branch probability prediction heuristics based on a program binary. Similarly, VESPA (Moreira et al., 2021) leverages low-level features extracted from a binary code. Rotem and Cummins (2021) extract features from the LLVM IR (Lattner, 2008; LLVM Project, 2024). Data flow within a program is fundamental for all optimizations (Brauckmann et al., 2020). While data-flow information can be obtained in other IRs, in Graal IR floating nodes that represent the data-flow are available by default, and their access does not require additional effort. Feature importance analysis from Fig. 8 confirms the importance of data-flow information and shows that 3 out of the top 5 most important features are floating nodes from the `FNodeCount` map. The Graal IR is a high-level intermediate representation, created immediately after the bytecode parsing, which enables more phases to utilize the predicted profiles. The challenge in defining a static profiler on top of Graal IR is the absence of low-level features such as assembly size and CPU cycles. We addressed this issue by defining high-level static estimates of the low-level features.

ML model. Calder et al. (1997) trained a DT model to solve the branch prediction task. To avoid overfitting, they post-prune the DT using minimal complexity pruning. We trained a DT model and applied the same pruning approach. When training the XGBoost ensemble, we limit the depth of each DT within the ensemble, as limiting the depth is more efficient than post-pruning. This is important as we have 1500 DTs in the XGBoost ensemble. By limiting the depth of each DT, we achieve the same results as when doing the post-pruning. Moreira et al. (2021) claim that their attempts to train the ensemble model end up consuming an excessively large amount of memory. Rotem and Cummins (2021) use the XGBoost ensemble for multi-class classification, while we use it for regression. This way, we predict profiles more precisely and avoid losing the information during rounding and classifying probabilities into discrete categories. On the other hand, VESPA employs the DNN regression model to predict the branches' probabilities. We chose the XGBoost ML model to drive GraalSP because it delivers the best runtime performance and imposes smaller compile-time overheads while its size, inference time, and training time are significantly smaller than the size, inference time, and training time of the DNN model. Although there is a potential for further improvements in the DNN model, the size of the DNN model, the inference time, and the training time are not likely to decrease significantly.

Instances' weights. Previous research applies various approaches to handle varying execution frequencies. To focus on more frequent branches, Calder et al. (1997) duplicate branches proportionally to

[19] A baseline configuration of a compiler is a default compiler configuration that does not use a static profiler.

[20] Moreira et al. (2021) provide only times required for generating binaries. We divided values for VESPA with the reported times of the default setup, averaged the obtained values, and converted the average to a percentage.

Table 8
Main characteristics of the state-of-the-art static profilers.

	Wu and Larus (1994)	VESPA (Moreira et al., 2021)	Rotem and Cummins (2021)	GraalSP
Features	Binary	Binary	LLVM IR	Graal IR
ML Based	No	Yes	Yes	Yes
ML Model		DNN	XGBoost	XGBoost
ML Technique		Regression	Multiclass classification	Regression
Instances' Weights		Normalized freq.	Ignore low freq.	Normalized freq.
Heuristics (count)	Yes (9)	No	No	Yes (2)
Target Platform	DYNIX/ptx C compiler	BOLT (Panchenko et al., 2019)	LLVM (Lattner, 2020)	GraalVM (Wimmer et al., 2019)

their normalized weights, while Desmet et al. (2005) add an extra feature to capture the execution frequencies of branches. Rotem and Cummins (2021) ignore branches with a small number of samples. Like VESPA (Moreira et al., 2021), we followed the approach proposed by Calder et al. and used the normalized branch weights for training.

Heuristics. Wu and Larus (1994) employ nine static heuristics that predict the execution probabilities of the branch instructions. On the other hand, we employ two heuristics to handle possible outliers. Other static profilers do not use heuristics.

Target platform. We integrated GraalSP in the Graal compiler, 23.0, Enterprise Edition, released in June 2023. Wu and Larus (1994) use the Sequent DYNIX/ptx C compiler, version 2.1. Most of the heuristics authors adopt from the work of Ball and Larus (1993), which are integrated into the Clang compiler and enabled by default (Moreira et al., 2021; Lattner, 2008). Moreira et al. (2021) use the binary optimizer BOLT to develop VESPA, while Rotem and Cummins (2021) develop the static profiler using the LLVM infrastructure.

8.3. Threats to validity

Various validity threats (e.g., internal and external) persist in experiment-based research (Wohlin et al., 2012). This section discusses the threats faced when designing and evaluating a static profiler.

Internal threats.. The experimental setup provides internal validity. The experiments were conducted in isolation, ensuring no other applications were running on the server during the benchmarking. This eliminated all potential external factors that might have influenced the results. To eliminate the measurement instability, we run the benchmarks multiple times. Our experiments are fully reproducible, as we have made all necessary resources publicly available, including the applications used to create the model training set (Oracle Corporation, 2023) and a detailed description of the training pipeline. We employed an identical version of the Graal compiler when benchmarking different profilers, ensuring an identical compiler's core in all experiments. Consequently, any disparities in the results correspond only to the variations introduced by different profiles.

External threats.. The training and test sets do not overlap while both contain real-world Java programs (Oracle Corporation, 2023; Prokopec et al., 2019b; Blackburn et al., 2006; Sewe et al., 2011). Therefore, we anticipate that the reported numbers on the test set are relevant. We conducted compile-time measurements on a development laptop to assess the impact of the ML model inference on the program compile time. This way, we report times relevant to all users who will compile programs using the Graal compiler on their machines.

9. Conclusions and future work

In this paper, we proposed GraalSP, a polyglot, efficient, and robust static profiler. We proposed a novel static feature set that characterizes branches based on the high-level, graph-based Graal IR. Besides the standard features that capture the program's control flow, we introduced static cost model features that estimate low-level code characteristics. To semi-automate the process of defining the feature set, we introduced map-based features that capture relevant information from the Graal IR, including numerous handcrafted features from prior research.

We created an extensive training dataset by instrumenting 44 real-world Java applications. A detailed analysis of the dataset revealed that instance weights are highly important. As the distribution of the weights is highly positively skewed, utilizing these weights leads to the creation of a smaller model with reduced inference time that focuses on frequent branches, thereby resulting in improved runtime performance.

We chose the XGBoost regression ML model to predict the probabilities of branches based on the proposed static feature set. We experimented with several ML models and chose the XGBoost model to achieve efficiency (it predicts in less than 0.5 ms) and to get the most accurate predictions (speedup of 5.22% on the test programs). In addition, the trained XGBoost ML model has only 290KB.

We integrated GraalSP into the Graal compiler, thus developing an end-to-end solution and proposing a robust and language-agnostic static profiler that operates on any language compiling to Java bytecode. This way, we improve the runtime performance of applications compiled with the Graal compiler. We achieved a geometric mean runtime speedup of 7.46% at the cost of increasing the binary size and compile time by only 3.91% and 12.01%, respectively. For future work, we plan to predict frequencies of virtual call targets using features extracted from Graal IR. This way, we plan to expand GraalSP and provide a lightweight static profiler that supports a context-sensitive PGO. As the Graal compiler relies on the graph-based IR, we plan to remove the manual feature engineering from the design of a static profiler, by utilizing GNN models. Also, by utilizing GNN as a more powerful and robust ML model compared to a tree ensemble, we plan to explore the possibility of removing the profile prediction heuristics from a static profiler.

CRediT authorship contribution statement

Milan Čugurović: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Milena Vujošević Janičić:** Writing – review & editing, Writing – original draft, Validation, Supervision, Investigation, Conceptualization. **Vojin Jovanović:** Writing – review & editing, Validation, Supervision, Investigation, Conceptualization. **Thomas Würthinger:** Writing – review, Supervision, Resources, Project administration, Investigation, Funding acquisition.

Table 9
Profile prediction heuristics of Wu and Larus (1994). The second column refers to the execution probability assigned to a branch (if the heuristic applies).

Heuristic	Probability	Description
Loop branch	0.88	Predict as taken a branch leading back to a loop's head and as not taken a branch that exits the loop.
Pointer	0.60	Predict that a pointer comparison with null or comparison between two pointers will result in failure.
Opcode	0.84	Predict that a comparison of an integer for less than zero, less than or equal zero, or equal to a constant will fail.
Guard	0.62	If a comparison involves a register being used as an operand, and that register is used before being defined in a successor block (SB) that does not post-dominate the block that contains the comparison, predict that a branch that connects the block that hosts the comparison with SB block is taken.
Loop exit	0.80	If a loop contains a comparison in which no successor is a loop head, predict that the comparison will not exit the loop.
Loop header	0.75	Predict that a branch to a successor that is a loop-header or a loop pre-header and does not post-dominate is taken.
Call	0.78	Predict that a branch to a successor that contains a call and does not post-dominate is not taken.
Store	0.55	Predict that a branch to a successor that contains a store instruction and does not post-dominate is not taken.
Return	0.72	Predict that a branch to a successor that contains a return instruction is not taken.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The manuscript includes links to the used data.

Acknowledgments

We thank Predrag Janičić (University of Belgrade), Mladen Nikolić (University of Belgrade), and Rodrigo Bruno (University of Lisbon) for thoughtful reviews and insightful discussions. Our gratitude also extends to the anonymous reviewers for their valuable feedback. This work was partially supported by the grant 451-03-47/2023-01/200104 from the Ministry of Science, Technological Development and Innovation of the Republic of Serbia and by the research grant provided by Oracle America, Inc. to the Faculty of Mathematics, University of Belgrade.

Appendix A. Wu and Larus's heuristics

Table 9 lists heuristics proposed by Wu and Larus (1994). Below, we provide implementation details for each of the heuristics.

Loop branch heuristic The Graal compiler parses loops to the binary *cfs*-nodes in which one of the branches corresponds to the loop body and the other corresponds to the loop exit. In such *cfs*-nodes, we assign the execution probability of 0.88 to the branch that corresponds to the loop body.

Pointer heuristics The execution takes one of the branches of a *cfs*-node depending on the output from a logic node that checks for the condition that indicates the branching in the program. We identify *cfs*-nodes that correspond to the pointer comparisons (by the `PointerEquals` logic node) and set the execution probability to 0.6 for the branches that will be taken if the comparison of a pointer with null or comparison between two pointers fails.

Opcode heuristic The Graal compiler parses the integer comparisons to the `IntegerLowerThanNode` and `IntegerEqualsNode`. We use these nodes to set to 0.84 the probabilities of the branches that correspond to the failed comparison of integers for less than zero, less than or equal to zero, or equal to a constant. The Graal compiler represents constants with the `ConstantNode`, and we use this node type to identify branches that correspond to the comparison with a constant.

Guard heuristic We use the floating nodes which capture data flow in the Graal IR in the same way as registers in the binary. We predict as taken, with the probability 0.62, a branch whose corresponding `NodeCountMap` contains one of the floating nodes: `UnaryNode`, `BinaryNode`, `LogicNegationNode`, `UnaryOpLogicNode`, and `BinaryOpLogicNode`.

Loop exit heuristic We identify a comparison in a loop as the *cfs*-node within the loop that does not have a branch that points back to the loop header. In that case, we assign an execution probability of 0.8 to the branch that does not point to the `LoopExitNode`, as that branch corresponds to the loop body and does not exit the loop.

Loop header heuristic We rely on the fact that the loops in the Graal IR start with a `LoopBeginNode` (for example, block B1 from Fig. 18). This allows us to detect branches that are loop headers or loop preheaders and to assign them an execution probability of 0.75.

Call heuristic We check the number of `InvokeNodes` from the `NodeCountMap` corresponding to the CFG blocks dominated by the block that the branch points to. If the number of `InvokeNodes` is larger than 0, we set the execution probability to 0.78 for that branch.

Store heuristic We check the number of `StoreFieldNodes`, `RawStoreNodes`, and `StoreIndexedNodes` from the `NodeCountMap` corresponding to the CFG blocks dominated by the block that the branch points to. For values greater than 0, we assign an execution probability of 0.45 to the corresponding branch.

Table 10
Libraries and benchmarks from the GraalVM Reachability Metadata Repository (Oracle Corporation, 2023) used for creating a training set.

No.	Library	Benchmark	Version
1.	org.opengauss	opengauss-jdb	3.1.0
2.	io.netty	netty-common	4.1.80
3.	io.grpc	grpc-netty	1.51.0
4.	com.sun.mail	jakarta.mail	2.0.1
5.	io.opentelemetry	opentelemetry-exporter-jaeger	1.19.0
6.	com.mysql	mysql-connector-j	8.0.31
7.	com.hazelcast	hazelcast	5.2.1
8.	com.github.ben-manes.caffeine	caffeine	3.1.2
9.	org.jboss.logging	jboss-logging	3.5.0
10.	org.apache.tomcat.embed	tomcat-embed-core	10.0.20
11.	org.apache.httpcomponents	httpclient	4.5.14
12.	org.liquibase	liquibase-core	4.17.0
13.	org.thymeleaf	thymeleaf	3.1.0.RC1
14.	org.glassfish.jaxb	jaxb-runtime	3.0.2
15.	io.jsonwebtoken	jjwt-gson	0.11.5
16.	com.google.protobuf	protobuf-java-util	3.21.12
17.	com.graphql-java	graphql-java	19.2
18.	com.github.luben	zstd-jni	1.5.2
19.	org.apache.activemq	activemq-broker	5.18.1
20.	javax.cache	cache-api	1.1.1
21.	org.eclipse.jetty	jetty-client	11.0.12
22.	org.hibernate.orm	hibernate-core	6.2.0
23.	io.undertow	undertow-core	2.2.19
24.	org.ehcache	ehcache-jakarta	3.10.8
25.	com.ecwid.consul	consul-api	1.4.5
26.	org.apache.commons	commons-pool2	2.11.1
27.	org.quartz-scheduler	quartz	2.3.2
28.	commons-logging	commons-logging	1.2
29.	org.hdrhistogram	HdrHistogram	2.1.12
30.	com.zaxxer	HikariCP	5.0.1
31.	ch.qos.logback	logback-classic	1.4.1
32.	jakarta.servlet	jakarta.servlet-api	5.0.0
33.	org.freemarker	freemarker	2.3.31
34.	org.postgresql	postgresql	42.3.4
35.	org.mockito	mockito-core	4.8.1
36.	org.testcontainers	testcontainers	1.17.6
37.	org.flywaydb	flyway-core	9.0.1
38.	org.example	library	0.0.1
39.	com.microsoft.sqlserver	mssql-jdbc	12.2.0.jre11
40.	org.jline	jline	3.21.0
41.	org.jetbrains.kotlin	kotlin-stdlib	1.7.10
42.	com.h2database	h2	2.1.210
43.	org.jooq	jooq	3.18.2
44.	net.java.dev.jna	jna	5.8.0

Return heuristic We check the number of `ReturnNodes` and `FarReturnNodes` from the `NodeCountMap` corresponding to the CFG blocks dominated by the block that the branch points to. For values greater than 0, we assign an execution probability of 0.28 to the corresponding branch.

Note that the *Loop exit* heuristic and the *Loop branch* heuristic have different targets. The *Loop exit* heuristic targets the *cfs*-nodes within the loop body, whereas the *Loop branch* heuristic targets *cfs*-nodes that correspond to the loops from the program's sources (e.g., the `1. If` *cfs*-node from Fig. 18 corresponds to the `for` loop from the `heapSort` function).

When heuristic checks the dominator relations between blocks, we use the CFG corresponding to the Graal IR. For example, we apply the *Guard* heuristic to a branch connecting blocks `CfsNodeHostBlock` (a block that hosts the *cfs*-node) and `BranchPointBlock` (a block that the CFG edge corresponding to the branch points to) only when block `BranchPointBlock` does not post-dominate block `CfsNodeHostBlock`.

As suggested by Wu and Larus, we use Dempster-Shafer's theory of evidence (Gordon and Shortliffe, 1984) to combine predictions from individual heuristics. We implemented the static profiler based on these heuristics as a separate phase within the Graal compiler.

Appendix B. Training and test set libraries

Table 10 lists programs that are used for creating a training set.

Table 11 lists programs that are used for creating a test set and that are also used as test benchmarks.

Appendix C. Detailed results

See Table 12 and Figs. 22–24.

Table 11
Programs used for creating a test set and also used as test benchmarks. The column marked with n refers to the number of runs we averaged to calculate the execution time of the benchmark. The last column (*Execution Time (ms)*) reports the average execution time of a benchmark (measured in milliseconds) and the standard deviation of execution time across different runs.

DaCapo suite, version 9.12

No.	Benchmark	Description	n	Execution Time (ms)
1.	avrora	Simulates programs run on a grid of AVR microcontrollers (Kunikowski et al., 2015).	20	6741.39 ± 18.24
2.	fop	Parses and converts XSL-FO files to PDF.	40	1206.28 ± 14.90
3.	h2	Runs a relational database management system H2 database engine (HSQLDB Group, 2023) via executing transactions in a banking application model.	40	27112.14 ± 130.72
4.	luindex	Uses Apache Lucene (Białecki et al., 2012; Lucene, 2010) to index the works of Shakespeare and the King James Bible.	20	5679.21 ± 68.53
5.	lusearch	Uses Apache Lucene to do a text search of keywords over a corpus of data comprising the works of Shakespeare and the King James Bible.	40	438.01 ± 6.11
6.	pmd	Uses the programming mistake detector (PMD Open Source Project, 2023), a cross-language static code analyzer that analyzes program source code and detects common programming flaws (unused variables, empty catch blocks, etc.).	30	15947.94 ± 180.13
7.	sunflow	Uses the Sunflow rendering engine (Kulla, 2023) to test how JVM handles computationally intensive tasks like ray tracing (Glassner, 1989).	30	1938.48 ± 29.38

Renaissance suite, version 1.14.0

No.	Benchmark	Description	n	Execution Time (ms)
8.	akka-uct	Executes the actor workload of the Unbalanced Cobwebbed Tree (Zhao and Jamali, 2013) in the Akka framework (Ortiz and Mishra, 2017; Srirama et al., 2021).	24	51116.57 ± 337.46
9.	finagle-chirper	Simulates a microblogging service using Twitter Finagle (Twiter Inc, 2023), a remote procedure call framework for building network applications.	90	8483.10 ± 83.10
10.	finagle-http	Emulates a high server load using the Twitter Finagle.	12	9223.50 ± 51.95
11.	fj-kmeans	Utilizes the Fork/Join framework to execute the k-means algorithm.	30	5796.09 ± 100.18
12.	future-genetic	Does the genetic algorithm function optimization using Jenetics (Wilhelmstötter, 2021), a Java genetic algorithm library.	50	14132.75 ± 78.24
13.	mnemonics	Solves the phone number mnemonics problem using JDK streams.	16	41063.38 ± 243.61
14.	par-mnemonics	Solves the phone number mnemonics problem using parallel streams.	16	33323.08 ± 179.94
15.	philosophers	Simulates the dining philosophers algorithm (Lehmann and Rabin, 1981) using the ScalaSTM framework (Bronson, 2023), a lightweight software transactional memory for Scala.	30	4640.16 ± 31.59
16.	reactors	Runs a benchmark similar to the Savina microbenchmark workloads (Imam and Sarkar, 2014) in a sequence on Reactors.IO (Scala Community, 2023), framework for distributed programming.	10	66946.44 ± 2485.89
17.	rx-scrabble	Solves the Scrabble puzzle using the functional reactive concepts (Nilsson et al., 2002) and the RxJava framework (ReactiveX, 2023).	80	1806.08 ± 5.86
18.	scala-doku	Solves the Sudoku puzzles using the Scala language.	20	23620.25 ± 99.52
19.	scala-kmeans	Utilizes Scala collections to perform the k-means algorithm.	50	2375.70 ± 82.19
20.	scala-stm-bench7	Uses the ScalaSTM framework to run the STMBench7 benchmark (Guerraoui et al., 2006).	60	7775.83 ± 65.02
21.	scrabble	Solves the Scrabble puzzle using the Java streams.	50	943.07 ± 9.00

DaCapo con Scala suite, non-versioned

No.	Benchmark	Description	n	Execution Time (ms)
22.	factorie	Uses the Factorie toolkit (Passos et al., 2013; McCallum et al., 2009) for deployable probabilistic modeling to extract topics using Latent Dirichlet Allocation (Blei et al., 2001, 2003).	60	89362.71 ± 842.55
23.	kiama	Uses Kiama (Sloane, 2009), a Scala library for language processing, to compile and execute programs written in Oberon (Sloane and Roberts, 2015) and programs written in language that extends to Landin's ISWIM language (Landin, 1966).	40	2030.87 ± 59.83
24.	scalac	Uses the Scalac compiler (Van Der Leun, 2017) to compile the Scalap classfile decoder (Canonical Ltd, 2023).	12	7425.68 ± 143.23
25.	scaladoc	Uses the Scala documentation tool (École Polytechnique Fédérale Lausanne (EPFL) Lausanne, Switzerland, 2023) to generate API documentation for the Scalap classfile decoder.	12	8685.27 ± 123.49
26.	scalap	Uses a Scala class file decoder to decode several class files from the Scala library.	12	603.56 ± 9.82
27.	scalariform	Uses the Scalariform code formatter (École Polytechnique Fédérale Lausanne (EPFL) Lausanne, Switzerland, 2023) to format source codes of the Scala library.	30	2533.38 ± 13.24
28.	scalaxb	Uses the Scalaxb XML data-binding tool (ScalaXB org, 2023) to compile real-world W3C XML Schemas (Gao et al., 2012) to Scala code.	60	2991.35 ± 9.33

Table 12
The geometric standard deviation of the results aggregation across the benchmark suites in the experiments of measuring the runtime performance, binary size, and compile time impact of static profilers and the dynamic profiler.

Runtime Speedup						
	Wu&Larus	**DT**	**DNN**	**XGBoost**	**GraalSP**	**Dynamic Profiling**
DaCapo	1.05	1.08	1.04	1.04	1.04	1.16
Renaissance	1.17	1.12	1.10	1.10	1.08	1.26
DaCapo con Scala	1.08	1.06	1.05	1.07	1.03	1.18
Binary Size Increase						
	Wu&Larus	**DT**	**DNN**	**XGBoost**	**GraalSP**	**Dynamic Profiling**
DaCapo	1.03	1.01	1.01	1.01	1.02	1.20
Renaissance	1.04	1.01	1.01	1.01	1.02	1.26
DaCapo con Scala	1.19	1.01	1.01	1.01	1.01	1.11
Compile Time Increase						
	Wu&Larus	**DT**	**DNN**	**XGBoost**	**GraalSP**	**Dynamic Profiling**
DaCapo	1.07	1.03	1.05	1.03	1.05	1.06
Renaissance	1.08	1.02	1.05	1.05	1.04	1.06
DaCapo con Scala	1.11	1.04	1.08	1.08	1.04	1.07

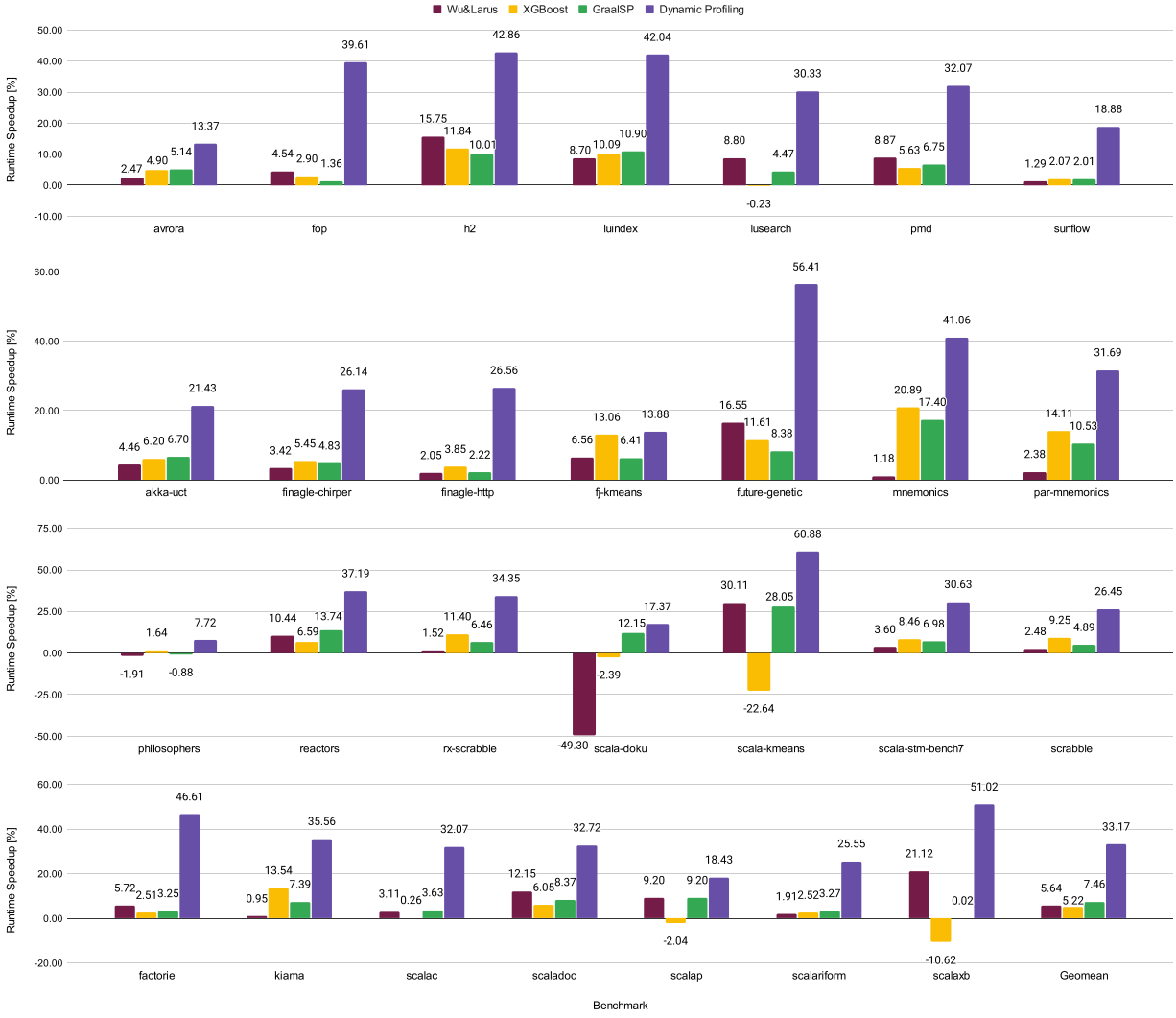


Fig. 22. Runtime speedup of GraalSP and profilers driven by Wu and Larus's heuristics, XGBoost model, and dynamically collected branches' profiles.

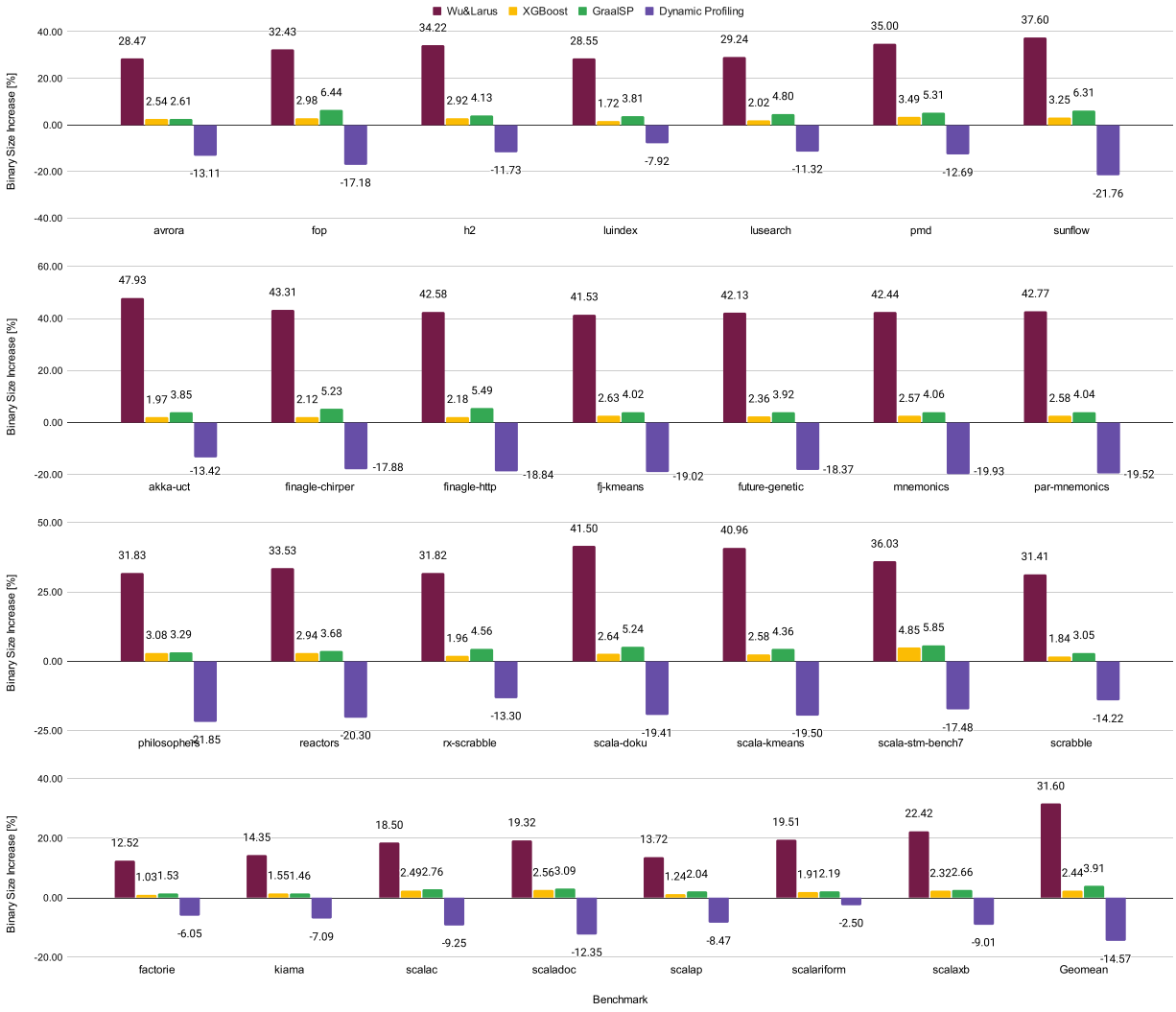


Fig. 23. Binary size increase of GraalSP and profilers driven by Wu and Larus's heuristics, XGBoost ML model, and dynamically collected branches profiles.

Fig. 24. Compile time increase of GraalSP and profilers driven by Wu and Larus's heuristics, XGBoost model, and dynamically collected branch profiles.

References

Abdi, H., Williams, L.J., 2010. Principal component analysis. Wiley Interdiscip. Rev.: Comput. Stat. 2 (4), 433–459.

Aggarwal, Charu C. (Ed.), 2014. Data classification algorithms and applications. CRC Press.

Aho, V.A., Lam, S.M., Ullman, J.D., 2007. Compilers Principles, Techniques and Tools. Pearson.

Alovisi, P., 2020. Static branch prediction through representation learning. https://www.diva-portal.org/smash/get/diva2:1450658/FULLTEXT01.pdf.

Alpaydin, E., 2020. Introduction To Machine Learning. MIT Press.

Aly, M., 2005. Survey on multiclass classification methods. Neural Netw. 19 (1–9), 2.

Amari, S., 1993. Backpropagation and stochastic gradient descent method. Neurocomputing 5 (4–5), 185–196.

Arnold, M., Ryder, B.G., 2001. A framework for reducing the cost of instrumented code. In: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation. pp. 168–179.

Ayers, A., Schooler, R., Gottlieb, R., 1997. Aggressive inlining. ACM SIGPLAN Notices 32 (5), 134–145.

Ball, T., Larus, J.R., 1993. Branch prediction for free. ACM SIGPLAN Notices 28 (6), 300–313.

Barbez, A., Khomh, F., Guéhéneuc, Y.-G., 2020. A machine-learning based ensemble method for anti-patterns detection. J. Syst. Softw. 161, 110486.

Bentéjac, C., Csörgő, A., Martínez-Muñoz, G., 2021. A comparative analysis of gradient boosting algorithms. Artif. Intell. Rev. 54, 1937–1967.

Berrar, D., 2019. Cross-validation. In: Encyclopedia of Bioinformatics and Computational Biology. ISBN: 978-0-12-811432-2, pp. 542–545. http://dx.doi.org/10.1016/B978-0-12-809633-8.20349-X.

Beyer, D., Löwe, S., Wendler, P., 2019. Reliable benchmarking: Requirements and solutions. Int. J. Softw. Tools Technol. Transf. 21 (1), 1–29.

Białecki, A., Muir, R., Ingersoll, G., Imagination, L., 2012. Apache Lucene 4. In: SIGIR 2012 Workshop on Open Source Information Retrieval. p. 17.

Biau, G., Scornet, E., 2016. A random forest guided tour. Test 25, 197–227.

Blackburn, S.M., Garner, R., Hoffmann, C., Khang, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., et al., 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. pp. 169–190.

Blei, D., Ng, A., Jordan, M., 2001. Latent dirichlet allocation. Adv. Neural Inf. Process. Syst. 14.

Blei, D.M., Ng, A.Y., Jordan, M.I., 2003. Latent dirichlet allocation. J. Mach. Learn. Res. 3 (Jan), 993–1022.

Boogerd, C., Moonen, L., 2006. Prioritizing software inspection results using static profiling. In: 2006 Sixth IEEE International Workshop on Source Code Analysis and Manipulation. IEEE, pp. 149–160.

Bottou, L., Curtis, F.E., Nocedal, J., 2018. Optimization methods for large-scale machine learning. SIAM Rev. 60 (2), 223–311.

Bračevac, O., Wei, G., Jia, S., Abeysinghe, S., Jiang, Y., Bao, Y., Rompf, T., 2023. Graph IRS for impure higher-order languages: making aggressive optimizations affordable with precise effect dependencies. In: Proceedings of the ACM on Programming Languages. Vol. 7, (OOPSLA2), pp. 400–430.

Brauckmann, A., Goens, A., Ertel, S., Castrillon, J., 2020. Compiler-based graph representations for deep learning models of code. In: Proceedings of the 29th International Conference on Compiler Construction. pp. 201–211.

Breiman, L., 2017. Classification and Regression Trees. Routledge.

Breiman, L., Friedman, J., Stone, C.J., Olshen, R.A., 1984. Classification and Regression Trees. CRC Press.

Brijain, M., Patel, R., Kushik, M., Rana, K., 2014. A survey on decision tree algorithm for classification. 2, (1), (ISSN: 2321-9939)

Bronson, N., the Scala STM Expert Group, 2023. Library-based software transactional memory for scala. https://nbronson.github.io/scala-stm/faq.html.

Bruno, R., Jovanovic, V., Wimmer, C., Alonso, G., 2021. Compiler-assisted object inlining with value fields. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. pp. 128–141.

Buse, R.P., Weimer, W., 2009. The road not taken: Estimating path execution frequency statically. In: 2009 IEEE 31st International Conference on Software Engineering. IEEE, pp. 144–154.

Calder, B., Grunwald, D., Jones, M., Lindsay, D., Martin, J., Mozer, M., Zorn, B., 1997. Evidence-based static branch prediction using machine learning. ACM Trans. Program. Lang. Syst. (TOPLAS) 19 (1), 188–222.

Canonical Ltd, 2023. Ubuntu manpages. https://manpages.ubuntu.com/.

Caruana, R., Niculescu-Mizil, A., 2006. An empirical comparison of supervised learning algorithms. In: Proceedings of the 23rd International Conference on Machine Learning. pp. 161–168.

Chandrashekar, G., Sahin, F., 2014. A survey on feature selection methods. Comput. Electr. Eng. 40 (1), 16–28.

Chang, P.P., Mahlke, S.A., Chen, W.Y., Warter, N.J., Hwu, W.-m.W., 1991. Impact: An architectural framework for multiple-instruction-issue processors. ACM SIGARCH Comput. Archit. News 19 (3), 266–275.

Chen, T., Guestrin, C., 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, New York, NY, USA, pp. 785–794. http://dx.doi.org/10.1145/2939672.2939785.

Chen, T., He, T., Benesty, M., Khotilovich, V., Tang, Y., Cho, H., Chen, K., Mitchell, R., Cano, I., Zhou, T., et al., 2015. Xgboost: extreme gradient boosting. R package version 0.4-2 1 (4), 1–4.

Chen, D., Li, D.X., Moseley, T., 2016. Autofdo: Automatic feedback-directed optimization for warehouse-scale applications. In: Proceedings of the 2016 International Symposium on Code Generation and Optimization. pp. 12–23.

Chen, D., Vachharajani, N., Hundt, R., Li, X., Eranian, S., Chen, W., Zheng, W., 2011. Taming hardware event samples for precise and versatile feedback directed optimizations. IEEE Trans. Comput. 62 (2), 376–389.

Chen, D., Vachharajani, N., Hundt, R., S.-w. Liao, Ramasamy, V., Yuan, P., Chen, W., Zheng, W., 2010. Taming hardware event samples for fdo compilation. In: Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization. pp. 42–52.

Cheng, W., Zhao, Q., Yu, B., Hiroshige, S., 2006. Tainttrace: Efficient flow tracing with dynamic binary rewriting. In: 11th IEEE Symposium on Computers and Communications. ISCC'06, IEEE, pp. 749–754.

Cho, H.K., Moseley, T., Hank, R., Bruening, D., Mahlke, S., 2013. Instant profiling: Instrumentation sampling for profiling datacenter applications. In: Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization. CGO, IEEE, pp. 1–10.

Choi, M., Park, J.H., Lim, S., Jeong, Y.-S., 2012. Achieving reliable system performance by fast recovery of branch miss prediction. J. Netw. Comput. Appl. 35 (3), 982–991.

Cummins, C., Fisches, Z.V., Ben-Nun, T., Hoefler, T., O'Boyle, M.F., Leather, H., 2021. Programl: A graph-based program representation for data flow analysis and compiler optimizations. In: International Conference on Machine Learning. PMLR, pp. 2244–2253.

Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K., 1991. Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Program. Lang. Syst. (TOPLAS) 13 (4), 451–490.

Dauphin, Y., De Vries, H., Bengio, Y., 2015. Equilibrated adaptive learning rates for non-convex optimization. Adv. Neural Inf. Process. Syst. 28.

Deitrich, B.L., Chen, B.C., Hwu, W., 1998. Improving static branch prediction in a compiler. In: Proceedings. 1998 International Conference on Parallel Architectures and Compilation Techniques (Cat. No. 98EX192). IEEE, pp. 214–221.

Demange, D., Fernández de Retana, Y., Pichardie, D., 2018. Semantic reasoning about the sea of nodes. In: Proceedings of the 27th International Conference on Compiler Construction. pp. 163–173.

Desmet, V., Eeckhout, L., De Bosschere, K., 2005. Using decision trees to improve program-based and profile-based static branch prediction. In: Asia-Pacific Conference on Advances in Computer Systems Architecture. Springer, pp. 336–352.

Detlefs, D., Agesen, O., 1999. Inlining of virtual methods. In: European Conference on Object-Oriented Programming. Springer, pp. 258–277.

Dhal, P., Azad, C., 2022. A comprehensive survey on feature selection in the various fields of machine learning. Appl. Intell. 1–39.

Dietterich, T., 1995. Overfitting and undercomputing in machine learning. ACM Comput. Surv. (CSUR) 27 (3), 326–327.

DuBois, P., 2013. MySQL. Addison-Wesley Professional.

Duboscq, G., Stadler, L., Würthinger, T., Simon, D., Wimmer, C., Mössenböck, H., 2013. Graal ir: An extensible declarative intermediate representation. In: Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop.

École Polytechnique Fédérale Lausanne (EPFL) Lausanne, Switzerland, 2023. Scala documentation. https://www.scala-lang.org/.

Epic Games Inc., 2022. Unreal engine 5.0 documentation. https://docs.unrealengine.com/5.0/en-US/.

Fagan, M., 2011. Design and code inspections to reduce errors in program development. In: Software Pioneers: Contributions To Software Engineering. Springer, pp. 575–607.

Feurer, M., Hutter, F., 2019. Hyperparameter optimization. In: Automated Machine Learning: Methods, Systems, Challenges. pp. 3–33.

Fisher, J.A., Freudenberger, S.M., 1992. Predicting conditional branch directions from previous runs of a program. ACM SIGPLAN Notices 27 (9), 85–95.

Fleming, P.J., Wallace, J.J., 1986. How not to lie with statistics: the correct way to summarize benchmark results. Commun. ACM 29 (3), 218–221.

Friedman, J., Hastie, T., Tibshirani, R., 2001. The Elements of Statistical Learning. Vol. 1, Springer series in statistics, New York.

Gao, S.S., Sperberg-McQueen, C.M., Thompson, H., 2012. W3C XML schema definition language (XSD) 1.1 part 1: Structures.

Ghosh, D., Vogt, A., 2012. Outliers: An evaluation of methodologies. In: Joint Statistical Meetings. Vol. 12, pp. 3455–3460.

Gilmer, J., Schoenholz, S.S., Riley, P.F., Vinyals, O., Dahl, G.E., 2017. Neural message passing for quantum chemistry. In: International Conference on Machine Learning. PMLR, pp. 1263–1272.

GithHub, 2024a. Source code of the indexdocs function from the dacapo luindex benchmark. https://github.com/dacapobench/dacapobench/blob/fd292e92f8c40495a6ca05ff3b8a77c6c4265606/benchmarks/bms/luindex/src/org/dacapo/luindex/Index.java#L177.

GithHub, 2024b. Source code of the kmeans function from the renaissance sala-kmeans benchmark. https://github.com/renaissance-benchmarks/renaissance/blob/c7209b4e2f6c4dcb2604b9daa273ce0eb1cbe85f/benchmarks/scala-stdlib/src/main/scala/org/renaissance/scala/stdlib/ScalaKmeans.scala#L103.

Glassner, A.S., 1989. An Introduction To Ray Tracing. Morgan Kaufmann.

Gong, Y., Liu, G., Xue, Y., Li, R., Meng, L., 2023. A survey on dataset quality in machine learning. Inf. Softw. Technol. 162, 107268. http://dx.doi.org/10.1016/j.infsof.2023.107268.

Google Corporation, 2023. Chromium documentation. https://chromium.googlesource.com/chromium/src/+/master/docs/README.md.

Gordon, J., Shortliffe, E.H., 1984. The dempster-shafer theory of evidence. In: Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project. Vol. 3, pp. 832–838.

Graham, S.L., Kessler, P.B., McKusick, M.K., 1982. Gprof: A call graph execution profiler. ACM Sigplan Notices 17 (6), 120–126.

Greff, K., Srivastava, R.K., Koutník, J., Steunebrink, B.R., Schmidhuber, J., 2016. Lstm: A search space odyssey. IEEE Trans. Neural Netw. Learn. Syst. 28 (10), 2222–2232.

Guerraoui, R., Kapalka, M., Vitek, J., 2006. Stmbench7: a benchmark for software transactional memory. Tech. rep..

Hahn, G.J., 1973. The coefficient of determination exposed. Chemtech 3 (10), 609–612.

Hassoun, M.H., et al., 1995. Fundamentals of Artificial Neural Networks. MIT Press.

Hastie, T., Tibshirani, R., Friedman, J.H., Friedman, J.H., 2009. The Elements of Statistical Learning: Data Mining, Inference, and Prediction. vol. 2, Springer.

Hawkins, D.M., 2004. The problem of overfitting. J. Chem. Inf. Comput. Sci. 44 (1), 1–12.

He, W., Mestre, J., Pupyrev, S., Wang, L., Yu, H., 2022. Profile inference revisited. In: Proceedings of the ACM on Programming Languages. Vol. 6, (POPL), pp. 1–24.

He, K., Zhang, X., Ren, S., Sun, J., 2016. Deep residual learning for image recognition. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. pp. 770–778.

HSQLDB Group, 2023. H2 database engine. https://www.h2database.com/html/main.html.

Huang, L., Joseph, A.D., Nelson, B., Rubinstein, B.I., Tygar, J.D., 2011. Adversarial machine learning. In: Proceedings of the 4th ACM Workshop on Security and Artificial Intelligence. pp. 43–58.

Hwang, Y.-S., Chen, P.-S., Lee, J.K., Ju, R.D.-C., 2003. Probabilistic points-to analysis. In: Languages and Compilers for Parallel Computing: 14th International Workshop, LCPC 2001, Cumberland Falls, KY, USA, August 1–3, 2001, Revised Papers 14. Springer, pp. 290–305.

Imam, S.M., Sarkar, V., 2014. Savina-an actor benchmark suite: Enabling empirical evaluation of actor libraries. In: Proceedings of the 4th International Workshop on Programming Based on Actors Agents & Decentralized Control. pp. 67–80.

Jain, A., Patel, H., Nagalapatti, L., Gupta, N., Mehta, S., Guttula, S., Mujumdar, S., Afzal, S., Sharma Mittal, R., Munigala, V., 2020. Overview and importance of data quality for machine learning tasks. In: Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. pp. 3561–3562.

Johnson, S., Valli, S., 2008. Hot method prediction using support vector machines. Ubiquitous Comput. Commun. J. 3 (4), 75–81.

Keller, J.M., Gray, M.R., Givens, J.A., 1985. A fuzzy k-nearest neighbor algorithm. IEEE Trans. Syst. Man Cybern. (4), 580–585.

Kennedy, K., Allen, J.R., 2001. Optimizing Compilers for Modern Architectures: A Dependence-Based Approach. Morgan Kaufmann Publishers Inc..

Kingma, D.P., Ba, J., 2014. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.

Kleen, A., 2005. A NUMA API for Linux. Novel Inc, https://halobates.de/numaapi3.pdf.

Kotsiantis, S.B., Kanellopoulos, D., Pintelas, P.E., 2006. Data preprocessing for supervised leaning. Int. J. Comput. Sci. 1 (2), 111–117.

Kotsiantis, S.B., Zaharakis, I., Pintelas, P., et al., 2007. Supervised machine learning: A review of classification techniques. Emerg. Artif. Intell. Appl. Comput. Eng. 160 (1), 3–24.

Kulla, C., 2023. Sunflow render system documentation. https://sunflow.sourceforge.net/.

Kumar, V., Minz, S., 2014. Feature selection: a literature review. SmartCR 4 (3), 211–229.

Kunikowski, W., Czerwiński, E., Olejnik, P., Awrejcewicz, J., 2015. An overview of atmega avr microcontrollers used in scientific research and industrial applications. Pomiary Automatyka Robotyka 19 (1), 15–19.

Kurakin, A., Goodfellow, I.J., Bengio, S., 2016. Adversarial machine learning at scale. In: International Conference on Learning Representations.

Kurita, T., 2019. Principal component analysis (pca). In: Computer Vision: A Reference Guide. pp. 1–4.

Landin, P.J., 1966. The next 700 programming languages. Commun. ACM 9 (3), 157–166.

Lattner, C., 2008. LLVM and clang: Next generation compiler technology. In: The BSD Conference. vol. 5, pp. 1–20.

Lattner, C., 2020. LLVM and API reference documentation: How to build Clang and LLVM with profile-guided optimizations. https://llvm.org/docs/HowToBuildWithPGO.html.

LeCun, Y., Bengio, Y., Hinton, G., 2015. Deep learning. nature 521 (7553), 436–444.

Lehmann, D., Rabin, M.O., 1981. On the advantages of free choice: A symmetric and fully distributed solution to the dining philosophers problem. In: Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 133–138.

Leopoldseder, D., 2017. Simulation-based code duplication for enhancing compiler optimizations. In: Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity. pp. 10–12.

Leopoldseder, D., Stadler, L., Rigger, M., Würthinger, T., Mössenböck, H., 2018a. 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. pp. 26–35.

Leopoldseder, D., Stadler, L., Wimmer, C., Mössenböck, H., 2015. Java-to-JavaScript translation via structured control flow reconstruction of compiler IR. ACM SIGPLAN Notices 51 (2), 91–103.

Leopoldseder, D., Stadler, L., Würthinger, T., Eisl, J., Simon, D., Mössenböck, H., 2018b. Dominance-based duplication simulation (dbds): code duplication to enable compiler optimizations. In: Proceedings of the 2018 International Symposium on Code Generation and Optimization. pp. 126–137.

Lerman, R.I., Yitzhaki, S., 1984. A note on the calculation and interpretation of the gini index. Econom. Lett. 15 (3–4), 363–368.

Li, J., Cheng, K., Wang, S., Morstatter, F., Trevino, R.P., Tang, J., Liu, H., 2017. Feature selection: A data perspective. ACM Comput. Surv. 50 (6), 1–45.

Li, Y., Zemel, R., Brockschmidt, M., Tarlow, D., 2016. Gated graph sequence neural networks. In: Proceedings of ICLR'16.

Liu, Y., Gao, Y., Yin, W., 2020. An improved analysis of stochastic gradient descent with momentum. Adv. Neural Inf. Process. Syst. 33, 18261–18271.

Liu, H., Glew, N., Petersen, L., Anderson, T.A., 2013. The intel labs haskell research compiler. In: Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell. pp. 105–116.

Liu, L., Jiang, H., He, P., Chen, W., Liu, X., Gao, J., Han, J., 2019. On the variance of the adaptive learning rate and beyond. In: International Conference on Learning Representations.

Lloyd, S., 1982. Least squares quantization in pcm. IEEE Trans. Inf. Theory 28 (2), 129–137.

LLVM Project, 2024. LLVM language reference manual. https://llvm.org/docs/LangRef.html.

Lucene, Apache, 2010. Apache Lucene-overview. http://lucene.apache.org/iava/docs/.

Luk, C., Muth, R., Patil, H., Cohn, R., Lowney, G., 2004. Ispike: a post-link optimizer for the intel/spl reg/itanium/spl reg/architecture. In: International Symposium on Code Generation and Optimization, 2004. CGO 2004. IEEE, pp. 15–26.

Maćkiewicz, A., Ratajczak, W., 1993. Principal components analysis (pca). Comput. Geosci. 19 (3), 303–342.

MacQueen, J., et al., 1967. Some methods for classification and analysis of multivariate observations. In: Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability. vol. 1, Oakland, CA, USA, pp. 281–297.

Mahapatra, A., Patra, P.K., 2018. Support vector machine for frequently executed method prediction. In: 2018 International Conference on Information Technology. ICIT, IEEE, pp. 193–198.

Manikandan, S., 2011. Measures of central tendency: The mean. J. Pharmacol. Pharmacother. 2 (2), 140.

Masud, A.N., Ciccozzi, F., 2020. More precise construction of static single assignment programs using reaching definitions. J. Syst. Softw. 166, 110590. http://dx.doi.org/10.1016/j.jss.2020.110590.

McCallum, A., Schultz, K., Singh, S., 2009. Factorie: Probabilistic programming via imperatively defined factor graphs. Adv. Neural Inf. Process. Syst. 22.

Medsker, L., Jain, L.C., 1999. Recurrent Neural Networks: Design and Applications. CRC Press.

Microsoft Corporation, 2021. Onnx Java Runtime. https://onnxruntime.ai/.

Mitchell, T.M., 2007. Machine Learning. vol. 1, McGraw-hill, New York.

Moreira, A.A., Ottoni, G., Quintão Pereira, F.M., 2021. Vespa: static profiling for binary optimization. In: Proceedings of the ACM on Programming Languages. Vol. 5, (OOPSLA), pp. 1–28.

Morgan, R., 1998. Building an Optimizing Compiler. Digital Press.

Mosaner, R., Barany, G., Leopoldseder, D., Mössenböck, H., 2022. Improving vectorization heuristics in a dynamic compiler with machine learning models. In: Proceedings of the 14th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages. pp. 36–47.

Moseley, T., Shye, A., Reddi, V.J., Grunwald, D., Peri, R., 2007. Shadow profiling: Hiding instrumentation costs with parallelism. In: International Symposium on Code Generation and Optimization. CGO'07, IEEE, pp. 198–208.

Mozilla Foundation, 2023. Firefox source tree documentation. https://firefox-source-docs.mozilla.org/index.html.

Murphy, K.P., 2022. Probabilistic Machine Learning: An Introduction. MIT Press.

Nagelkerke, N.J., 1991. A note on a general definition of the coefficient of determination. Biometrika 78 (3), 691–692.

Nethercote, N., Seward, J., 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. ACM Sigplan notices 42 (6), 89–100.

Nilsson, H., Courtney, A., Peterson, J., 2002. Functional reactive programming, continued. In: Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell. pp. 51–64.

Novillo, D., 2014. SamplePGO-the power of profile guided optimizations without the usability burden. In: 2014 LLVM Compiler Infrastructure in HPC. IEEE, pp. 22–28.

Oracle Corporation, 2023. Graalvm reachability metadata repository. https://github.com/oracle/graalvm-reachability-metadata.

Oracle Corporation, 2024. Ideal graph visualizer. https://www.graalvm.org/latest/tools/igv/.

Ortiz, H.V., Mishra, P., 2017. Akka Cookbook. Packt Publishing Ltd.

Osisanwo, F., Akinsola, J., Awodele, O., Hinmikaiye, J., Olakanmi, O., Akinjobi, J., et al., 2017. Supervised machine learning algorithms: classification and comparison. Int. J. Comput. Trends Technol. (IJCTT) 48 (3), 128–138.

Ottoni, G., Maher, B., 2017. Optimizing function placement for large-scale data-center applications. In: 2017 IEEE/ACM International Symposium on Code Generation and Optimization. CGO, IEEE, pp. 233–244.

Panchenko, M., Auler, R., Nell, B., Ottoni, G., 2019. Bolt: a practical binary optimizer for data centers and beyond. In: 2019 IEEE/ACM International Symposium on Code Generation and Optimization. CGO, IEEE, pp. 2–14.

Passos, A., Vilnis, L., McCallum, A., 2013. Optimization and learning in factorie. In: NIPS Workshop on Optimization for Machine Learning. OPT, Citeseer.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., Chintala, S., 2019. Pytorch: An imperative style, high-performance deep learning library. In: Advances in Neural Information Processing Systems. Vol. 32, Curran Associates, Inc., pp. 8024–8035.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., et al., 2011. Scikit-learn: Machine learning in python. J. Mach. Learn. Res. 12, 2825–2830.

Pettis, K., Hansen, R.C., 1990. Profile guided code positioning. SIGPLAN Not. 25 (6), 16–27. http://dx.doi.org/10.1145/93548.93550.

PMD Open Source Project, 2023. PMD documentation. https://docs.pmd-code.org/latest/index.html.

Prokopec, A., Duboscq, G., Leopoldseder, D., Wïrthinger, T., 2019a. An optimization-driven incremental inline substitution algorithm for just-in-time compilers. In: 2019 IEEE/ACM International Symposium on Code Generation and Optimization. CGO, IEEE, pp. 164–179.

Prokopec, A., Rosà, A., Leopoldseder, D., Duboscq, G., Tuma, P., Studener, M., Bulej, L., Zheng, Y., Villazón, A., Simon, D., et al., 2019b. Renaissance: benchmarking suite for parallel applications on the JVM. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 31–47.

ReactiveX, 2023. ReactiveX documentation. https://reactivex.io/documentation/observable.html.

Rodríguez, P., Bautista, M.A., Gonzalez, J., Escalera, S., 2018. Beyond one-hot encoding: Lower dimensional target embedding. Image Vis. Comput. 75, 21–31.

Rokach, L., Maimon, O.Z., 2007. Data Mining with Decision Trees: Theory and Applications. Vol. 69, World scientific.

Rotem, N., Cummins, C., 2021. Profile guided optimization without profiles: A machine learning approach. arXiv preprint arXiv:2112.14679.

Sagi, O., Rokach, L., 2018. Ensemble learning: A survey. Wiley Interdiscip. Rev.: Data Min. Knowl. Discov. 8 (4), e1249.

Scala Community, 2023. Reactors documentation. http://reactors.io/learn/.

ScalaXB org, 2023. Scalaxb. https://scalaxb.org/.

Scheifler, R.W., 1977. An analysis of inline substitution for a structured programming language. Commun. ACM 20 (9), 647–654.

Sewe, A., Mezini, M., Sarimbekov, A., Binder, W., 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. pp. 657–676.

Shahhosseini, M., Hu, G., Pham, H., 2022. Optimizing ensemble weights and hyperparameters of machine learning models for regression problems. Mach. Learn. Appl. 7, 100251.
Shalev-Shwartz, S., Ben-David, S., 2014. Understanding Machine Learning: From Theory To Algorithms. Cambridge University Press.
Shih, C.-Y., Svoboda, D., Su, S.-J., Liao, W.-C., 2021. Static branch prediction for llvm using machine learning. https://drakesvoboda.com/StaticBranchPrediction.pdf.
Sloane, A.M., 2009. Lightweight language processing in kiama. In: International Summer School on Generative and Transformational Techniques in Software Engineering. Springer, pp. 408–425.
Sloane, A.M., Roberts, M., 2015. Oberon-0 in kiama. Sci. Comput. Program. 114, 20–32.
Srirama, S.N., Dick, F.M.S., Adhikari, M., 2021. Akka framework based on the Actor model for executing distributed Fog Computing applications. Future Gener. Comput. Syst. 117, 439–452.
Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., Salakhutdinov, R., 2014. Dropout: a simple way to prevent neural networks from overfitting. J. Mach. Learn. Res. 15 (1), 1929–1958.
Stadler, L., Duboscq, G., Mössenböck, H., Würthinger, T., Simon, D., 2013. An experimental study of the influence of dynamic compiler optimizations on scala performance. In: Proceedings of the 4th Workshop on Scala. pp. 1–8.
Stadler, L., Würthinger, T., Mössenböck, H., 2014. Partial escape analysis and scalar replacement for Java. In: Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization. pp. 165–174.
Stallman, R.M., et al., 2003. Using the Gnu Compiler Collection. 4, (02), Free Software Foundation.
Stallman, R.M., et al., 2012. Using the GNU Compiler Collection. Free Software Foundation.
Steidl, M., Felderer, M., Ramler, R., 2023. The pipeline for the continuous development of artificial intelligence models—current state of research and practice. J. Syst. Softw. 199, 111615.
Steinwart, I., Christmann, A., 2008. Support Vector Machines. Springer Science & Business Media.
Stephenson, M., Amarasinghe, S., 2004. Predicting unroll factors using nearest neighbors.
Stone, M., 1978. Cross-validation: A review. Statistics 9 (1), 127–139.
Sun, Q., Zhao, J., Chen, Y., 2011. Probabilistic points-to analysis for java. In: Compiler Construction: 20th International Conference, CC 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26–April 3 2011. Proceedings 20. Springer, pp. 62–81.
Tuchler, M., Singer, A.C., Koetter, R., 2002. Minimum mean squared error equalization using a priori information. IEEE Trans. Signal Process. 50 (3), 673–683.
Twiter Inc, 2023. Twiter finagle. https://twitter.github.io/finagle/.
Van Der Leun, V., 2017. Introduction to JVM Languages. Packt Publishing Ltd.
Van Dyk, D.A., Meng, X.L., 2001. The art of data augmentation. J. Comput. Graph. Statist. 10 (1), 1–50.
Wang, Z., Bovik, A.C., 2009. Mean squared error: Love it or leave it? a new look at signal fidelity measures. IEEE Signal Process. Mag. 26 (1), 98–117.
Whaley, J., 2000. A portable sampling-based profiler for java virtual machines. In: Proceedings of the ACM 2000 Conference on Java Grande. pp. 78–87.
Wicht, B., Vitillo, R.A., Chen, D., Levinthal, D., 2014. Hardware counted profile-guided optimization. arXiv preprint arXiv:1411.6361.
Wilhelmstötter, F., 2021. Jenetics. http://jenetics.io.
Wimmer, C., Stancu, C., Hofer, P., Jovanovic, V., Wögerer, P., Kessler, P.B., Pliss, O., Würthinger, T., 2019. Initialize once, start fast: application initialization at build time. In: Proceedings of the ACM on Programming Languages. Vol. 3, (OOPSLA), pp. 1–29.
Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A., 2012. Experimentation in Software Engineering. Springer Science & Business Media.
Wold, S., Esbensen, K., Geladi, P., 1987. Principal component analysis. Chemometr. Intell. Labor. Syst. 2 (1–3), 37–52.
Wong, W.-F., 1999. Source level static branch prediction. Comput. J. 42 (2), 142–149.
Wright, R.E., 1995. Logistic regression. In: Reading and understanding multivariate statistics. pp. 217–244.
Wu, Y., Larus, J.R., 1994. Static branch frequency and program profile analysis. In: Proceedings of the 27th Annual International Symposium on Microarchitecture. pp. 1–11.
Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., Philip, S.Y., 2020. A comprehensive survey on graph neural networks. IEEE Trans. Neural Netw. Learn. Syst. 32 (1), 4–24.
Wu, Z., Shen, C., Van Den Hengel, A., 2019. Wider or deeper: Revisiting the resnet model for visual recognition. Pattern Recognit. 90, 119–133.
Würthinger, T., 2011. Extending the graal compiler to optimize libraries. In: Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion. pp. 41–42.
Yang, L., Shami, A., 2020. On hyperparameter optimization of machine learning algorithms: Theory and practice. Neurocomputing 415, 295–316.
Yu, S., Wang, T., Wang, J., 2022. Data augmentation by program transformation. J. Syst. Softw. 190, 111304. http://dx.doi.org/10.1016/j.jss.2022.111304.
Zekany, S., Rings, D., Harada, N., Laurenzano, M.A., Tang, L., Mars, J., 2016. Crystalball: Statically analyzing runtime behavior via deep sequence learning. In: 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO, IEEE, pp. 1–12.
Zhang, C., Xu, H., Zhang, S., Zhao, J., Chen, Y., 2011. Frequency estimation of virtual call targets for object-oriented programs. In: European Conference on Object-Oriented Programming. Springer, pp. 510–532.
Zhao, X., Jamali, N., 2013. Load balancing non-uniform parallel computations. In: Proceedings of the 2013 Workshop on Programming Based on Actors, Agents, and Decentralized Control. pp. 97–108.
Zhou, J., Cui, G., Hu, S., Zhang, Z., Yang, C., Liu, Z., Wang, L., Li, C., Sun, M., 2020. Graph neural networks: A review of methods and applications. AI Open 1, 57–81.

Milan Čugurović is a Ph.D. candidate at the Faculty of Mathematics, University of Belgrade. He is also a researcher at Oracle Labs, Serbia. He received a B.Sc and M.Sc in mathematics from the Faculty of Mathematics, University of Belgrade. His research interests include Machine Learning and Compilers.

Milena Vujošević Janičić is an Associate Professor at the Department of Computer Science and Head of the Computer Science Study Program at the Faculty of Mathematics, University of Belgrade. She also works as a principal researcher at Oracle Labs, Serbia. She received a Ph.D. degree in Computer Science from the University of Belgrade. Her main research interests are in compilers, automated bug finding, and the application of software verification techniques in different fields.

Vojin Jovanović works at Oracle Labs, Switzerland on GraalVM Native Image. Vojin co-designed profile-guided optimizations, profile inference, class initialization, reflection, the testing suite, the build tools, and the benchmarking infrastructure. His main research interests include compilers and virtual machines. He received a Ph.D. degree in the Scala Lab at the Ecole Polytechnique Fédérale de Lausanne under the supervision of Prof. Martin Odersky.

Thomas Würthinger is a Vice President at Oracle Labs Switzerland and the founder and director of the Graal Compiler project. His research interests include Virtual Machines, Feedback-directed Runtime Optimizations, and Static Program Analysis. He received a Ph.D. degree from the Johannes Kepler University Linz.