JOHANNES KEPLER
UNIVERSITY LINZ

Submitted by
DI Raphael Mosaner, BSc

Submitted at
Institute for System Software

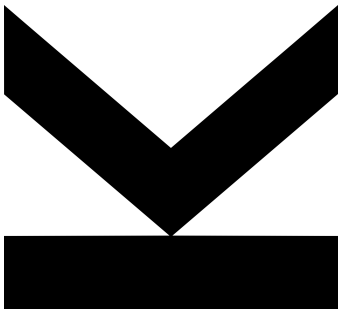
Supervisor and
First Evaluator
o. Univ.-Prof. DI Dr. Dr. h.c. Hanspeter Mössenböck

Second Evaluator
a. Univ.-Prof. DI Dr. Andreas Krall

Co-Supervisor
Dr. David Leopoldseder

April, 2023

Machine-Learning-Based Optimization Heuristics in Dynamic Compilers



Doctoral Thesis

to obtain the academic degree of

Doktor der technischen Wissenschaften

in the Doctoral Program

Technische Wissenschaften

JOHANNES KEPLER UNIVERSITY LINZ
Altenbergerstraße 69
4040 Linz, Austria
www.jku.at
DVR 0093696

Statutory Declaration

I hereby declare that the thesis submitted is my own unaided work, that I have not used other than the sources indicated, and that all direct and indirect sources are acknowledged as references.

Linz, April 28, 2023

DI Raphael Mosaner, BSc

Abstract

Modern, optimizing compilers rely on hundreds of heuristics to decide whether and how to apply optimizations during compilation. These heuristics are typically hand-crafted and designed to achieve good performance on a set of benchmark programs which are based on real-world applications. This manual process has multiple shortcomings: It requires experienced compiler engineers as well as continuous re-evaluation and tweaking of heuristics if compiler internals change. In addition, maintaining multiple, domain-specific heuristics is often considered infeasible, which results in the deployment of one-size-fits-all heuristics.

Data-driven approaches, based on machine learning, have been shown to outperform hand-crafted compiler heuristics in that they can suggest optimization decisions which yield better performing programs after compilation. Nevertheless, only a single recent machine learning approach has made it into a production-level compiler. Existing research primarily focuses on using machine learning in static compilers, where the compilation process is stable and reproducible. In contrast, dynamic compilers run in parallel to the executed program where they compile frequently executed code. On top of that, the use of profiling-based speculative optimizations adds an additional level of non-determinism to dynamic compilation. This lack of consistency aggravates the use of data-driven approaches in such systems.

In this thesis, we make multiple contributions to the use of machine learning in dynamic compilers. We address the problems of data stability and consistency by proposing compilation forking, a technique for extracting performance data from method-local optimization decisions in arbitrary programs. Based on this data, we train several machine learning models to replace optimization heuristics for loop optimizations, such as peeling, unrolling or vectorization. Furthermore, we present self-optimizing compiler heuristics, based on machine learning models which are continuously refined with new data at the user site. This approach also enables the creation of domain-specific heuristics which we showed to outperform one-size-fits-all heuristics significantly. Apart from deploying machine learning in production systems, we also outline how models can assist compiler engineers during heuristics design and evaluation.

We implemented our approaches in the GraalVM compiler, which is among the most highly optimizing Java compilers on the market. Our learned models are either on par with the well-tuned heuristics in the GraalVM or outperform them significantly with only minor regressions. In addition to that, our assistive approach enabled improvements in existing heuristics without the actual deployment of machine learning models in production systems.

Kurzfassung

Moderne optimierende Compiler verwenden hunderte von Heuristiken um zu entscheiden, ob und wie Optimierungen während einer Compilation durchgeführt werden sollen. Diese Heuristiken werden typischerweise von Hand geschrieben und sind dahingehend optimiert, dass sie für gewisse Referenzprogramme, die auf realen Programmen basieren, gute Ergebnisse liefern. Dieser manuelle Prozess hat aber mehrere Nachteile: Das Erstellen von Heuristiken erfordert erfahrene Compiler-Ingenieure, die diese Heuristiken nach Änderungen im Compiler ständig neu bewerten und anpassen müssen. Außerdem ist es meist nicht praktikabel, mehrere Heuristiken für verschiedene Anwendungsdomänen zu warten, weshalb Heuristiken in der Regel allgemein gehalten sind.

In der Literatur wurde bereits gezeigt, dass datengetriebene Ansätze, die auf maschinellem Lernen basieren, zu besseren Optimierungsentscheidungen führen als händisch erzeugte Heuristiken, was die Performanz von übersetzten Programmen verbessert. Trotzdem hat es bisher nur ein einziger Ansatz mit maschinellem Lernen in einen bekannten, kommerziell genutzten Compiler geschafft. Existierende Forschungsarbeiten konzentrieren sich vor allem auf statische Compiler, bei denen der Übersetzungsprozess sehr stabil und reproduzierbar ist. Im Gegensatz dazu übersetzen dynamische Compiler Codeteile zu nicht vorhersehbaren Zeitpunkten während der Programmausführung. Zusätzlich führen spekulative Optimierungen, die auf Nutzungsprofilen basieren, zu einem gewissen Nicht-Determinismus im Compiler. Der Mangel an Reproduzierbarkeit erschwert den Einsatz datengetriebener Ansätze in solchen Systemen.

In dieser Dissertation tragen wir in mehrfacher Hinsicht zum Einsatz von maschinellem Lernen in dynamischen Compilern bei. Wir sorgen für die Stabilität und Konsistenz von dynamischer Übersetzung indem wir *Compilation Forking* vorstellen, das es uns erlaubt, die Effektivität von Optimierungsentscheidungen innerhalb von Methoden beliebiger Programme zu messen. Basierend auf diesen Messdaten, trainieren wir verschiedene Modelle mittels maschinellem Lernen und ersetzen damit manuelle Heuristiken für Schleifenoptimierungen, wie Peeling, Unrolling und Vektorisierung. Ferner präsentieren wir einen Ansatz, um Heuristiken fortlaufend zu optimieren, indem das dahinterstehende Modell wiederholt mit neuen Daten zur Laufzeit trainiert und verbessert wird. Dieser

Ansatz erlaubt es auch, Heuristiken für spezielle Anwendungsdomänen zu erzeugen, die, wie wir zeigen konnten, bessere Ergebnisse liefern als allgemein gehaltene Heuristiken. Neben dem tatsächlichen Einsatz von maschinell trainierten Modellen zur Laufzeit, zeigen wir auch, wie diese Modelle Ingenieuren helfen können, die bestehenden Heuristiken zu verbessern und zu evaluieren.

Alle unsere Ansätze wurden im GraalVM Compiler implementiert, einem der besten optimierenden Java Compiler am Markt. Unsere trainierten Modelle liefern zumindest ähnliche Ergebnisse wie die manuell optimierten Heuristiken im GraalVM Compiler; oft sind unsere Modelle signifikant besser und nur selten etwas schlechter. Zusätzlich unterstützt unser Ansatz Ingenieure bei der Verbesserungen der manuell erstellten Heuristiken, was auch ohne die Verwendung von maschinell gelernten Modellen nützlich ist.

Contents

Part I

Introduction and Overview

Chapter 1

Introduction

1.1 Problem Setting

Dynamic compilation [12; 42] is a technique to compile and optimize programs *just-in-time* (JIT), while they are executed. In contrast to static compilation, where programs are compiled *ahead-of-time* (AOT), dynamic compilers can usually rely on run-time information that is collected prior to compilation and is used to apply optimizations based on the program's behavior.

Dynamic Compilation In a dynamic execution environment, such as the Java Virtual Machine (JVM) [71], a program is initially compiled with a simple baseline compiler or directly executed in an interpreter. This results in poor performance during program start-up—called *warmup* [14]—during which the runtime environment also gathers profiling information, such as function invocation counts, loop frequencies and branch probabilities. When program parts are considered *hot*, i.e. frequently executed and therefore worth to be optimized, the runtime environment invokes a dynamic compiler. This triggers an optimized compilation of the hot program parts during which the previously collected profiling information is used to apply aggressive and often speculative optimizations. If, for example, a dynamic call site only encountered one concrete receiver type `T` during interpretation, the dynamic dispatch in the compiled code can be replaced with a concrete call to the method of `T`. The assumption, that the callee is always of type `T`, needs to hold during execution, which is ensured by a *guard*. If, at some point, this assumption is invalidated, deoptimization [70] switches back to an interpreted version of the code and typically triggers a re-compilation with the updated assumptions, or re-profiles the code in the interpreter. At some point, dynamically compiled programs typically reach a steady state of peak performance where no more compilations or optimizations are performed. This marks the end of the program warmup [14].

Compiler Optimizations Compilers, dynamic or static, are equipped with a multitude of optimizations, developed over decades of research. Optimizations are semantic-preserving transformations of the code or an intermediate representation of it during compilation. Their predominant goal is to increase the performance of the compiled program, which can be a higher throughput or a reduced execution time. In achieving this goal, trade-offs between performance gain and code size growth, or, in case of dynamic compilers, compile time increases, which impact the program warmup, have to be made. For example, loop unrolling reduces the number of how often the loop condition has to be evaluated, but the code size increase of this transformation can pose problems for subsequent optimizations.

Compiler Heuristics When and how to apply compiler optimizations is subject to compiler heuristics. These heuristics can take static code structure and dynamic profiling information into account and produce an optimization decision. For example, depending on the number of instructions inside a loop and the profiled probability of entering the loop, loop peeling [13] or unrolling [13] might be either applied or omitted. Compiler heuristics are typically hand-crafted. Their design process is iterative where compiler experts define conditions and thresholds for optimizations and evaluate their impact on the performance with a set of benchmark programs. These benchmark programs are selected with the goal to correlate with real-world programs.

The research presented in this thesis focuses on dynamic compilation systems.

1.2 Problem Statement

Hand-crafted compiler heuristics suffer from multiple shortcomings. Designing heuristics correctly requires comprehensive experience in compilers and optimizations. Loop peeling [13], for instance, is a transformation which moves one or more loop iterations to before the loop. Applying this transformation to the loop in Listing 1.1 would enable subsequent optimizations which eventually lead to the simplified and more efficient code in Listing 1.2.

```
1 int redundant = 0;
2 for (int i = 0 ; i < 100; i++ ) {
3     dst[i] = src[i] + redundant;
4     redundant = i;
5 }
```

Listing 1.1: Loop with redundant variable before peeling.

```
1 dst[0] = src[0];
2 for (int i = 1 ; i < 100; i++ ) {
3     dst[i] = src[i] + (i-1);
4
5 }
```

Listing 1.2: Loop with redundant variable removed after peeling.

Keeping track of all follow-up optimizations while designing a heuristic (e.g., whether to peel a loop or not) is tedious if done by hand. Changes in the compiler, integration of new optimizations or re-ordering of existing phases requires evaluating and updating established heuristics [4].

Optimizations and heuristics are typically evaluated on a set of benchmark programs of manageable size. These are either micro-benchmarks, where the impact of specific optimizations can be easily investigated but interplay with other optimizations is neglected, or larger programs where aggregated performance impacts are reported but internal speedups and slowdowns might have cancelled out. In addition, benchmarks are usually selected so that they represent a good mixture of all possible programs. This leads to heuristics being designed in a one-size-fits-all manner with the goal on working *well* on the *average* program. Hand-crafting heuristics for every specific domain or architecture or even for an execution with specific input data does not scale in terms of development and maintenance effort.

Human errors, limitations in benchmark selection and the impracticality of hand-crafting heuristics for different domains have led to data-driven approaches for heuristics design. Machine learning has been used to automatically create (near-)optimal compilation decisions based on raw data, removing the need for hand-crafted, error prone heuristics [10; 96; 157]. The established workflow for supervised learning of compiler heuristics is shown in Figure 1.1. In supervised learning [37] a machine learning model is trained with data, where the desired prediction output is known. First, program *features*, such

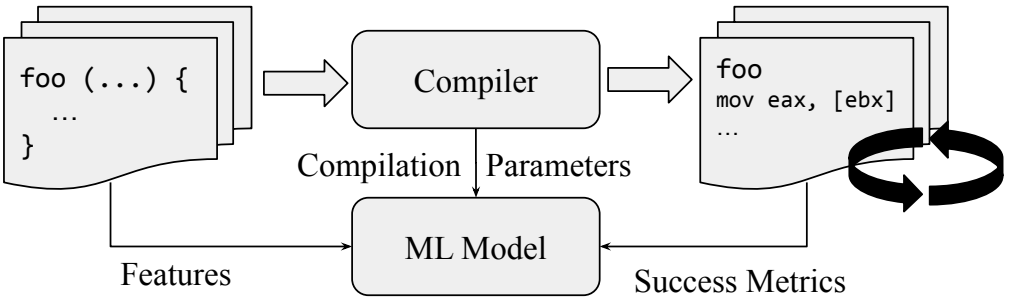


Figure 1.1: Abstract depiction of supervised learning for compilers.

as control flow characteristics or the number of instructions inside a loop, are extracted. These features will be the input to the machine learning model and describe the code to be optimized. After extracting the features, the sample program is compiled multiple times with different compilation parameters, such as loop unroll factors or vector lengths. The selected compilation parameter values are monitored as well. When executing the compiled programs, success metrics are measured, such as execution time or code size. In

that way, triplets of *program features*, *compilation parameters* and *success metrics* need to be extracted for many programs. For each program, the compilation parameter configuration which resulted in the compilation with the best performance is assigned as the *label* for the program's feature set. Finally, the features and the corresponding labels form the *training data*, which can be used to create a machine learning model. The trained model takes as input a set of program features and predicts as output the compilation parameters which optimize the success metric.

Obtaining training data can be a challenging task with respect to the success metric measurements. This is especially true for optimization parameters which are selected method-locally, such as loop unrolling factors, where the impact on particular loops needs to be measured in isolation. Hence, related work on machine learning in static compilers often learns global or method-specific compiler flags rather than method-local optimization decisions.

In dynamic compilers, obtaining training data is even more challenging. In order to identify the compilation parameter value which results in the best performance, multiple compilations of the same code need to be compared. These versions need to be compiled identically, apart from the compilation parameter whose impact needs to be measured. However, dynamic compilers produce less consistent results than static compilers, because they run in parallel to the executed program. If the compilation of one method takes longer than usual, due to memory or CPU load, more profiling information for not yet compiled methods can be gathered. Therefore, profile-based optimizations of future compilations can be performed differently. Ultimately, identical source programs can result in different machine code when compiled dynamically. This aggravates measuring the impact of compilation parameters consistently across multiple compilations. In addition, using machine learning during dynamic compilation directly impacts the total execution time of the program. Therefore, dynamic compilers need to be much more careful with the use of complex models.

This thesis contributes solutions to the difficulties of using machine learning in dynamic compilers to improve optimization heuristics and to assist compiler engineers in their work.

1.3 State-of-the-Art

Machine learning has been used in compiler research for decades [10; 96; 157], with projects, such as MilepostGCC [52; 53] or OpenTuner [8] having gained wide popularity.

To put our research in the context of previous work, we shortly discuss the state-of-the-art of techniques and approaches, used in static and in dynamic compilers. An in depth comparison to related work can be found in Chapter 9.

Iterative Compilation Iterative compilation [17] aims to find the optimal performance of programs by compiling them repeatedly with different optimization parameters. Genetic algorithms are typically used to efficiently explore the, often infeasibly large, state space which is spanned by the set of adjustable compiler flags [25; 150]. Iterative compilation, based on efficient (genetic) search algorithms, is still used for optimizing programs [119; 125]. However, it can also be considered as the origin for using machine learning in compilers [96]. Supervised approaches use iterative compilation to find the *best* set of parameters for a compilation, which is then used as input for training a machine learning model [10; 96; 157]. This way, the expensive task of compiling programs multiple times can be performed offline, whereas the trained model is then used as knowledge base to predict good configurations online.

Obtaining Training Data If explicit training data for supervised learning is required, current approaches often use custom (micro-)benchmarks. For compiler flag tuning [8; 53], iterative compilation and end-to-end performance measurements of whole programs can be used to obtain labelled data sets [96]. Method-local performance measurements are typically performed by instrumenting code regions with timestamps [140] or by extracting hardware performance counters [26; 124], e.g. with PAPI [153] or the HPCToolKit [3]. The more training data is provided during training, the better the model will perform [35]. However, obtaining large sets of training data is still considered a problem in current research, which often falls back to synthetically creating benchmark programs with genetic algorithms or deep learning [35; 155].

Machine Learning Algorithms Over the last years, with new developments in the areas of deep learning and neural networks [15; 141], these techniques have become a de facto standard across many domains [6], including compilers. Deep neural networks are universal function approximators, which can learn arbitrary input-output relationships if a sufficiently complex network architecture is chosen. Due to the prevalent use of graph-based intermediate representations in compilers, graph neural networks are explored in current compiler-related research [19]. Non-deep-learning-based models, such as support vector machines [32] or decision trees [20], are used less frequently but can provide accurate predictions for isolated tasks as well. Decision-tree-based models also have the advantage of being human-readable [143]. More recently, reinforcement learning [77] has

found its way into compiler construction [63; 104; 156]. This kind of learning algorithm makes no distinction between training and prediction phases. Feedback about the quality of the last prediction is immediately used to update the underlying model at run time.

Features Features are the input to a machine learning model [96; 151]. In the context of compiler optimizations, they describe the source code which is compiled or its intermediate representation, which is often graph-based. Frequently used features shared among related works are: number of memory or arithmetic instructions, number of branches [91; 143; 151]; graph features [19; 123], such as edge counts or node counts. Dynamic features, derived from profiling information, such as loop frequency or concrete data types, are used more rarely due to the focus on machine learning in static compilers. There are also approaches which use features that describe the *behavior* of code, extracted with performance counters [124], rather than its static *structure* [24]. The potentially available features are well explored. Feature pre-processing steps, such as correlation or importance analysis are often tied to the specific problem at hand and are not solved universally.

Optimizations Current research on machine learning in compilers often attempts to find holistic solutions across multiple optimizations. These approaches address phase ordering [91; 104], optimization selection [74; 124] or flag tuning [25; 143]. Sometimes, loops or loop nests are being optimized holistically, by taking supported loop optimizations into account and trading them off against each other [89; 105]. Heuristics for specific optimizations are considered far less frequently in research of the past few years. The more frequently addressed single optimization heuristics which are investigated with machine learning are: inlining [25; 143], loop unrolling [149] and vectorization [63; 108; 109]. We believe that an isolated view on single optimizations can help to boost the effectiveness of these optimizations.

ML Entry Barriers There are many research projects which show that machine learning models can outperform hand-crafted heuristics. However, we are only aware of a single state-of-the-art industry compiler which uses machine learning in production [154]. This approach, however, optimizes for code size rather than for performance. Current research addresses this scarcity by providing frameworks [8; 36; 154] for lowering the entry barrier for compiler engineers to get familiar with machine learning. They provide interfaces to common compiler optimizations and different learning algorithms to use. We believe that these efforts are worth pursuing, as they close the gap between machine learning research and compiler construction.

1.4 Remaining Challenges

Albeit being researched for a long time, machine learning is still not picked up by commercial production compilers. Some unsolved challenges only hamper its use in dynamic compilers, which we focus in this thesis, whereas other problems are inherent to the use of machine learning, regardless of static or dynamic compilation.

Black-box Deployment Machine learning is sometimes considered as wizardry, considering that it solves difficult problems in a way which is often not understandable by humans. Frameworks, such as scikit-learn [127], PyTorch [126], Keras [28] or Tensorflow [2], open the use of machine learning to a broader community of non-experts, by providing abstraction layers to hide the mathematics behind training and prediction. The deployment of models into larger systems is facilitated by libraries, such as ONNXRuntime [120], Tribuo [129], or DL4J [152]. This shifts the machine learning knowledge to the maintainers of the libraries and away from the immediate users of the frameworks, when designing an actual model architecture and a training pipeline. After deploying a machine learning model in a compiler, it might solve tasks better than compiler engineers, but it is often not evident why. If a model performs even worse on some programs, debugging the regression and finding its root cause in either training data, model architecture, hyperparameters or learning algorithm is challenging, especially for compiler engineers without in-depth ML knowledge. We believe that this is the main reason, why machine learning is not used in production compilers. Therefore, a substantial challenge is to retain maintainability and understandability when using machine learning in a compiler.

Training Data Creating a well-performing machine learning model depends on the availability and quality of training data. Obtaining enough training data for learning compiler-internal tasks is difficult. This is confirmed by related work, which addresses the lack of available training data [96] or research on how to generate artificial training data or benchmark programs [56; 155]. Creating training data requires to have a sufficient amount of (benchmark) programs to execute, which will produce accurate measurements of how isolated optimization decisions impact certain program regions. Thereby, optimizations can be performed on an arbitrary level of granularity, i.e. method-local. Related work in static compilation has found solutions and workarounds for instrumenting and measuring the performance of differently compiled program snippets. However, similar approaches in dynamically compiled programs pose additional challenges, as two compilations of the same program likely result in differences in the compilation process. We consider

obtaining the impact of method-local optimization decisions in a dynamic compiler as having been neglected so far and, thus, as an unsolved challenge.

User Behavior If two identical programs, functions or code snippets are used with vastly different inputs or environments, the compilation has to be adapted to achieve optimal performance. For example, existing approaches use the CPU architecture as a feature to base compilation decisions on the underlying hardware. However, there will always be implicit knowledge of the system which is not modelled explicitly as part of the features. Additionally, user behavior or provided input data can hardly be reflected in a static set of features. Related work tends to aim for models which perform well on general workloads [154], whereas optimizing for user specific environments or behavior is only addressed on rare occasions [119].

Automated Update Machine learning models are usually untouched after training and deployment, with few exceptions [151]. If models have good generalization properties, frequent updates might not be necessary [154]. For specialized models, however, changes in the domain need to be reflected in the model as well. There are approaches which use pre-trained models and refine them later with specialized data [63]. However, cyclic and automated update processes are not yet available, even with reinforcement learning [77] on the rise.

1.5 Contributions

1.5.1 Scientific Contributions

This thesis contributes to the state-of-the-art of using machine learning in dynamic compilers for improving optimization heuristics. The scientific contributions in chronological order are:

Heuristics Design and Validation We show how machine learning can be used to assist compiler engineers in improving existing hand-crafted heuristics. In [110], we proposed a pipeline where compiler engineers use machine learning during the heuristics design process. An implementation of this pipeline in the GraalVM compiler [170], which used a so-called *assistance mode* [116], led to improvements in existing hand-crafted heuristics for estimating code size impacts of optimizations. In a similar way, we found performance

bugs in vectorization heuristics after deploying our learned models [111]. In addition, we also outlined how the use of human-readable, decision-tree-based models can help in the initial heuristics design process, similar to [143].

Compilation Forking Related work [35; 155] mentions the lack of available training data, which hinders solving compiler-related tasks with machine learning. We introduce *compilation forking* [113], a novel technique which enables large-scale performance data generation for method-local optimization parameters in a dynamic compiler. We present how *compilation forking* helps to overcome obstacles introduced by dynamic compilation which were either not solved in related work or overlooked. *Compilation forking* has been successfully used for obtaining training data in multiple of our published work [111; 112; 113].

Self-optimizing Compiler Heuristics We present an approach [112] where new models or previously deployed pre-trained models are automatically tailored to currently executed programs. Our approach shows that compilation forking, together with deoptimization allows this tedious task to happen at user site without requiring control over the environment or having particular benchmark programs. These self-optimizing heuristics achieve significant speedups compared to heuristics which are designed for global use. The ideas behind self-optimizing heuristics and compilation forking have also been patented [115] together with Oracle Labs.

Learned Optimization Heuristics We present multiple case studies where we replaced existing compiler heuristics with learned models. In [116] we showed how the code size impact of optimizations can be estimated with the help of neural networks. Subsequent models aim towards improving the performance of loop-related optimizations. For our research with compilation forking [113], we created machine learning models for loop peeling and loop unrolling. Our models achieve similar performance as the highly optimized GraalVM compiler when replacing the hand-crafted heuristics. In a subsequent work [112], we used deep learning to obtain highly-specialized models for loop peeling in specific benchmarks. These highly specialized models outperformed the GraalVM compiler significantly on several benchmarks. In our final study [111], we learned the optimal unroll factor to be used for vectorized loops. We analyzed the impact of different sets of training data, as well as different feature and model configurations. The resulting models achieve significant speedups over current static heuristics.

1.5.2 Technical Contributions

Some of the contributions presented in this thesis have been integrated and are shipped as part of the GraalVM.

Timestamp Instrumentation Developed as part of compilation forking, the IR graph instrumentation for measuring the aggregated execution time of methods within program executions has been merged into the GraalVM and helps engineers by providing fine-grained performance measurements.

Compilation Forking The approach as a whole will be integrated into GraalVM's benchmarking infrastructure as part of future work.

Existing Heuristics Using machine learning in an assistive way, we were able to improve hand-crafted heuristics in the GraalVM. This includes updating the weights of nodes in the deployed node cost model [99] and fixing performance bugs in GraalVM's duplication and vectorization algorithms.

1.5.3 Publications

The core contributions of this thesis, addressing novel solutions for optimizing heuristics in dynamic compilers, are summarized in five peer-reviewed publications. The unaltered author versions of these publications are found in Part II. These papers are:

[110] Machine learning to ease understanding of data driven compiler optimizations. **Raphael Mosaner**. In *Companion Proceedings of the 2020 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity* (SPLASH Companion 2020). pp. 4–6. DOI: 10.1145/3426430.3429451

[116] Using machine learning to predict the code size impact of duplication heuristics in a dynamic compiler. **Raphael Mosaner**, David Leopoldseder, Lukas Stadler, and Hanspeter Mössenböck. In *Proceedings of the 18th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes* (MPLR 2021), pp. 127–135. DOI: 10.1145/3475738.3480943

[113] Compilation Forking: A Fast and Flexible Way of Generating Data for Compiler-Internal Machine Learning Tasks. **Raphael Mosaner**, David Leopoldseder, Wolfgang Kisling, Lukas Stadler, and Hanspeter Mössenböck. In *The Art, Science, and Engineering of Programming*, 2023, Vol. 7, Issue 1, Article 3, pp. 1-29. DOI: 10.22152/programming-journal.org/2023/7/3

[112] Machine-Learning-Based Self-Optimizing Compiler Heuristics. **Raphael Mosaner**, David Leopoldseder, Wolfgang Kisling, Lukas Stadler, and Hanspeter Mössenböck. In *Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes* (MPLR 2022), pp. 98–111. DOI: 10.1145/3546918.3546921

[111] Improving Vectorization Heuristics in a Dynamic Compiler with Machine Learning Models. **Raphael Mosaner**, Gergö Barany, David Leopoldseder, and Hanspeter Mössenböck. In *Proceedings of the 14th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages* (VMIL 2022), pp. 36-47. DOI: 10.1145/3563838.3567679

One additional, peer-reviewed publication has been made outside the scope of this thesis and is therefore not included in Part II:

[114] Supporting On-stack Replacement in Unstructured Languages by Loop Reconstruction and Extraction. **Raphael Mosaner**, David Leopoldseder, Manuel Rigger, Roland Schatz, and Hanspeter Mössenböck. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes* (MPLR 2019), pp. 1-13. DOI: 10.1145/3357390.3361030
In this study, we showed how on-stack replacement can be utilized in bytecode-interpreter-like languages. We did this by reconstructing loops and encapsulating them in separate execution units which can be compiled on their own when becoming hot. An implementation in the GraalVM compiler showed significant reductions in program warmup time.

Apart from the publications in scientific journals and conference proceedings, the idea of compilation forking and its application to optimize heuristics at run-time has resulted in a patent:

[115] Online Machine Learning Based Compilation.
Raphael Mosaner, David Leopoldseder and Lukas Stadler.
U.S. Patent Number 11.392.356 filed February 26th, 2021, Issued July 19th, 2022

1.6 Limitations

The contributions of this thesis provide advancements to how machine learning can be used in dynamic compilers. Nevertheless, the presented approaches still have limitations, which are either inherent or have to be addressed as part of future work.

Measurement Noise The use of the processor time-stamp counter (*RDTSCP*-command) for performing execution time measurements is accepted across related work [93; 140]. However, related work often uses coarse-grained measurements, for example only at the start and at the end of long-running methods. Our performance measurements are more fine-grained and monitor the impact of method-local optimization decisions. Following the uncertainty principle, measurement noise grows with more fine-grained measurements. Controlling the measurement noise has been a challenging task throughout this thesis, which we still consider not fully solved. In Chapter 6 and Chapter 7, we described our pre-processing steps to remove data points for which it cannot be guaranteed that noise has no impact when calculating the label. We are certain to have advanced the state-of-the-art when it comes to obtaining accurate measurements for arbitrary code. Nevertheless, measuring optimizations with minor impacts on the surrounding function is still limited by the level of measurement noise.

Environment We deliberately designed our approaches [112; 113] without the need for a fully-controlled environment. However, context switches, for example, can cause huge outliers for measured execution times. We had to take additional measures to either detect large outliers on-the-fly or exclude data points with large variance from our training data.

Scalability Compilation forking [113] creates multiple versions of a function, which are recombined after individual compilation. This poses limitations in terms of scalability. The compile time until fork recombination increases linearly with the number of versions. Tasks after recombination (register allocation, code generation, etc.) can scale worse if their run-time complexity scales polynomially with the size of the graph, for example. The code size increases during compilation forking can also be a limiting factor if too many versions are explored. As discussed in [113], compilation forking should therefore not be used to exhaustively explore large state spaces.

Model Generalization Generalization and overfitting can be crucial factors when training machine learning models. Designing models which perform well on heterogeneous sets of data is a difficult task, as experiments in [113] show. The more homogeneous the data is, e.g. benchmarks from the same suite [113], the better the trained models perform on similar data. Our study on self-optimizing heuristics [112] has shown that highly overfitted models are able to achieve the best results. Leaning either towards worse-performing, general models or leaning towards highly overfitted well-performing models is a design decision which we consider as a limiting factor of not only our approach but of machine learning in general.

Dynamic ML Overhead Invoking large machine learning models during dynamic compilation has an immediate impact on the program warmup and total execution time. This is a limitation which cannot be circumvented. In [111], we showed that model loading can be a very expensive task. Therefore, short-running programs where few model decisions are required are advised to use pruned models. If models need to be invoked frequently during dynamic compilation, the compile time overhead may become significant. However, optimizing the usage of machine learning models was not the focus of this thesis.

1.7 Project Context

The research conducted in this thesis has been made—and was funded—in the context of the GraalVM [170] project, which originated from a long-standing research collaboration between Oracle Labs[1] (formerly Sun Microsystems) and the Institute for System Software[2] (SSW) at the Johannes Kepler University Linz. This research collaboration dates back to a sabbatical of Prof. Hanspeter Mössenböck, the head of the SSW, at Sun Microsystems in 2000 where he improved the intermediate representation in HotSpot's C1 compiler [84] and implemented a graph-coloring register allocator [117]. Subsequent research introduced new algorithms and concepts into the HotSpot VM:

- Mössenböck and Pfeiffer [118] proposed an algorithm for linear scan register allocation for HotSpot's C1 compiler, which was more performant than the previously implemented graph-coloring register allocator [117]. The algorithm was further refined by Wimmer and Mössenböck [160] who implemented multiple optimizations, which improved the quality and the performance of compiled code.

[1] `https://labs.oracle.com`
[2] `https://ssw.jku.at`

- Kotzmann and Mössenböck [82] implemented escape analysis in HotSpot's C1 compiler and added means for tracking optimized objects and methods to safely handle deoptimization [83].
- Wimmer and Mössenböck [161] implemented object colocation in the HotSpot VM, which groups objects on the heap according to access patterns. Based on this work, they added automated object inlining [162; 164] and array inlining [163] to HotSpot.
- Würthinger et al. [166; 167; 168; 169] modified HotSpot to support dynamic updates of running programs via class re-definition.
- Häubl and Mössenböck [64] researched trace compilation, inlining of traces [65] and the transition between interpreted and compiled traces caused for example by exceptions [66].

Since the successful introduction of the GraalVM [170], several PhD students were involved in the collaboration with Oracle Labs where they conducted research in the context of the GraalVM compiler and the Truffle framework [165]:

- Stadler et al. improved the compiler start-up time by introducing caching of IR graphs and priority queuing of compilation tasks [145]. They analyzed the performance impact of compiler optimizations, which were designed for Java programs, on Scala benchmarks [146]. Furthermore, they proposed *partial* escape analysis, which enables optimizing non-escaping objects for individual branches [147].
- Duboscq et al. introduced an intermediate representation for the GraalVM compiler, the Graal IR [45; 46], which is based on the *sea-of-nodes* [29] concept in HotSpot's C2 compiler, but provides additional features for supporting speculative optimizations and deoptimization.
- Grimmer et al. [58; 59] implemented TruffleC, which enables execution of C programs in the GraalVM by using the Truffle framework [165].
- Grimmer et al. [60] introduced language interoperability in the Truffle framework, which enabled efficient access to data structures or methods from other Truffle-based languages. They especially achieved large speedups when using C extensions in other languages [61].
- Rigger et al. [134] extended on TruffleC [58] to build a Truffle-based LLVM interpreter, called Sulong [131; 133]. Based on Sulong, they addressed undefined behavior in C [135], enabled safe execution of unsafe programs on the JVM [132; 134] and detected bugs in C programs [136; 137].

- Daloze et al. [39] researched guest language safepoints and their usage in a Truffle-based implementation of Ruby. Their work extended into thread-safety and parallelization for dynamically-typed languages [38; 40].
- Eisl et al. [47] proposed trace register allocation, where register allocation is performed for small sections of a method, and compared it to GraalVM's holistic linear scan register allocator. They investigated trade-offs between compile time and performance of compiled code, by using different allocation strategies for individual traces [48] and proposed a theoretical model for parallelizing trace register allocation.
- Leopoldseder et al. [97; 100] researched how code duplication can enable compiler optimizations. They simulated the optimization opportunities after a duplication [100] and used a cost model [99] to trade-off code size increase and expected performance gain. Based on their algorithm, they improved loop unrolling in the GraalVM compiler [98].
- Kreindl et al. [85; 86] proposed *TruffleTaint*, a platform, which enables taint propagation across language boundaries [86]. *TruffleTaint* can flexibly select properties for taint labels and their propagation [88] and has relatively low run-time overhead, due to GraalVM's speculative optimizations [87].
- Kloibhofer and Makor [78; 102] currently research how dynamically analyzing processed data can be used to select efficient compilation and execution strategies, such as transforming arrays to columnar storage for faster access [79; 103].

1.8 Outline

This thesis is structured into three parts. The first part, continues with providing background information to concepts related to the GraalVM and machine learning techniques in Chapter 2 and summarizes and connects the scientific publications of this thesis in Chapter 3. The second part presents all papers, which were published in the course of this thesis in Chapters 4 to 8. The author versions are unaltered to the initial publications and are self-contained in terms of the bibliography. The third part finally discusses related work in Chapter 9, gives an outlook to future work in Chapter 10, and recapitulates the key insights of our research in Chapter 11.

Chapter 2

Background

This chapter spans a bridge to the scientific publications in Part II, which due to the required brevity of conference papers often fall short of discussing background concepts in a sufficiently detailed manner. Thus, this chapter introduces important concepts of the GraalVM [170] in the context of which the research of this thesis has been conducted. Furthermore, it provides a terminology of machine-learning-related concepts and gives an overview of the types of models which were employed in Part II.

2.1 GraalVM

The approaches presented in this thesis, along with case studies and experimental evaluations have been implemented and performed in the GraalVM. The GraalVM [170], depicted in Figure 2.1, is a Java virtual machine (VM) [75] based on the HotSpot VM [71], which uses the GraalVM compiler [170] for aggressive optimizations and the Truffle framework [170; 171] for polyglot execution of programs that cannot be compiled to Java bytecode. The GraalVM also provides ahead-of-time (AOT) compilation to native

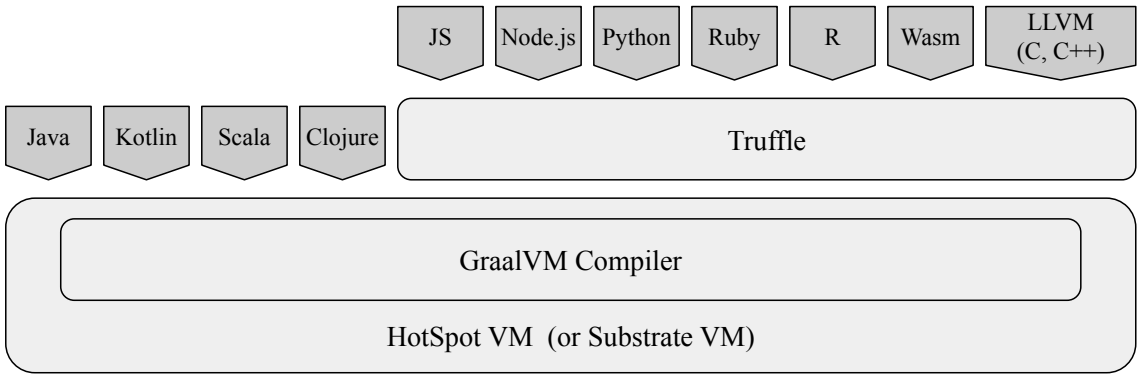


Figure 2.1: GraalVM architecture.

platform executables. In this *native image* mode, the GraalVM compiler is deployed in the SubstrateVM rather than in the HotSpot VM. However, this thesis uses the GraalVM compiler only in the context of the HotSpot VM.

2.1.1 HotSpot VM

The HotSpot VM [71] is the most widely used Java virtual machine and is part of the OpenJDK [121], the official reference implementation of Java [73] since version 7. Dynamic compilation with multiple optimization levels (tiers) allow for, both, aggressive, profiling-based optimizations and relatively fast program start-up times.

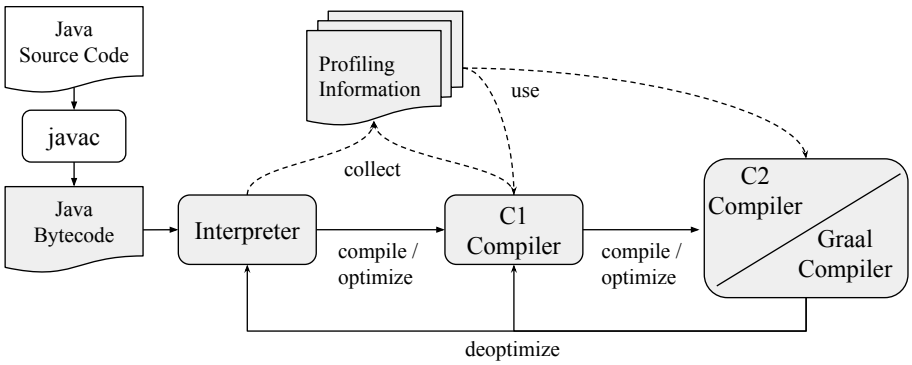


Figure 2.2: HotSpot's tiered compilation.

Figure 2.2 summarizes HotSpot's tiered compilation. A Java program, compiled to Java bytecode, is initially executed by HotSpot's template-based interpreter, which sequentially maps every bytecode to assembly code snippets of the target machine [71]. Program execution during interpretation is slow, but compiling to bytecode is much faster than compiling to optimized machine code, which only pays off for small portions of the program that account for most of the execution time. The interpreter collects profiling information, which serves two tasks: identifying what parts of a program to compile and how to compile them. Method invocation counts and loop iteration counts—monitored by counting the number of executed backward jumps—are used to identify *hot*, i.e., frequently executed, code parts; thus the name *Hot*Spot. If a method has been executed frequently enough, HotSpot triggers the dynamic compilation of this method, and subsequent invocations use the compiled and optimized version of the method. High loop iteration counts can also lead to a method becoming hot, which accounts for rarely executed, but long-running methods due to frequently executed loops. In such cases, on-stack replacement (OSR) [41] can be used for switching to a compiled version between loop iterations while the method is executed. When compiling a method, profiling information is used to identify beneficial optimizations or to apply speculative optimizations. For example, loops can be optimized

differently depending on their profiled iteration counts, or virtual calls can be removed if the profiling information for this call site only encounters a single receiver type. If profiling-based assumptions later turn out to be wrong, HotSpot uses deoptimization [70] to switch back to an interpreted or less optimized version of the method, often followed by a re-compilation with adapted assumptions.

HotSpot uses multiple compilers and configurations, which allows a good trade-off between compile time and optimization level. The C1 compiler [84] allows for fast but less optimized compilations. It is used if methods are becoming hot and need to be sped up compared to their execution in the interpreter. For simple methods, which do not have much optimization potential, the C1-compiled version is often sufficiently fast. However, C1 continues to collect profiling information and can trigger the more expensive but highly optimizing compilation via the C2 compiler [122] or the GraalVM compiler [170]. Figure 2.3 summarizes the performance of a method during HotSpot's tiered compilation.

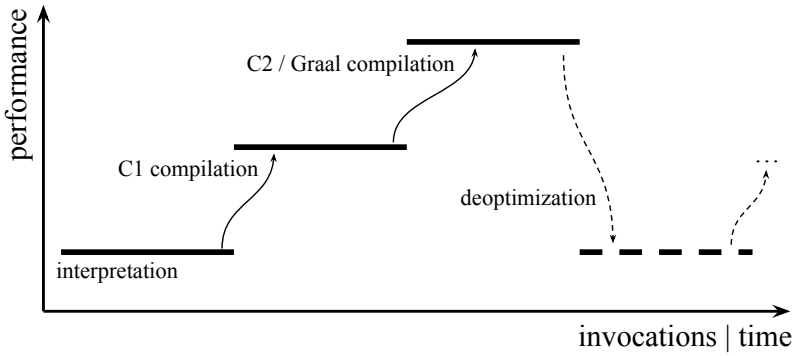


Figure 2.3: Performance of a method during tiered compilation.

At some point, programs usually achieve a steady state of peak performance, if no further compilations or deoptimizations need to be performed. This marks the end of the warmup phase.

2.1.2 GraalVM Compiler

The GraalVM compiler [170] is a highly-optimizing dynamic compiler written in Java. It replaces the C2 compiler [122] in the HotSpot-based GraalVM by using the JVM compiler interface (JVMCI) [76]. At the point of writing this thesis, the GraalVM compiler can apply over 60 optimizations across multiple abstraction layers, partitioned into frontend and backend. An overview of the internal structure of the GraalVM compiler can be seen in Figure 2.4.

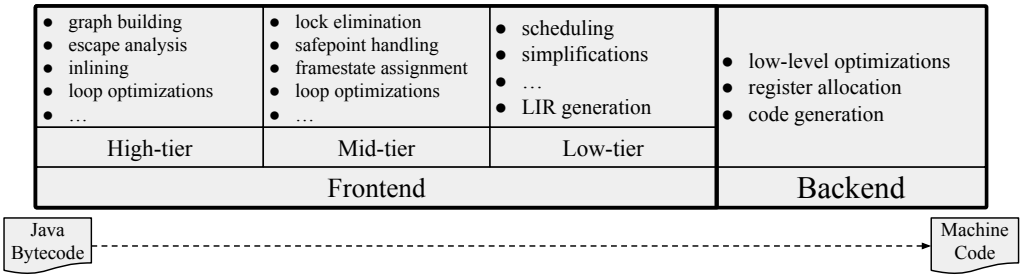


Figure 2.4: Overview of the GraalVM compiler.

Initially, the compiled Java bytecode is turned into a graph-based intermediate representation, the Graal IR [45; 46], which facilitates the optimization process. The graph building followed by the high-tier, the mid-tier and the low-tier form the compiler frontend. Each of these frontend tiers encapsulates multiple optimization phases which are performed on the IR. At the end of each tier, the IR is de-sugared and transformed into more low-level instructions. This process is called lowering. In the high-tier, IR nodes correspond closely to the Java bytecode. The compiler applies optimizations such as inlining, partial escape analysis, duplication or loop optimizations. After lowering the IR to mid-tier, Java bytecode-like instructions are replaced with architecture-agnostic machine-level instructions. For example, address-based memory semantics are introduced instead of Java object accesses. Optimizations in the mid-tier contain lock elimination, safepoint [139] handling for loops and partial loop unrolling [98]. The low-tier introduces architecture-specific instructions, e.g. memory addressing specific to amd64 [7] or aarch64 [1]. The main purpose of the low-tier is to prepare the IR for LIR generation, which is the intermediate representation used in the backend. This includes a graph scheduling algorithm for creating the basic block structure for the control-flow-graph-like LIR. The compiler backend contains some low-level optimizations, the register allocation and the code generation.

All optimizations which are addressed in this thesis are part of the compiler frontend. Loop peeling is part of the high-tier, partial loop unrolling is found in the mid-tier and vectorization is applied in the low-tier, because it is architecture-specific. The instrumentation for extracting timestamps is added after all major optimizations as the last phase of the low-tier.

2.1.3 Graal IR

The GraalVM compiler uses a graph-based intermediate representation in its frontend, which is called Graal IR [45; 46]. It is based on the sea-of-nodes concept [29], which models both control flow and data flow and was first implemented for the C2 compiler.

The sea-of-nodes IR has been developed as an enhancement of the static single assignment (SSA) form [138]. Code in SSA form has each variable assigned only at a single location which simplifies usage-definition analysis and benefits many optimizations. Programs can be transformed into SSA form by introducing artificial variables for each assignment and so-called ϕ-instructions at control flow merges to combine the artificial variables from the incoming branches. GCC [55] or LLVM [92] use SSA form on top of control flow graphs (CFG), where each graph node represents a basic block, which encapsulates instructions in a fixed order without incoming or outgoing jumps. The sea-of-nodes concept implements SSA form but relaxes the strict order of instructions in basic blocks which allows for a more flexible placement of instructions.

Graal IR represents control flow and data flow in a single graph with different edge types, as shown in Figure 2.5. Control flow edges (red), pointing downwards, impose a fixed

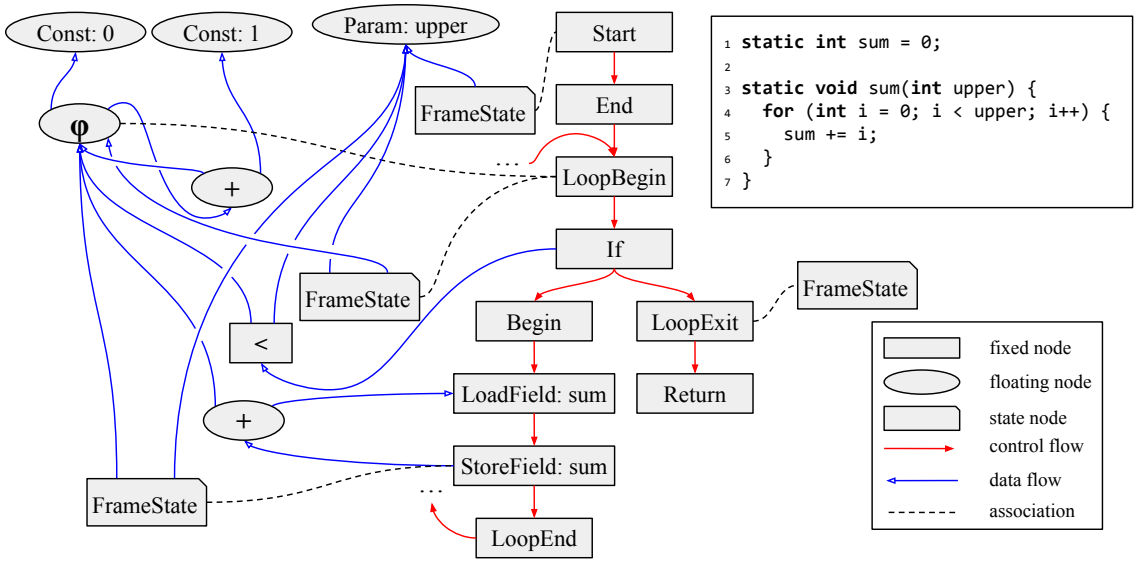


Figure 2.5: Graal IR example.

order on the connected nodes. These nodes are called fixed nodes and model branches, loops or operations with side effects, such as memory reads or writes. Data flow edges (blue) point upwards from data usages to definitions, represented by nodes which can be floating. This means, that they do not have strict requirements regarding their position in the graph, as long as each value is defined before it is used. The actual position where a floating node is executed is fixed as late as possible in a process called scheduling. A schedule of the whole graph needs to be created for the transition to the backend's CFG-based LIR. As Graal IR is in SSA form, ϕ nodes are attached to control flow merges to pick a value depending on the executed branch. In Figure 2.5, the value of the ϕ node representing `i` is either `0` if the loop is entered via the pre-header or `i+1` if entered via

the back edge. Finally, *FrameState* nodes contain all information which is required by the interpreter to continue execution after deoptimization. They are attached to side-effecting nodes or control flow positions where deoptimization is supported.

In this thesis, we performed all instrumentation of the code directly on its Graal IR representation, using the appropriate nodes on the respective level of abstraction.

2.1.4 Truffle

The Truffle [170; 171] language implementation framework is part of the GraalVM and can be used for implementing abstract syntax tree (AST) interpreters for arbitrary languages. Truffle enables execution of programs, which are not available as Java bytecode, on GraalVM, so that they profit from all optimizations which are provided by the GraalVM compiler. In addition, Truffle interpreters are self-optimizing using partial evaluation based on the first Futamura projection [54]: The interpreter is specialized to a specific input program, and AST operations are inlined until the tree collapses to a single node. Due to the GraalVM compiler, Truffle implementations of languages such as JavaScript or Python achieve performance close to native execution.

While Truffle is not explicitly addressed in this thesis, it enabled evaluating our approach for non-Java workloads. In Chapter 7 we encountered significantly larger speedups for JavaScript benchmarks, which indicated that hand-crafted heuristics in the GraalVM compiler were not optimized for all languages equally well.

2.2 Machine Learning

In this section, we briefly discuss machine-learning-related terminology and concepts which were used in this thesis.

2.2.1 Terminology

Feature A feature is a measurable property of an object of interest. A feature vector contains all features which describe the object of interest. In the context of this thesis, features describe properties of code or its behavior when executed. Examples of features are the number of `AddNodes` in a code's Graal IR representation or the profiled frequency of a loop.

Target A target is a property of an object of interest whose value should be predicted by a machine learning model. Examples of targets in this thesis are the code size of a method after compilation or the unroll factor which maximizes the performance of a loop.

Label A label is the "true" value of a target property. In this thesis, labels are often identified via success metric measurements. For example, when measuring the performance of a loop with different unroll factors, the factor that provides to highest speedup becomes the label for the target "unroll factor".

Data Point Data points or samples consist of the measured feature vectors for particular objects and their corresponding target label(s). The total of all data points denote the data set which is used for training and testing a model.

Training / Learning Training or learning is the process of using a machine learning algorithm to build a machine learning model based on the provided training data. Supervised learning [37], which is used in this thesis, uses labelled feature vectors to establish a mapping from features to targets during training. Unsupervised learning, in contrast, does not require labelled data and can, for example, be used to identify patterns in feature vectors.

Prediction / Inference Prediction or inference is the process of providing a tuple of feature values—a feature vector—to a machine learning model which produces one or multiple target values as output. These outputs are called predictions.

Classification - Regression Classification models predict a result from a set of defined classes, whereas regression models predict arbitrary numeric values. Predicting the best loop unroll factor $\in 1, 2, 4, 8, 16$ would be a classification task, whereas predicting the size of the machine code after compilation would be a regression task.

Cross-validation Cross-validation [81] is a technique to determine the prediction accuracy of a machine learning model and its capability to generalize to unseen data. The data set is split into multiple subsets. Then a model is trained on the union of all but one of these subsets and the remaining subset is used for testing the model, by comparing its labels to the model's predictions. This process is repeated for all combinations of training and test data and the model is rated with the average prediction accuracy.

Overfitting A model is overfitted [144] to the training data, if it has perfectly learned all its peculiarities, including noise. As a result, the model works very well on the known data, but lacks the ability to generalize to new data, where it performs poorly.

2.2.2 Machine Learning Models

Over the past decades, numerous machine learning algorithms have been developed, each with its own slight variations and hyperparameters to configure the architecture and the training of the model. In the research conducted in this thesis, we focused on neural networks [15; 141] and decision trees [20; 68], which are briefly summarized in this section.

Neural Networks Artificial neural networks (ANN) [6; 141] are a widely used class of machine learning models which imitate human or animal brains. Figure 2.6 depicts the general structure of ANNs. They consist of an input and an output layer as well

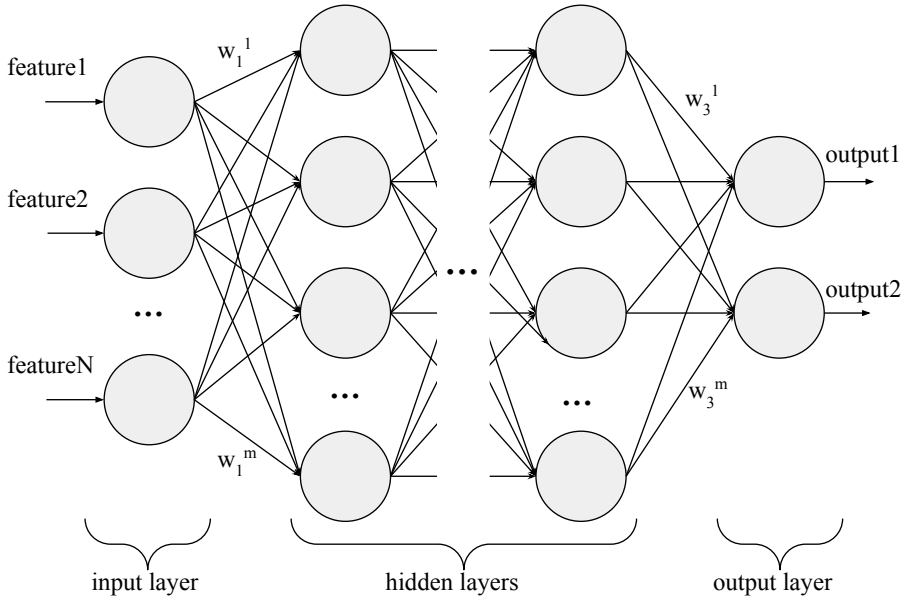


Figure 2.6: Neural network architecture.

as so-called hidden layers in-between; multiple hidden layers led to the notion of deep neural networks (DNN) or deep learning [6; 141] . Layers consist of neurons—named after their counterparts in the human brain—which are connected with other neurons unidirectionally. Each of these connections has a weight, which is determined during training. When data is flowing through the network, the value (called activation) of each neuron is calculated by the weighted sum of all input values. The activation is then fed into an activation function, which introduces non-linearity, and its result is the output of the neuron which is forwarded to the next layer. The model architecture (i.e. the number and types of layers and how neurons are connected) as well as the use of batch normalization [72] or dropout [144] layers affect the training process. Apart from the

model architecture, two additional hyperparameters affecting the training process are: the *batch size*, which defines how many samples need to be processed before model parameters are updated, and the *learning rate*, which defines how much the weights are adapted for each processed batch of training data. Hyperparameters often have to be determined empirically.

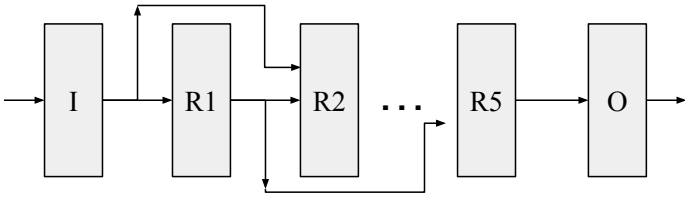


Figure 2.7: Residual neural network with skip connections.

Residual neural networks [67], as shown in Figure 2.7 where each box corresponds to one layer of neurons, are another shape of neural networks, which we used in this thesis. They use skip connections to speed up the learning process of deep networks and to enable training with less data, because the training initially focuses on the smaller "sub-networks" which emerge from skipping other layers [67].

Random Forests While neural networks often make decisions which are not comprehensible by humans, random forests [68] are human-readable. A random forest model consists of multiple decision trees [20], each being trained on a randomly selected subset of the data. The final output of a random forest is either the average or a majority vote over the outputs of its decision trees. Figure 2.8 depicts a snippet of a decision tree for a classification problem to identify whether an object belongs to either class *A* or to class *B* based on three features. During training, the algorithm identifies conditions on the features, to separate the training data according to the class labels. During prediction, a decision tree starts at its root node and evaluates the conditions based on the object's fea-

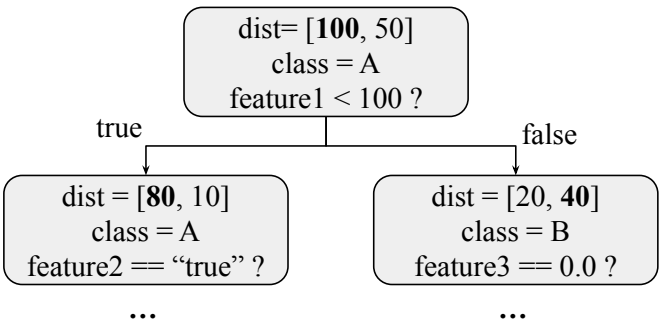


Figure 2.8: Snippet of a decision tree.

tures until a leaf node is reached. Then it assigns the class, which has been attached to the leaf node during training, to the object. Each node in Figure 2.8 contains the distribution of the training data in the first line, the most frequent class in the training data in the second line and the next split condition. For example, in the tree's root node, 100 data points from the training data belong to class A and 50 to class B, making A the dominant class for this node. Therefore, if the tree had no further layers, every data point would be predicted to be of class A, which would result in an accuracy of 66.67% for the training data. However, the condition $feature1 < 100$ separates the training data further, resulting in the nodes of the second layer. Objects, with $feature1 < 100$ would be classified as A and objects with $feature1 >= 100$ would be classified as B. The bottom right node shows that 20 A objects from the training data would be falsely classified as B, because B is the most frequent class in the distribution of the node. For the training data, 120 of all 150 data points (80%) would be classified correctly based on the first split condition. Additional split conditions, such as $feature2 == "true"$ or $feature3 == 0.0$, would separate the training data further. The maximum depth of decision trees is a hyperparameter which can be specified. When increased, more and more conditions are used to eventually separate the training data perfectly. However, this facilitates overfitting and degrades the overall model performance on unknown data, as discussed in Chapter 8.

Chapter 3

Overview

This chapter presents an overview of the research topics which are presented in this thesis and how they are connected. Each section summarizes one of the research papers which are presented in Chapter 4 to Chapter 8.

3.1 Machine Learning to Assist Compiler Engineers

Machine learning has been used in research compilers for decades [10; 96; 157], with sophisticated projects such as MilestoneGCC [52; 53] getting popularity nearly 15 years ago. Besides rapid developments in new machine learning techniques and compiler research picking up on the latest learning algorithms [19; 63; 156], industry compilers still refrain from using machine learning. In our paper *Machine Learning to Ease Understanding of Data Driven Compiler Optimizations* [110], we identified the degradation of maintainability and understandability as the major obstacles for deploying machine learning black boxes in compilers. Related work uses an all-or-nothing approach when it comes to machine learning in compilers: Compiler heuristics are either hand-crafted and understood by experts, or they are automatically derived from data by using machine learning algorithms, but then hardly maintainable.

We propose a different approach [110], where we envision machine learning as a tool for compiler engineers to improve existing heuristics while retaining maintainability. Figure 3.1 depicts our feedback-driven approach, where machine learning does not seize control of major decisions. Our initial vision shows that machine learning can either *assist* the compiler, by taking over sub-tasks where no suitable hand-crafted heuristics are available or it can guide compiler engineers when improving existing heuristics. This

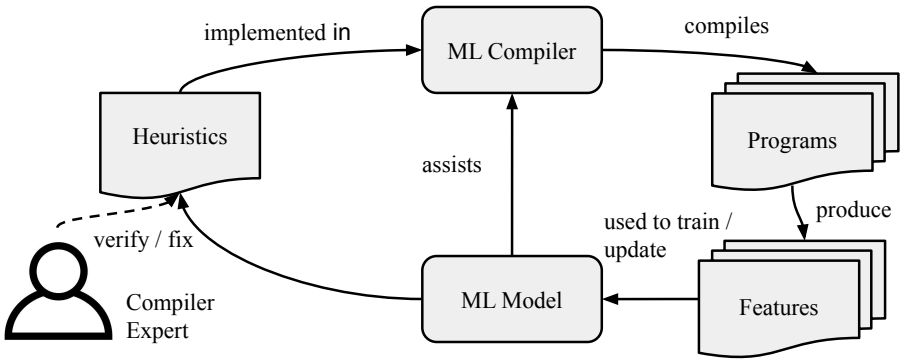


Figure 3.1: Machine learning in compilers - feedback cycle [110].

works by comparing outputs from the machine learning model to decisions based on the hand-crafted heuristics. Both important use cases are embedded in a loop to indicate the importance of re-evaluating or updating heuristics after new data has arrived. This can either be an immediate adaptation to the external environment if new programs are compiled or if the same programs are used in a different way, which can be inferred from the profiling information. It can also be an adaptation to internal changes caused by ongoing compiler development, such as updating optimization logic or adding new optimization passes. The larger feedback loop, involving compiler engineers, is coupled to the compiler development process to verify that heuristics fit the changed system. In case of performance bugs in unknown programs, machine learning models can automatically notify engineers of overlooked patterns in their heuristics.

3.2 Predicting the Code Size Impact of Duplication

Our follow-up research [116] implements the vision of *assistive* machine learning in the context of code duplication. Duplication is an *enabling optimization*, which, on its own, has no performance benefits but relies on subsequent optimizations which can only be applied after code segments have been duplicated. Listing 3.1 to Listing 3.3 [116] depict this optimization process. The `return` statement in Listing 3.1 is duplicated into both predecessor branches, which results in the larger code in Listing 3.2. However, due to the knowledge about `x`, additional optimizations can be applied, resulting in the more performant and smaller code in Listing 3.3.

```
1 if (x > 0) {
2     phi = x;
3
4 } else {
5     phi = 0;
6
7 }
8 return phi + 2;
```

Listing 3.1: Before duplication.

```
1 if (x > 0) {
2     phi = x;
3     return phi + 2;
4 } else {
5     phi = 0;
6     return phi + 2;
7 }
8
```

Listing 3.2: After duplication.

```
1 if (x > 0) {
2
3     return x + 2;
4 } else {
5
6     return 2;
7 }
8
```

Listing 3.3: After optimization.

Duplicating all code from control flow merges would result in the highest optimization potential [100], but on many occasions just leads to code size increases without performance gains. The GraalVM compiler uses an elaborate heuristic [99; 100] for making a trade-off between the expected gain, i.e. speedup, of a duplication and the expected cost in terms of the code size increase. Replacing this heuristic with a learned model would result in a hard to verify black box. In accordance with our previous work [110], we replaced only a minor sub-task with a machine learning model. The goal was to learn the code size impact of duplications in order to validate the underlying estimation heuristic [99] in the GraalVM compiler. This heuristic builds on a cost model, which assigns each node type in the GraalIR [45; 46] an abstract static size [99]. These abstract sizes were carefully chosen by hand.

Machine learning has been used in the past to estimate the size of compiled programs [30; 31; 154]. However, duplication in the GraalVM compiler is embedded between many other compiler phases. Therefore, while duplication is applied to regions of the IR graph, the duplicated nodes are likely to be changed by subsequent optimization passes. This rendered intuitive linear regression models [158], which we used in initial experiments, insufficient. Experiments with deep neural networks [15; 141], which are capable of capturing the non-linearities, reduced the number of incorrect predictions. We implemented an assistance mode where compiler engineers are informed about different duplication decisions depending on the underlying code size estimator—either heuristic or ML model. This assistance mode has been used by compiler engineers to unveil misconfigurations in the hand-crafted heuristics for estimating the code size impact.

We evaluated or approach on five established benchmark suites: *DaCapo* [16], *Scala-DaCapo* [142], *Renaissance* [130], *Octane* [27], *JetStream* [128]. Our evaluation includes three configurations: 1) the default GraalVM compiler with the hand-crafted heuristics for estimating code size impacts, 2) the GraalVM compiler where machine learning models replace these hand-crafted heuristics and 3) the GraalVM compiler with improvements

in the hand-crafted heuristics which were triggered by our machine-learning-based assistance mode. Speedups on some benchmark suites but slowdowns on others indicate that one-size-fits-all heuristics do not perform equally well on all programs. Detailed benchmark results can be found in the corresponding paper in Chapter 5.

The training data for this research was easily obtainable, as acquiring the code size of compiled programs is hardly susceptible to measurement noise. Learning optimal duplication heuristics with respect to the performance impacts of method-local duplication decisions would be more difficult. Creating models based on performance measurements in a dynamic compiler poses challenges regarding consistency and measurement noise which are not yet resolved in related work.

3.3 Compilation Forking

The core contribution of this thesis is the concept of *compilation forking* [113], which solves the problem of obtaining performance measurements for method-local optimizations in a dynamic compiler. *Compilation forking* enhances the benchmarking capabilities of a dynamic compiler and can be used for arbitrary user programs during conventional execution. We used *compilation forking* to generate the training data for our performance-based machine learning models.

In static compilers, programs can be compiled multiple times with different compiler flags to obtain the most performant flag combination. This process is called *iterative compilation* [17] or auto-tuning [10] and optimizes the program as a whole. For example, iterative compilation can identify the best global loop unroll factor for a program. However, this approach is not suitable for identifying the best loop unroll factor for each function, or even loop, individually. Related work has addressed this by re-compiling only functions (or parts of functions) and adding instrumentation for measuring the function's run time. This approach works for static compilation but requires a controlled environment in which compilation happens in exactly the same way in every re-compilation.

Measurement Noise In our research, we identified three sources of measurement noise: 1) *compilation noise*, which is specific to dynamic compilation where the compiler runs in parallel to the executed programs. This can lead to different compilations of the same source function, depending on the moment of compilation and the current profiling

information. 2) *usage noise*, where a method's execution times have a high variance depending on parameters or global variables. 3) *environment noise*, which is caused by CPU frequency scaling, scheduling or caching.

Compilation Forking We designed *compilation forking* to address these sources of measurement noise without the need of a controlled environment. Figure 3.2 shows its abstract process on source code level. In our implementation, all transformations and instrumentations are performed on the compiler IR. The dynamic compiler compiles a function up to

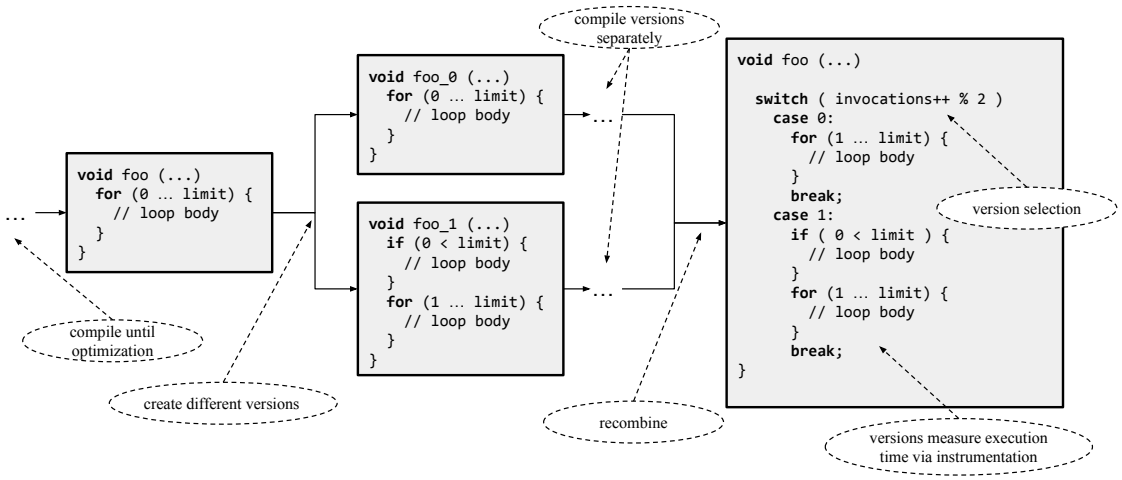


Figure 3.2: Compilation forking [113] (simplified). Timestamp instrumentation omitted.

the point where an optimization of interest is applied—*loop peeling* in case of Figure 3.2. Then, it makes multiple copies (*forks*) of this intermediate compilation state and applies optimizations with different parameters to each copy. Figure 3.2 shows `foo_0` where the loop is not peeled and `foo_1` where the first loop iteration is peeled. This approach ensures that all versions share the same past, i.e., the same *compilation noise* up to the *fork* point. After the compiler has finished compiling the versions independently, it adds to each version an instrumentation for measuring its aggregated *self time* during execution (omitted in Figure 3.2), which excludes calls to other functions as well as safepoints, where garbage collection can take place. Then, the compiler recombines the compiled and instrumented versions and adds to the recombined function a dispatch logic to execute one of the forks each time the recombined function is called. Thus, the forks are executed in a round-robin scheme, which ensures that each version is called consistently throughout different phases of the program execution. This averages out *usage noise*. *Environment noise* cannot be fully controlled and is handled by a custom outlier detection instrumentation.

We claim that *compilation forking* produces consistent measurements of method-local compilation decisions in a dynamic compiler. To verify this claim, we used *compilation forking* to generate training data for creating two optimization heuristics with machine learning. Each time a compilation is forked, we collect features of the compilation and infer the best parameter from the performance measurements of each fork. Following this approach, we created one machine learning model for deciding whether to peel the first loop iteration or not, and another model for selecting a loop unroll factor $\in 1, 2, 4, 8, 16, 32$. Our experimental results, which can be found in full extent in the paper contained in Chapter 6, showed that models which are trained on data produced by *compilation forking* perform similarly to the highly-tuned hand-crafted heuristics in the GraalVM compiler. This indicates that *compilation forking* enables creating accurate performance data in dynamic compilers.

We also evaluated the impact of *compilation forking* on the total program compile time, the execution time and the code size. Compile time and code size grow linearly with the number of forks, whereas the program execution time is affected by the instrumentation and depends on the proportion of method size to instrumentation instructions. However, the execution time overheads only occur outside the measurement regions. The detailed evaluation for each benchmark suite can be found in Chapter 6.

In our initial experiments, we performed *compilation forking* offline to create data for model training. After deploying the trained model, *compilation forking* is disabled. However, as *compilation forking* runs transparently to the user, it can also be used in a production system, which inspired our next research topic.

3.4 Self-optimizing Models

Traditional approaches, which use machine learning in compilers [10; 157], have a clear distinction between the model training phase, which happens offline, and the model usage, which happens after deploying the compiler. Reinforcement learning [77] is an exception where the performance of each compiled program can be used immediately as feedback to improve the machine learning model. However, existing approaches [63; 104] use reinforcement learning only prior to deploying the compiler.

In our research on *self-optimizing compiler heuristics* [112] we propose a system where model training, model usage and model refinement can happen in production at user site. We dynamically switch between data generation and model usage in a single program run.

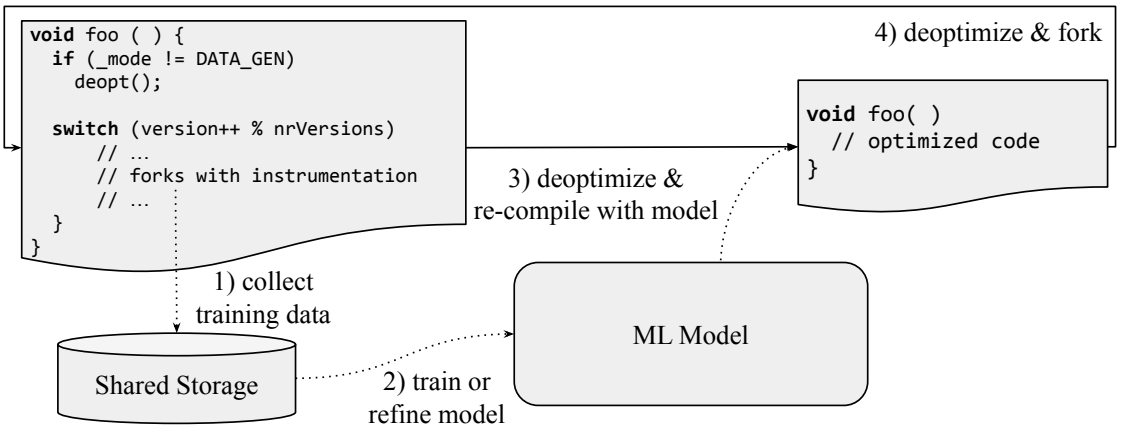


Figure 3.3: Self-optimizing compiler heuristics [112]. Simplified workflow.

Figure 3.3 summarizes the processes within our system; a more detailed architecture and component description can be found in Chapter 7. During data generation, all functions are compiled with *compilation forking* [113] and the extracted feature and performance data for learning an optimization heuristic is collected in a shared storage. After a pre-defined time, a learning server fetches the generated training data and either creates a new model or updates an existing model. Then the server provides the machine learning model to the compiler, which deoptimizes [70] and re-compiles the functions without forking. During re-compilation, the machine learning model works as a heuristic for selecting the learned optimization parameter values to improve the program's performance. For long-running server applications, this process can be executed multiple times in a single program execution to account for changes in the program's usage over time.

We selected a client-server architecture to relief the user system from model training, which requires additional software and hardware. The learning server fulfills several tasks: it pre-processes the training data, removes less informative or potentially noisy data and, when creating a new model, selects relevant features. The pre-processing steps, the features and the model architecture can be found in Chapter 7.

We evaluated our approach in the GraalVM compiler [170] with the loop peeling optimization, which has already been used in our research on *compilation forking* [113]. However, in the research on self-optimizing models [112], the goal was to obtain highly-specialized models, and overfitting was deliberately taken into account. We formulated two hypotheses: 1) that highly-specialized machine-learning-based heuristics can increase the peak performance of compiled programs and 2) that pre-trained models can be tuned towards a new environment during dynamic compilation. We tested the first hypothesis by creating a new loop peeling model for each benchmark using the automatized approach

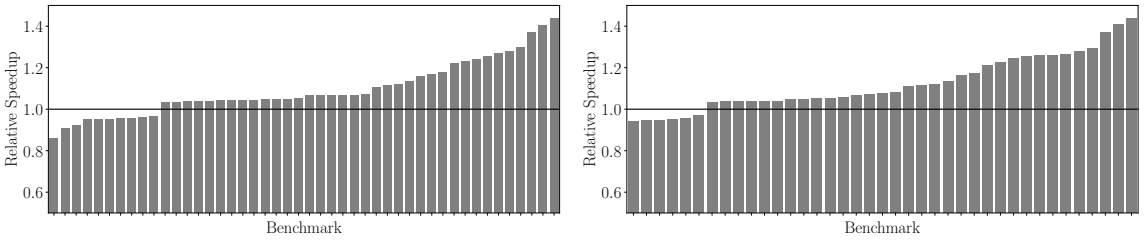
from Figure 3.3. For benchmarks from the *JetStream* [128] and *Octane* [27] suites, multiple significant speedups could be measured, four of them exceeding 30%. To test the second hypothesis, we first executed our approach on the *Xalan* benchmark from the *DaCapo* [16] suite, to create a new model. We tested this *Xalan* model on the *gcc-loops* benchmark from the *JetStream* suite, which showed no speedup at all. After automatically refining the model with the data from the *gcc-loops* benchmark, a significant speedup could be encountered for *gcc-loops*. The performance of the *Xalan* benchmark when using the model refined for *gcc-loops* showed no regressions. For detailed results for each benchmark we refer to Chapter 7.

With our research on self-optimizing compiler heuristics we showed that machine learning can be used to automatically create or refine heuristics, which can lead to significant improvements in peak performance. In contrast to related work, our models are automatically refined at the user site without any re-deployment of the compiler.

3.5 Unrolling of Vectorized Loops

Our work on learning heuristics for loop peeling [112; 113] showed that minor transformations can have large performance impacts. This is often caused by interactions with other optimizations, such as vectorization. In our research on unrolling of vectorized loops [111], we investigated potential performance gains when tailoring loop unrolling towards vectorization with machine learning.

This research originated from the absence of dynamic heuristics in the GraalVM compiler for finding the optimal unroll factor for vectorized loops. Currently, the heuristics for selecting the unroll factor for vectorized loops are based on global compiler parameters and benefit some benchmarks but degrade the performance of others. We conducted a study where we learned dynamic heuristics for selecting the unroll factor $VU \in 1, 2, 4, 8, 16$ for vectorized loops. For this study, we refrained from deep learning and used Random Forests [68], which are human-readable and can provide insight into what features and thresholds are important. As training data, we compiled a set of 231 micro-benchmarks with *compilation forking* and extracted program features as well as the optimal unroll factor for each vectorized loop. These are the same micro-benchmarks which have been used by Oracle engineers when designing the vectorization heuristics in the GraalVM compiler. This facilitates a fair comparison between static (global) heuristics and a learned model.



(a) lib | ¬graph | ¬pruned. **(b)** lib | ¬graph | pruned.

Figure 3.4: Relative speedup compared to default GraalVM [111]. Higher is better.

We evaluated three training configuration parameters: 1) including or excluding training data stemming from standard library functions, 2) including or excluding features which describe the whole IR graph surrounding the vectorized loop, and 3) pruning or not pruning the decision trees to a pre-defined depth. The three binary parameters resulted in eight models which we evaluated in terms of accuracy and predicted distribution. Insights from the training process, for example, that tree pruning counters overfitting and yields more accurate results, are discussed in greater detail in Chapter 8. Figure 3.4 shows the performance results of deploying two of the eight models in the compiler. The performance is normalized to the default GraalVM; bars above the horizontal line indicate speedups from the learned model. On average, each of our models outperformed the existing static heuristic by 8%-12%; the best models always involved pruned decision trees.

Due to large compile time increases for some models, we conducted an in-depth analysis on where compile time is spent when using the random forest classifier in production. The detailed results can be found in Chapter 8. This analysis concluded that model loading is an expensive task which suggests pruning decision trees to decrease the model size.

Part II

Publications

Chapter 4

Machine Learning in Dynamic Compilers

This chapter includes the initial publication of this thesis, which outlines our vision of using machine learning in dynamic compilers to improve existing hand-crafted heuristics and assist compiler engineers in the development process.

Paper: Raphael Mosaner. 2020. Machine learning to ease understanding of data driven compiler optimizations. In *Companion Proceedings of the 2020 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity* (SPLASH Companion 2020). Association for Computing Machinery, New York, NY, USA, 4–6. https://doi.org/10.1145/3426430.3429451

Note: This paper was published as part of a Doctoral Symposium. In the corresponding student research competition at SPLASH 2020, this work (paper and poster presentation) was awarded the first place in the graduate category.

Machine Learning to Ease Understanding of Data Driven Compiler Optimizations*

Raphael Mosaner
raphael.mosaner@jku.at
Johannes Kepler University
Linz, Austria

Abstract

Optimizing compilers use—often hand-crafted—heuristics to control optimizations such as inlining or loop unrolling. These heuristics are based on data such as size and structure of the parts to be optimized. A compilation, however, produces much more (platform specific) data that one could use as a basis for an optimization decision. We thus propose the use of machine learning (ML) to derive better optimization decisions from this wealth of data and to tackle the shortcomings of hand-crafted heuristics. Ultimately, we want to shed light on the quality and performance of optimizations by using empirical data with automated feedback and updates in a production compiler.

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

Keywords: Machine Learning, Neural Network, Regression, Dynamic Compiler, Optimization, Heuristics

ACM Reference Format:
Raphael Mosaner. 2020. Machine Learning to Ease Understanding of Data Driven Compiler Optimizations. In *Proceedings of the 2020 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH Companion '20), November 15–20, 2020, Virtual, USA.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3426430.3429451

1 Motivation

There is a large amount of metrics which cause compilers to take vastly different decisions when dynamically compiling code [10]: CPU features, code features, timing or profiling data. Machine learning can be—and has been [10]—successfully used to find near-optimal parameters for driving compiler optimizations. Such parameters include inlining depth, loop unrolling factors or cost models for assessing the quality of optimization opportunities. However, learning-based solutions are often employed as a black box, causing their adoptions by compiler developers to be rather low. Thus, machine learning hardly finds its way into dynamic production compilers. None of HotSpot, JavaScript V8 [11] or Graal compiler [13] are using machine learning to make decisions during dynamic compilation to our knowledge. For LLVM, there is a recent approach[1] trying to use reinforcement learning to improve heuristics in a static compilation setup, which is not in production either. Our motivation is thus, to leverage the advantages of machine learning in the domain of compiler development by creating an iterative approach for incrementally evaluating and optimizing compiler decisions for a state-of-the-art dynamic compiler.

2 Problem

Compiler optimizations often rely on hand-crafted heuristics, which are fine-tuned by compiler experts to provide near-optimal results with respect to pre-defined success metrics. Those metrics are highly domain-specific, like peak performance for long-running applications or minimal code size for embedded software. Tuning compiler heuristics is often an incremental process involving a learning-by-doing approach for compiler developers, as indicated in Figure 1a. Therefore, the quality of hand-crafted heuristics reflects the expertise of compiler engineers and the benchmarks that are used for creating and evaluating the heuristics. In practice, those heuristics are often static and use a one-size-fits-all approach [10]. The pragmatic reason is, that the wide range of customers with varying requirements but no expertise in performance engineering need to be provided with a default solution covering most use cases. Besides, performance heuristics are hardly ever changed, because of unforeseeable implications to the system as a whole, which can cause performance regressions as result of misinterpreted data. There are essentially three problems with hand-crafted heuristics:

- they require domain expertise
- they are often static and one-size-fits-all
- they require manual maintenance and updates based on human-interpreted data

Machine learning can be used to significantly reduce these problems by providing an automated, data driven approach,

*This research project is partially funded by Oracle Labs.

SPLASH Companion '20, November 15–20, 2020, Virtual, USA

[1] http://lists.llvm.org/pipermail/llvm-dev/2020-April/140763.html

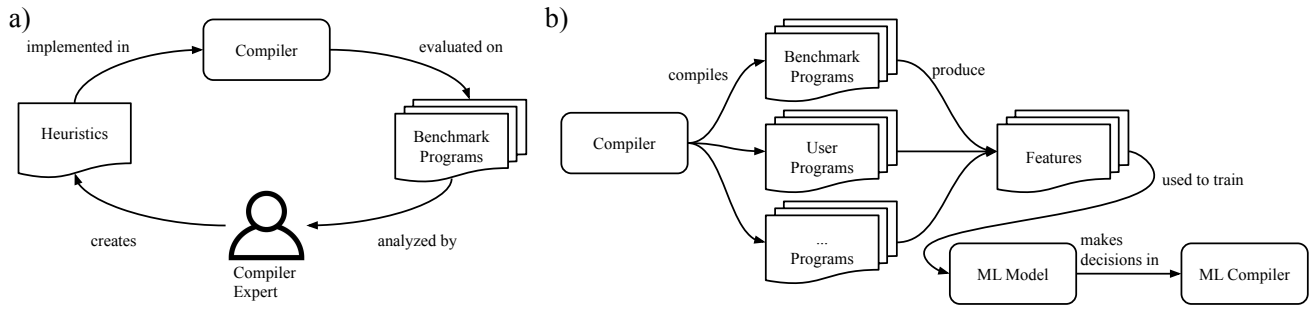


Figure 1. Workflows in existing compilers. a) depicts the iterative process of compiler experts optimizing heuristics and b) illustrates the traditional black-box approach when machine learning is employed in compilers.

which can be used for creating custom heuristics or optimization decisions for different environments. There has been a variety of research in this area over the last decades [2, 12]. Figure 1b depicts the traditional approach for introducing machine learning in a compiler. However, using machine learning as a black box may complicate maintenance and further compiler development on top it. Embedding machine learning into a compiler is also time-consuming, especially in just-in-time (JIT) compilers where compile time directly impacts run time and performance of a program. Thus, machine learning should be used complementary to domain knowledge, to both verify and improve optimization heuristics while introducing automation and maintaining understandability at the same time.

3 Approach

In this paper we propose an approach where machine learning is used in an assistive way to support compiler optimizations. Figure 2 depicts the high-level workflow to obtain understandable, yet data-driven improvements for particular optimizations. It combines features from machine learning—such as automatic adaptation of existing heuristics—with supporting compiler experts to derive new knowledge. It allows an assessment of existing compiler decisions by comparing them against findings that are purely derived from data. There are several studies [7, 10] where machine learning has performed better than human-crafted heuristics. However, they lack any feedback into existing optimizations. The feedback loop in our approach, as indicated in Figure 2, can either be fully automated to react to changes in the environment online, or by providing a compiler expert with information which can be analyzed offline to improve the heuristics.

When defining success metrics for our approach, we have to consider its multi-dimensionality in a dynamic production compiler. We are targeting:

- *performance* of the compiled program
- *compilation time / warmup*
- *maintainability* in the context of automated feedback
- *understandability* of compiler internals (data analysis)

More general, a performance improvement is tied to the performance metrics of the underlying optimization, which

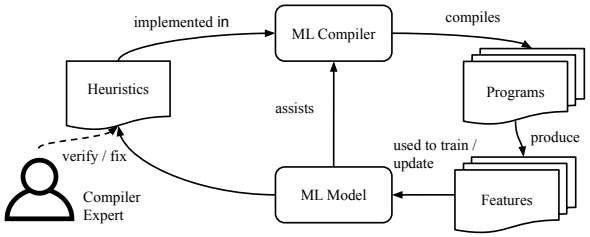


Figure 2. Workflow of assistive machine learning in compilers. Machine learning is used to automatically provide feedback based on observed compilation data.

might include a trade-off between execution time and code size or memory usage. Besides, with automated feedback and optimizations enabled via machine learning, success metrics can be found on the soft side of a compiler, including easier maintainability and more domain-specific compilers in general. For our machine learning pipeline, we use the following abstract steps, which are embedded in Figure 2:

Data Generation: Similar to human expertise, a machine learning model has to build up its knowledge initially. Thus, we need to generate a sufficient amount of data, by compiling a set of benchmark suites (cf. Section 3.1) to extract program characteristics. For future projects we plan to expand the set of learning data, by compiling standard libraries or user programs to train models for particular domains.

Feature Engineering: In a machine learning task, a target value is predicted using a set of input features fed into a model. For the domain of compilation, these features can be roughly grouped into static, dynamic, and graph-based [12]. Their number can be important when it comes to model size and prediction speed, which both are crucial factors in a dynamic production compiler. Depending on the problem context, the number of features can be reduced by removing correlating features. Principal component analysis (PCA)[1] might be a viable option to obtain maximum information from a minimum number of features. However, PCA creates new features by combining existing ones, which reduces overall understandability and should therefore be omitted if there are more intuitive ways for reducing features.

Learning: There is a variety of different learning techniques to build machine learning models [2, 12]—most of

them are applied offline. However, we plan to automate the process of updating the model online after new data is encountered.

Feedback: One paramount component in our assistive ML approach is the feedback loop which manifests itself on multiple occasions. As indicated in Figure 2, the ML model should be automatically updated after new data has emerged. Furthermore, feedback regarding the quality of heuristics should be automatically incorporated by updating (static) heuristics. Ultimately, compiler experts should be provided with data to investigate compiler internals based on findings from learned data.

3.1 Evaluation Methodology

We claim that machine learning can greatly help with improving compiler optimizations. To evaluate this claim, we implement our approach in the Graal compiler [4, 13]. We plan to train our predictors using benchmark suites such as *dacapo* [3], *scala-dacapo* [9], *renaissance* [8], *octane*[2] and *jetstream*[3] for an initial evaluation. Regarding performance, we want to compare the expert-created heuristics in Graal against our ML predictors with respect to compile time, code size, and peak performance. For these comparisons we will conduct experiments with known benchmarks as well as unknown user programs to also assess the generalization of both models under comparison.

3.2 Case Study

In this section, we describe a case study on how to improve an existing compiler optimization with assistance of machine learning. The targeted optimization is *code duplication* [5]. Its idea is to copy code at control flow merges into the predecessors blocks, which can enable further optimizations. Leopoldseder et al. [6] use a trade-off between estimated code growth versus estimated number of saved cycles to trigger duplication. They created a cost model for annotating each node of the compiler's intermediate representation (IR) with an estimated abstract size and number of execution cycles. This cost model is hand-crafted by compiler experts and provides significant performance improvements when used in heuristics.

Instead of assigning abstract sizes for each node, our approach uses machine learning to more accurately predict the code size after compilation. We trained an ANN for learning the non-linear relation between the number of IR nodes (features) and the code size (target) *after several optimization phases*. Using ML as an assistive technology, speed-ups of up to seven percent were encountered for some benchmarks. Currently, our feedback process provides the compiler expert with compilation units were duplication decisions differ to gain insight into flaws of the existing cost model. For instance, we were able to find and fix issues where size changes were underestimated leading to code size bloats (15% for jetstream's *towers* and *containers* benchmarks) without any performance improvements in return.

4 Conclusion

The approach presented in this paper aims to close the gap between the domains of dynamic compiler optimization and machine learning in a production environment. It tries to use both disciplines in a complementary way. Instead of replacing compiler logic by ML black boxes and give up on understandability we rather assist existing compiler optimizations and incorporate findings from ML to extend expert knowledge. The proposed approach in the domain of code duplication can be seen as one application area. In the future, we plan to transfer it also to other domains such as inlining or profile-guided optimizations (PGO).

References

[1] H. Abdi and L. Williams. 2010. Principal Component Analysis. (2010). https://doi.org/10.1002/wics.101

[2] A. Ashouri, W. Killian, J. Cavazos, G. Palermo, and C. Silvano. 2018. A Survey on Compiler Autotuning Using Machine Learning. (2018). https://doi.org/10.1145/3197978

[3] S. Blackburn, R. Garner, C. Hoffmann, A. Khang, K. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, E. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. ACM. https://doi.org/10.1145/1167473.1167488

[4] G. Duboscq, L. Stadler, T. Würthinger, D. Simon, C. Wimmer, and H. Mössenböck. 2013. Graal IR: An Extensible Declarative Intermediate Representation.

[5] D. Leopoldseder, L. Stadler, T. Würthinger, J. Eisl, D. Simon, and H. Mössenböck. 2018. Dominance-based Duplication Simulation (DBDS): Code Duplication to Enable Compiler Optimizations. In *CGO*. https://doi.org/10.1145/3168811

[6] D. Leopoldseder, L. Stadler, T. Würthinger, J. Eisl, D. Simon, and H. Mössenböck. 2018. Dominance-Based Duplication Simulation (DBDS): Code Duplication to Enable Compiler Optimizations. In *CGO*. https://doi.org/10.1145/3168811

[7] A. Monsifrot, F. Bodin, and R. Quiniou. 2002. A Machine Learning Approach to Automatic Production of Compiler Heuristics. In *AIMSA*. Springer-Verlag. http://dl.acm.org/citation.cfm?id=646053.677574

[8] A. Prokopec, A. Rosà, D. Leopoldseder, G. Duboscq, P. Tůma, M. Studener, L. Bulej, Y. Zheng, A. Villazón, D. Simon, T. Würthinger, and W. Binder. 2019. Renaissance: Benchmarking Suite for Parallel Applications on the JVM. ACM. https://doi.org/10.1145/3314221.3314637

[9] A. Sewe, M. Mezini, A. Sarimbekov, and W. Binder. 2011. Da Capo Con Scala: Design and Analysis of a Scala Benchmark Suite for the Java Virtual Machine. ACM. https://doi.org/10.1145/2048066.2048118

[10] D. Simon, J. Cavazos, C. Wimmer, and S. Kulkarni. 2013. Automatic Construction of Inlining Heuristics Using Machine Learning. In *CGO*. https://doi.org/10.1109/CGO.2013.6495004

[11] V8 JavaScript Compiler 2020. https://github.com/v8/v8

[12] Z. Wang and M. O'Boyle. 2018. Machine Learning in Compiler Optimization. (2018). https://doi.org/10.1109/JPROC.2018.2817118

[13] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. 2013. One VM to Rule Them All. In *Onward!* ACM. https://doi.org/10.1145/2509578.2509581

[2] https://github.com/chromium/octane

[3] https://browserbench.org/JetStream/

Chapter 5

Predicting Code Size

This chapter includes the paper which presents our first attempts of using machine learning during dynamic compilation by training a model for predicting the code size impact of optimizations.

Paper: Raphael Mosaner, David Leopoldseder, Lukas Stadler, and Hanspeter Mössenböck. 2021. Using machine learning to predict the code size impact of duplication heuristics in a dynamic compiler. In *Proceedings of the 18th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes* (MPLR 2021). Association for Computing Machinery, New York, NY, USA, 127–135. https://doi.org/10.1145/3475738.3480943

Using Machine Learning to Predict the Code Size Impact of Duplication Heuristics in a Dynamic Compiler*

Raphael Mosaner
raphael.mosaner@jku.at
Johannes Kepler University
Linz, Austria

David Leopoldseder
david.leopoldseder@oracle.com
Oracle Labs
Vienna, Austria

Lukas Stadler
lukas.stadler@oracle.com
Oracle Labs
Linz, Austria

Hanspeter Mössenböck
hanspeter.moessenboeck@jku.at
Johannes Kepler University
Linz, Austria

Abstract

Code duplication is a major opportunity to enable optimizations in subsequent compiler phases. However, duplicating code prematurely or too liberally can result in tremendous code size increases. Thus, modern compilers use trade-offs between estimated costs in terms of code size increase and benefits in terms of performance increase. In the context of this ongoing research project, we propose the use of machine learning to provide trade-off functions with accurate predictions for code size impact. To evaluate our approach, we implemented a neural network predictor in the GraalVM compiler and compared its performance against a human-crafted, highly tuned heuristic. First results show promising performance improvements, leading to code size reductions of more than 10% for several benchmarks. Additionally, we present an assistance mode for finding flaws in the human-crafted heuristic, leading to improvements for the duplication optimization itself.

CCS Concepts: • **General and reference** → *Performance*; *Empirical studies*; • **Software and its engineering** → **Just-in-time compilers**; **Dynamic compilers**; • **Computing methodologies** → **Supervised learning by regression**; **Neural networks**.

Keywords: Code Duplication, Machine Learning, Neural Networks, Regression, Dynamic Compiler, Optimization, Heuristics

*This research project is partially funded by Oracle Labs.

MPLR '21, September 29–30, 2021, Münster, Germany

ACM Reference Format:
Raphael Mosaner, David Leopoldseder, Lukas Stadler, and Hanspeter Mössenböck. 2021. Using Machine Learning to Predict the Code Size Impact of Duplication Heuristics in a Dynamic Compiler. In *Proceedings of the 18th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR '21), September 29–30, 2021, Münster, Germany*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3475738.3480943

1 Introduction

Modern, optimizing compilers are complex software systems, requiring broad domain expertise to grasp the impacts of transformations on subsequent optimizations in order to make right overall decisions. Dynamic compilers, which use profiling-based speculative optimization and deoptimization [33] tend to rely even more on large amounts of metrics to guide the compilation. These metrics can be CPU features, code features, timing, memory or profiling data [28]. Using these metrics, heuristics for selecting optimization parameter values are defined, which include inlining heuristics [2], loop unrolling factors [26] or cost models [20] for assessing the quality of optimization opportunities. The latter are especially important when it comes to *enabling optimizations* where the reward of a—possibly costly—transformation might not be evident right away. Code duplication [21] is such an enabling optimization, which can lead to increased performance and sometimes even to reduced code size, despite its name. This is only possible by employing highly tuned human-crafted cost models [20] for estimating the potential of future optimizations which will be enabled by duplicated code. However, when done prematurely, duplication can lead to code size bloats. Thus, trade-offs for code size and performance have to be made in the process. Trading-off *estimated* code size and performance can be tedious, as the impact of subsequent optimizations has to be taken into account. We therefore propose the use of machine learning for predicting the code size impact over several optimization

phases. Machine learning can be—and has been [3, 28, 32]—successfully used to find near-optimal parameters for driving compiler optimizations. Code size is often considered as an important success metric [8, 9], however, we are not aware of research on directly predicting the code size impact in a dynamic compilation pipeline. This paper contributes the following:

- A machine learning pipeline for predicting the code size impact of optimization passes in dynamic compilers.
- An elaborate evaluation of a trained neural network compared to a human-crafted estimator using the same input features.
- An outline of an assistance mode, to facilitate finding bugs in human-crafted heuristics.

The rest of this paper is structured as follows. Section 2 discusses some background on the duplication optimization as well as related work on machine learning in compilers. Section 3 presents our approach for using a trained model for predicting the code size impact and deploying it as part of a duplication heuristic. Section 4 presents our evaluation methodology, followed by Section 5 which shows results and an analysis thereof. Eventually, in Section 6 we point out where this research is heading, by discussing future work.

2 Background

This ongoing work applies machine learning to assist a code duplication heuristic [20, 21] by predicting its code size impact. We therefore discuss the code duplication optimization, initially. Then, related work on machine learning in compilers is outlined.

2.1 Code Duplication

Code duplication, as proposed by Leopoldseder et al. [21], is an enabling optimization, which copies code at control flow merges into predecessor blocks to enable subsequent optimizations. Figure 1 depicts a scenario for applying duplication to enable constant folding. The first code snippet shows a conditional assignment to *phi* which is used in an arithmetic operation at the control flow merge (i.e. `return`). By pulling the *return* statement upwards into both branches, duplicating it in the process, constant folding can be applied in a later optimization phase. In this toy example, code size can be even reduced eventually. However, in the majority of cases, duplication is especially useful for fast-path optimizations, where code size is increased for less important branches to enable optimizations for the fast-path [21].

Leopoldseder et al. [20] trade-off code size growth and performance gain, by introducing a cost model for graph-based compiler intermediate program representations (IR) [12, 13]. They assign abstract sizes and cycles to each IR node, modeling the node's impact on the final compilation in terms of code size and execution time. Figure 2 shows the IR graph

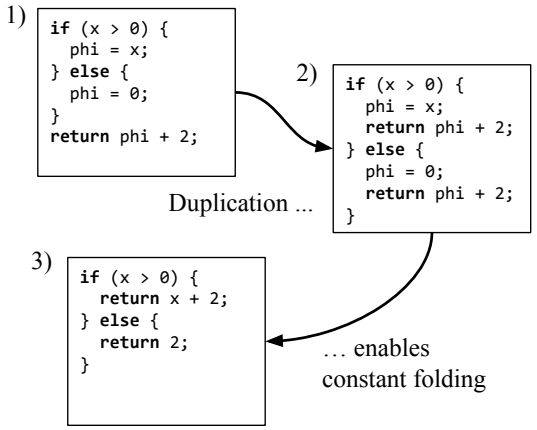


Figure 1. Code duplication as enabling optimization.

for the `return`-statement from Figure 1 annotated with abstract sizes. When Leopoldseder et al. [21] evaluate a duplication opportunity, they simulate enabled optimizations like constant folding or conditional elimination. Using the cost model, estimations about code size and execution cycles can be made for graphs before and after simulated duplication. A duplication is only performed, if the estimated benefit, calculated from reduction in cycles, exceeds the cost, which corresponds to code size increase. Furthermore, a threshold for maximum code size limits overall applicability of duplication.

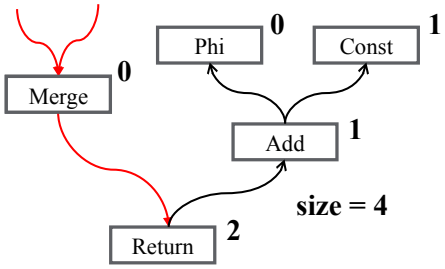


Figure 2. IR graph with assigned node sizes / costs. Upward edges denote data flow, downward edges denote control flow.

2.2 Machine Learning in Compilers

Machine learning has been [3, 19, 28, 32] successfully used to find near-optimal parameters for driving compiler optimizations. Initially, iterative compilation [5, 19] was used for re-compiling the same program multiple times with modified compilation parameters, ultimately converging on a (near-)optimal compilation strategy. There is extensive work on finding the best global compiler flag setup for given programs [3, 32]. By abstracting code to a set of descriptive features, machine learning models have then been introduced to establish a more general relationship between

code patterns and optimization parameters [3, 19, 32]. The model types vary, from decision trees [23, 28], genetic algorithms [7, 8, 30, 31] to neural networks [6, 10, 22, 28]. Especially the use of deep neural networks [10, 19] has allowed outsourcing engineering effort for feature engineering and feature importance analysis to the model. Thus, the traditional offline learning pipelines have adopted modern deep neural networks as their main instrument. However, a second branch has emerged, incorporating reinforcement learning [15, 17] in compilers, continuously improving the compilation process by rewarding advantageous decisions.

Domain-wise, machine learning models for making inlining [7, 28] or vectorization decisions [15], finding loop unrolling factors [23, 30] or tackling the phase ordering problem [17] have been proposed in the past. Apart from directly predicting the best optimization decisions, more general approaches aim towards predicting the performance impact of any compilation decision [11, 22]. However, while code size was a target for optimization in the past [8, 9] it has been neglected in more recent literature [3].

In contrast to the reportedly good results, learning-based solutions are often employed as a black box, causing their adoptions by compiler developers to be rather low. Thus, despite extensive research and successful implementations in research compilers such as Jikes RVM [7] or MILEPOST GCC [14], machine learning hardly finds its way into dynamic production compilers. To the best of our knowledge, none of the compilers in HotSpot, JavaScript V8 [1] or Graal [34] are using machine learning to make decisions during dynamic compilation. For LLVM, there is a recent approach[2] trying to use reinforcement learning to improve heuristics in a static compilation setup, which is not in production either.

3 Approach

In this section, we present an approach for replacing the code size estimation heuristic of the GraalVM compiler [34] with a learned model. As for every machine learning task, this process can be subdivided into sub-tasks for *feature engineering*, *data generation* and *model training*. In the following, these components will be discussed, outlining peculiarities and trade-offs we had to take. At the very end, we cover the deployment in the compiler.

3.1 Feature Engineering

In a machine learning task, a mapping between feature values and a prediction target value is established via application of a learning algorithm. The features contain all relevant data for describing the system or environment at hand and are used as input for the inference model. In the domain of compilers, features can be roughly grouped into static, dynamic, and graph-based features [32]. Due to Graal's intermediate representation (IR) [12, 13], resembling a control-flow-data-flow graph (CDFG), most features describing a compilation can be categorized as graph-based. Also the node counts, used in the existing heuristic, are graph-based features, which however correlate to static features taken from source code. For example, the feature *#AddNode* would correspond to the number of add operations in the source code. While having access to more versatile data than the one used in the human-crafted node cost model, we initially wanted to experiment with the same features for better comparability. Therefore, our feature vectors consist of counters for over 450 different node types which are currently used in the Graal IR. This large number of features can be important when it comes to model size and prediction speed, which both are crucial factors in a dynamic production compiler. Thus, depending on the problem context, the number of features can be reduced by removing correlating features. Also principal component analysis (PCA) [1] might be a viable option to obtain maximum information from a minimum number of features. PCA achieves this, by projecting data points from an n-dimensional space, where n is the number of features, into an m-dimensional space, with $m < n$, while keeping the information loss minimal. In our initial experiments, we refrained from using any feature reduction measures. Firstly, because we deem a comparison between a human-crafted and a learned model more interesting if the same features are used. Secondly, because the use of PCA, which creates a reduced feature set by linearly combining existing features, would degrade understandability and transparency of the model. For future work, we also plan to consider other features than node counts, and will experiment with feature reduction.

3.2 Data Generation

Generating data for training a machine learning model requires extracting feature values and corresponding target labels during compilation and execution. Extracting the node count features is straight-forward, as for each duplication candidate the exact set of nodes to be duplicated is known. However, extracting the target label, which is the code size impact of a duplication, is a tedious task for multiple reasons. The impact of a duplication can only be measured by comparing—ceteris paribus—two compilations C and C' of the same function which differ only in the duplication decision for one duplication candidate X:

$$impact_{size} = size(C_{dupX}) - size(C'_{\neg dupX}) \quad (1)$$

To create C and C' in this way, full control over the compilation process would be required. This is not given in a dynamic compiler as compiler flags typically enable or disable optimizations only globally [32]. Furthermore, in a dynamic compiler the compilation process itself is non-deterministic,

[1] https://github.com/v8/v8

[2] http://lists.llvm.org/pipermail/llvm-dev/2020-April/140763.html

as compiler and application threads are executed in parallel. Thus, non-determinism can be introduced by timing or memory thresholds [29] during compilation but also by deoptimization [16]. Therefore, modifying the compiler to only locally enable duplication would still result in noisy data when using Equation (1) for generating impact labels.

To account for the problems regarding non-determinism and measurement granularity, we decided to use a different prediction target compared to the node cost model presented by Leopoldseder et al. [20]. Rather than predicting the code size *impact*, which is the code size increase or decrease of a particular duplication, our model learns the relationship between the whole graph of a compilation to the final byte code size. This is only a minor change to the machine learning model training as discussed in Section 3.3, but removes the dependency on consistency between data points from different compilations. Section 3.4 addresses how the trained model is incorporated in the compiler to predict the impact of particular optimizations.

In our initial approach, data generation happens at two stages in the compiler: Node count features for the whole graph are extracted after the duplication phase and the final code size is extracted at the end of the compilation pipeline. This happens in a single compilation pass. For generating a sufficient amount of data, a set of benchmark suites (cf. Section 4) was executed to extract node counts and code sizes, resulting in about 300 000 feature vectors. For future projects we plan to expand the set of learning data, by compiling standard libraries or user programs to train models for particular domains. Furthermore, we want to explore ways to reduce compiler non-determinism for generating compilations which are comparable—ceteris paribus—to each other.

3.3 Model Training

Initially, we wanted to train a linear regression-like model to gain insight into weaknesses of the human-crafted heuristics. The idea was to use learned coefficients to replace existing, abstract node costs. However, neither linear, nor polynomial regression models could be trained in a way that the validation set did not exhibit many outliers. This is because of subsequent compiler phases which manipulate the graph and distort any linear relationship, as depicted in Figure 3. Therefore, we trained an artificial neural network (ANN) for learning the non-linear relationship between the number of IR nodes after duplication and the code size *after subsequent optimization phases*. As input vector, we used node counts (features) and byte code size (target) as depicted in Table 1. The network architecture was fairly simple, with two dense hidden layers bisecting the number of inputs twice, using *rectified linear unit* (ReLu) activation functions. In between these hidden layers, batch normalization was responsible for reducing overfitting. We used the Adam optimizer [18] with a learning rate of 10^{-3} and *mean absolute percentage error* (MAPE) as loss function. Figure 4 shows the accuracy of the resulting neural network, visualized as bar plot. For this evaluation, the DaCapo [4] benchmark suite was used for evaluation and therefore excluded from the training. The x-axis depicts the relative error between predicted and actual byte code size, which is aggregated in buckets of size 0.1 (or 10%). On the y-axis, the relative number of methods for each bucket is labeled. Altogether, over 70% of the predictions

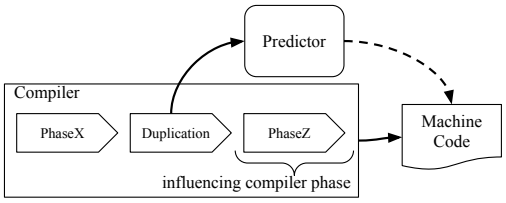


Figure 3. Subsequent compiler phases cause the relationship between compiler graph nodes at duplication time and final machine code size to be non-linear.

Table 1. Input data for training the machine learning model. One function corresponds to one data point which holds node counts (features) and target (code size)

Function	#AddNode	#IfNode	#(…)Node	size
$f1_{benchX}$	27	8	…	924
$f2_{benchX}$	16	1	…	438
$f1_{benchY}$	4	0	…	102

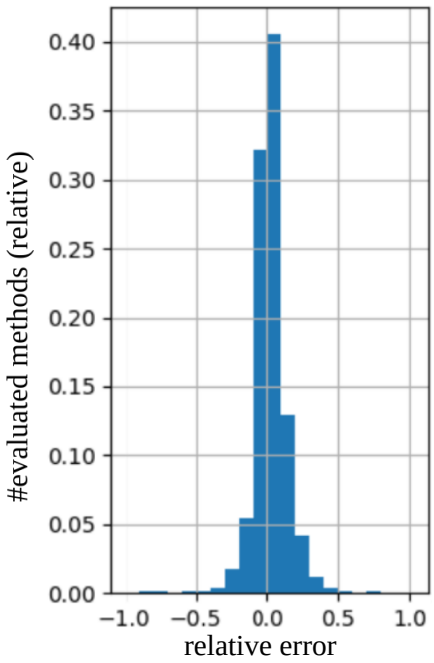


Figure 4. Prediction accuracy of the neural network for the DaCapo benchmark suite.

are less than 10% off from the target label. While this initial network performed very well, we imagine that a tuning of hyperparameters and the network architecture can still improve the results. Furthermore, we also want to experiment with simpler models, like random forest or support vector regressors and compare their predictive power against the neural network.

3.4 Deployment

Due to compiler non-determinism and the lack of comparability, as discussed in Section 3.2, we decided to train our predictor to learn the relationship between node counts from the total IR graph and the total code size. This design choice required adaptations in the deployment in the compiler to fit the model architecture. For each duplication decision our predictor has to be invoked twice to estimate the code size impact of a duplication. First, to predict the final code size when duplication is performed, by adding the counts of all nodes to be duplicated D to the node counts of the whole graph G. Second, to predict the code size with the duplication opportunity being ignored, by using only the graph's node counts G as features. The impact of duplication is calculated as the difference of both predictions:

$$impact_{size} = predict_{size}(G + D) - predict_{size}(G)$$

Using this approach, we can avoid pervasive modifications in the compiler for the data generation as discussed in Section 3.2. Additionally, in contrast to predicting the code size impact from only the duplicated nodes, we assume that providing surrounding nodes to the model will improve the overall accuracy.

During compilation, duplication candidates in GraalVM are evaluated one after another. Therefore, the node count features after duplication D_1 are used as input features for duplication D_2. This ensures an implicit incremental incorporation of previous duplication decisions via updated input features.

Assistance Mode. In addition to replacing the human-crafted code size estimation, we also created a simple assistance mode. When enabled, the duplication heuristic is executed twice, once with the node cost model and once with the neural network predictions in place. If the heuristic produces different duplication decisions based on the used predictor, an output is generated to be analyzed by compiler experts. This output contains information on the location of the duplication candidate, including function and, more precisely, start and end of the duplicated regions in the compiler graph. Besides, a list of nodes in the duplication region is provided, to more easily find patterns where one of the models misperforms.

4 Evaluation Methodology

We claim that machine learning can greatly help to replace and improve human-crafted estimation heuristics. To evaluate this claim, we implemented our approach for predicting byte code size at intermediate points in the compilation pipeline in the GraalVM compiler [12, 34]. Initially, we trained our predictors using well-known benchmark suites such as *DaCapo* [4], *Scala-DaCapo* [27], *Renaissance* [24], *Octane*[3] and *Jetstream*[4]. To cope with non-deterministic compilation behavior, we executed each benchmark multiple times, resulting in over 300 000 data points for training. As training happens offline, we initially omitted an extensive evaluation of training cost but plan this for future work.

While we presented the accuracy of our cross-validated predictor in Section 3.3, a comparison to GraalVM's predictor is hard to make. The existing node cost model predicts code size increases by using abstract node costs, whereas the machine learning predictor works with the code size in bytes. Therefore, we compared the expert-created heuristics in GraalVM against our machine learning predictors with respect to compile time, code size, and peak performance for newly executed benchmark runs. We are aware that the same training programs are used for evaluation. Due to fresh benchmark runs, features for larger compilations are hardly ever equal to the training data. However, smaller functions will likely be equal in training and validation data. In future work, we want to train multiple models more extensively, to evaluate the performance in GraalVM with a full cross-validation over all benchmark suites. We want to point out, that even with cross-validation, many datapoints for standard library methods will be seen in both training and validation data.

5 Results

In this section, we present the performance of the GraalVM compiler with the machine learning model replacing the human-crafted heuristics. Then, we will outline how we improved the existing heuristics based on the findings provided by the afore-mentioned assistance mode.

5.1 Machine Learning Predictor in GraalVM

Figure 5 shows our evaluation of run time, code size and compile time for the Jetstream benchmark suite. Further results can be found in Appendix A. A condensed overview of performance metrics can be found in Table 2. Configuration *GraalVM* depicts the setup with the node cost model used for predicting the code size impact of duplications. *GraalVM_Fixed* presents the version of GraalVM, with machine learning-induced fixes for the node cost model in place. Finally, configuration *ML* uses our neural network predictor for estimating code size changes. In the following, we will

[3] https://github.com/chromium/octane
[4] https://browserbench.org/JetStream/

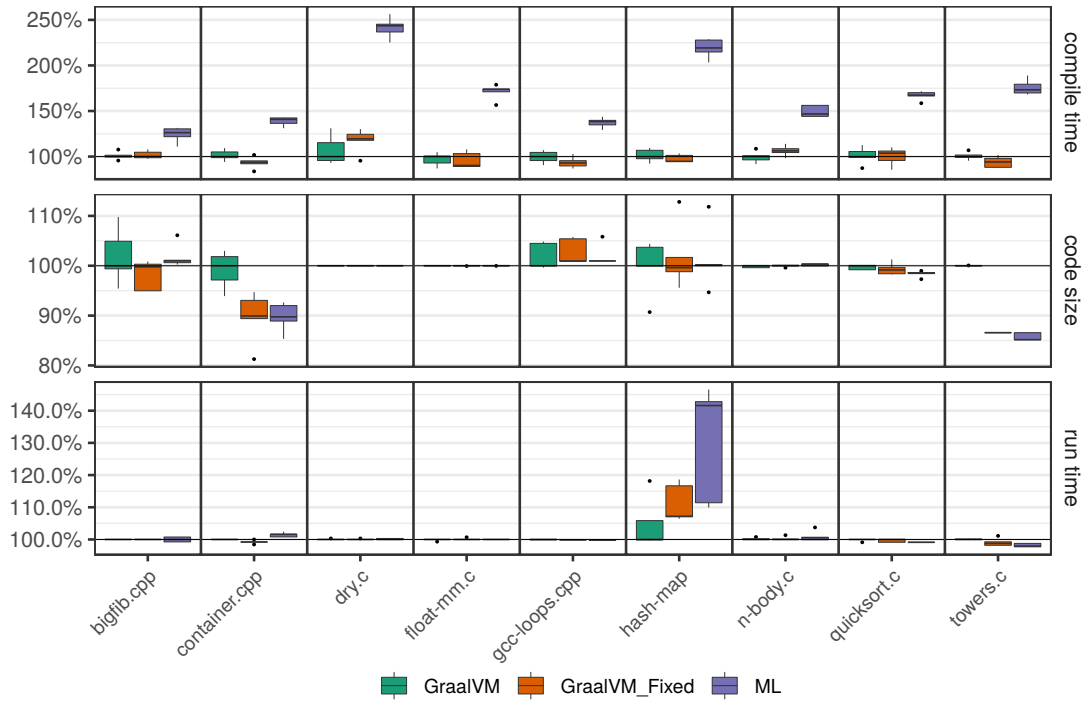


Figure 5. Jetstream benchmark suite results. Lower is better.

Table 2. Geometric mean relative differences normalized to config *GraalVM*. Lower is better.

	GraalVM_Fixed			ML		
	Compile Time	**Code Size**	**Run Time**	**Compile Time**	**Code Size**	**Run Time**
DaCapo	1.0328779	1.0242670	0.9889665	1.3657769	1.2544667	1.0126013
Scala-DaCapo	1.0210548	1.0135434	0.9822896	1.3132403	1.0609208	0.8567309
Renaissance	1.0237099	1.0386490	1.0106240	1.3675285	1.2336947	0.9789691
Octane	1.0187792	1.0061069	1.0024897	1.2637224	1.0211426	1.0005519
Jetstream	0.9953381	0.9739181	1.0098880	1.6543358	0.9749413	1.0286236

compare these configurations regarding compile time, run time and code size.

Compile Time. Especially for compile time, huge overheads are to be expected when using a neural network predictor in production. For most benchmarks, this overhead does not exceed 50%- However there are cases where compile time is slower by 3x. This factor highly depends on the number of duplication opportunities in benchmarks, linearly impacting the number of machine learning predictions. For each duplication, all nodes of the graph have to be counted, in contrast to just the nodes in the duplicated region. This task may also be conducted multiple times for one graph. Hence, there is room for optimization, by caching the whole graph's node counts. Additionally, we will cache the prediction result of the baseline graph to only invoke the neural network once per duplication. Furthermore, the usage of an external library along with model loading and executing the forward pass of the network, exceeds the simplicity of a Java local sum-of-products in the default node cost model. Mendis et al. [22] report equally fast prediction speed of their neural network compared to human-crafted heuristics for throughput prediction. While we do not believe that a *deployed* neural network can beat the simple node cost estimation function, we are certain that optimizing the prediction and deployment process can yield a lot better compilation performance.

Run Time. For most benchmarks, run time (i.e. peak performance) is not affected by using the neural network. Due to increased compile time, the number of warmup iterations had to be increased until a steady state of peak performance was reached. Larger slowdowns of over 10% can be seen for *hash-map* (Jetstream), *avrora* (DaCapo) or *gameboy* (Octane). On the other side, *scaladoc* (Scala-DaCapo), *reactors* (Renaissance) and *typescript* (Octane) exhibit notable speedups. Altogether, no trend towards either better or worse peak performance is visible in our benchmarks. However, we did not hypothesize about particular run time impacts, as the ML

predictor only replaces the code size estimation in the duplication heuristic. Peak performance impacts can be therefore also attributed to the interplay between the changed code size predictor and the remaining heuristics.

Code Size. As expected, impacts on final code size are visible for most benchmarks. A general trend towards larger code size can be seen, which implies more liberal duplication decisions being made with our size predictions in place. However, thresholds in the duplication heuristic are carefully crafted and optimized towards the node cost model. Hence, there are several knobs to be tweaked for better interpretation of the results. Interestingly, benchmarks like *container* and *towers* (both Jetstream) show huge code size reductions when using the learned model. This sub-optimal behavior of duplication led us to implementing the assistance mode.

5.2 Assistance Mode

For some benchmarks, the deployed machine learning model unveiled potential for improving the existing duplication heuristic. Using the assistive approach described in Section 3.4, we were able to guide compiler experts to find patterns where duplication in GraalVM misperformed. As a result, several bugs in the human-crafted duplication heuristic could be found. The biggest being a faulty graph traversal, rendering duplication impact estimation useless for some graph patterns. Additionally, the node cost model had some sub-optimally chosen node sizes. This caused some nodes (e.g. *ReturnNode* and loop related nodes) to be deemed less impactful than they actually are. Configuration *GraalVM_Fixed* in Figure 5 shows the performance of GraalVM with our fixes for both, the duplication heuristic and the node costs in place. Especially for *container* and *towers*, significant code size reductions could be achieved. We received reports that our fixes in GraalVM also improved performance for some Sulong [25] benchmarks (*fannkuch-redux*[5] and a set of internal benchmarks doing decimal number arithmetic) by up to 15%. This assistance mode can be seen as guided fuzzing, where promising paths are explored depending on decisions by a trained machine learning model.

6 Future Work

We presented initial results on replacing human-crafted code size estimation heuristics with a neural network predictor in a dynamic, production compiler. In future work, we plan to improve our approach in following ways.

Feature Selection. For our initial experiments, we used the same features as the human-crafted code size estimator. We expect that a more careful feature selection can improve prediction accuracy even further. Aggregation of node types, or omitting less relevant node types can reduce the network complexity due to fewer inputs. Principal component analysis or similar techniques might be able to trim the feature space down without degrading performance immensely.

Machine Learning Model. We plan to further experiment with hyperparameter tuning for the used neural network, to find optimal parameters for the number of hidden layers, neurons per layer or learning rate. Furthermore, we plan to compare the ANN to other models, such as support vector or decision trees regression.

Compile Time Overhead. Our evaluation showed that the use of an external Java library for loading and executing a stored neural network at run time drastically impacts compile time. We want to reduce this overhead by investigating its origins, replacing, when necessary the library by a simplified but highly tuned implementation for executing the forward pass of our neural network solely.

Prediction Target. While we showed initial results for predicting the code size impact of a duplication, we want to investigate the performance impact as well. Furthermore, duplication itself would be interesting to be replaced by a learned model, rather than just learning sub-heuristics.

Evaluation. For our model evaluation, we used cross-validation to show the prediction quality. With more time at hand, we also want to train multiple models for an extensive leave-one-out cross-validation for the performance evaluation of the deployed model. We might also try to deal with the standard library functions which are found in multiple benchmark suites.

7 Conclusion

This paper presents initial research on an approach for using machine learning to predict impacts of compiler optimizations on final code size. Therefore, we trained a neural network using aggregated node counts taken from compiler IR graphs as features. We implemented our approach in the context of code duplication, where we replaced a human-crafted model in the GraalVM compiler with a neural network predictor. The neural network transparently learns the impact of subsequent compiler phases and predicts the size of the byte code which is eventually emitted by the compiler. Over 70% of our predictions are in a range of 10% around the target value. We evaluated performance of the deployed predictor with respect to compile time, code size and run time. While compile time suffers from additional workload for feature aggregation and network prediction, run time and code size showed both improvements and regressions for different benchmarks. Based on our approach we added an assistance mode, to find patterns where the human-crafted model misbehaved. Using this machine-learning-guided debugging, improvements in the GraalVM compiler could be deployed.

[5] https://benchmarksgame-team.pages.debian.net/benchmarksgame/description/fannkuchredux.html

A Appendix

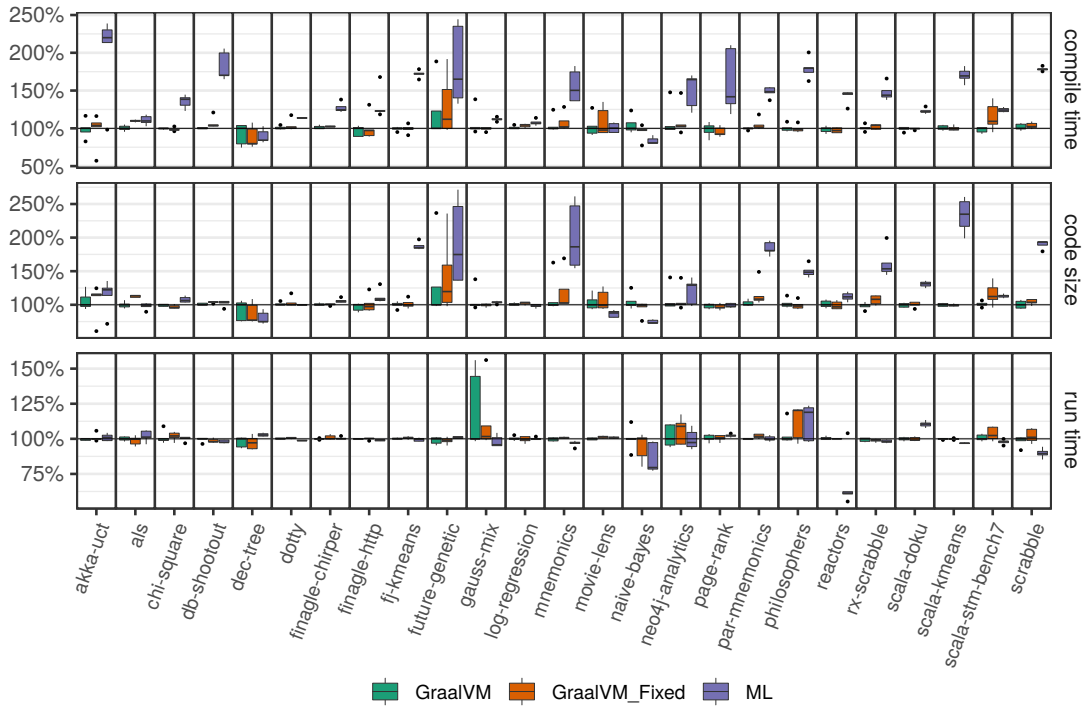


Figure 6. Renaissance benchmark suite results. Lower is better.

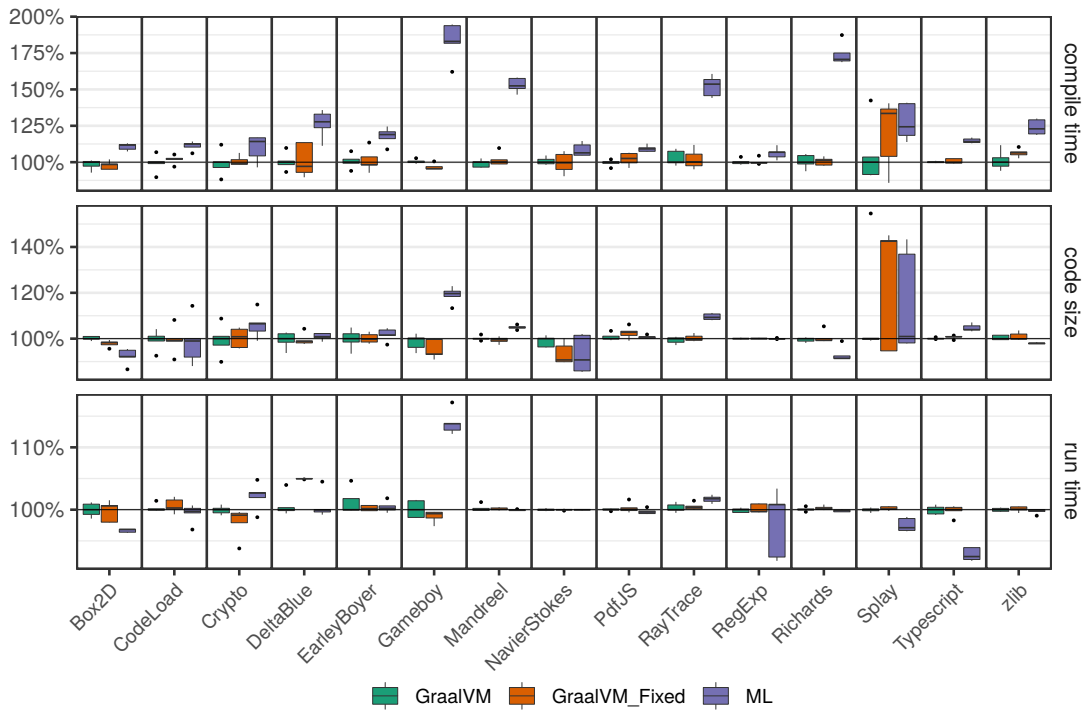


Figure 7. Octane benchmark suite results. Lower is better.

References

[1] H. Abdi and L. Williams. 2010. Principal Component Analysis. *WIREs Comput. Stat.* 2, 4 (July 2010), 433–459. https://doi.org/10.1002/wics.101

[2] M. Arnold, S. Fink, V. Sarkar, and P. Sweeney. 2000. A Comparative Study of Static and Profile-Based Heuristics for Inlining *(DYNAMO '00)*. ACM, New York, NY, USA, 52–64. https://doi.org/10.1145/351397.351416

[3] A. Ashouri, W. Killian, J. Cavazos, G. Palermo, and C. Silvano. 2018. A Survey on Compiler Autotuning Using Machine Learning. *ACM Comput. Surv.* 51, 5, Article 96 (Sept. 2018), 42 pages. https://doi.org/10.1145/3197978

[4] S. Blackburn, R. Garner, C. Hoffmann, A. Khang, K. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis *(OOPSLA '06)*. ACM, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488

[5] F. Bodin, T. Kisuki, P. Knijnenburg, M. Boyle, and E. Rohou. 2000. Iterative compilation in a non-linear optimisation space. *Workshop on Profile and Feedback-Directed Compilation* (03 2000).

[6] A. Brauckmann, A. Goens, S. Ertel, and J. Castrillon. 2020. Compiler-Based Graph Representations for Deep Learning Models of Code *(CC '20)*. ACM, New York, NY, USA, 201–211. https://doi.org/10.1145/3377555.3377894

[7] J. Cavazos and M. O'Boyle. 2005. Automatic Tuning of Inlining Heuristics. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing (SC '05)*. IEEE Computer Society, USA, 14. https://doi.org/10.1109/SC.2005.14

[8] K. Cooper, P. Schielke, and D. Subramanian. 1999. Optimizing for Reduced Code Space Using Genetic Algorithms. *SIGPLAN Not.* 34, 7 (May 1999), 1–9. https://doi.org/10.1145/315253.314414

[9] K. Cooper, D. Subramanian, and L. Torczon. 2002. Adaptive Optimizing Compilers for the 21st Century. *J. Supercomput.* 23, 1 (Aug. 2002), 7–22. https://doi.org/10.1023/A:1015729001611

[10] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather. 2017. End-to-End Deep Learning of Optimization Heuristics *(PACT '17)*. 219–232. https://doi.org/10.1109/PACT.2017.24

[11] C. Dubach, J. Cavazos, B. Franke, G. Fursin, M. O'Boyle, and O. Temam. 2007. Fast Compiler Optimisation Evaluation Using Code-Feature Based Performance Prediction *(CF '07)*. ACM, New York, NY, USA, 131–142. https://doi.org/10.1145/1242531.1242553

[12] G. Duboscq, L. Stadler, Th. Würthinger, D. Simon, C. Wimmer, and H. Mössenböck. 2013. Graal IR: An Extensible Declarative Intermediate Representation. In *Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop*.

[13] G. Duboscq, T. Würthinger, L. Stadler, C. Wimmer, D. Simon, and H. Mössenböck. 2013. An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler *(VMIL '13)*. ACM, New York, NY, USA, 1–10. https://doi.org/10.1145/2542142.2542143

[14] G. Fursin, C. Miranda, O. Temam, M. Namolaru, A. Zaks, B. Mendelson, E. Bonilla, J. Thomson, H. Leather, C. Williams, M. O'Boyle, P. Barnard, E. Ashton, E. Courtois, and F. Bodin. 2008. MILEPOST GCC: machine learning based research compiler. (06 2008). https://doi.org/10.1007/s10766-010-0161-2

[15] A. Haj-Ali, N. Ahmed, T. Willke, Y. Shao, K. Asanovic, and I. Stoica. 2020. NeuroVectorizer: End-to-End Vectorization with Deep Reinforcement Learning *(CGO '20)*. ACM, New York, NY, USA, 242–255. https://doi.org/10.1145/3368826.3377928

[16] U. Hölzle, C. Chambers, and D. Ungar. 1992. Debugging Optimized Code with Dynamic Deoptimization *(PLDI '92)*. ACM, New York, NY, USA, 32–43. https://doi.org/10.1145/143095.143114

[17] Q. Huang, A. Haj-Ali, W. Moses, J. Xiang, I. Stoica, K. Asanovic, and J. Wawrzynek. 2019. AutoPhase: Compiler Phase-Ordering for HLS with Deep Reinforcement Learning *(FCCM '19)*. IEEE, 308–308.

[18] D. Kingma and J. Ba. 2014. Adam: A Method for Stochastic Optimization. *International Conference on Learning Representations* (12 2014).

[19] H. Leather and C. Cummins. 2020. Machine Learning in Compilers: Past, Present and Future. In *2020 Forum for Specification and Design Languages (FDL)*. 1–8. https://doi.org/10.1109/FDL50818.2020.9232934

[20] D. Leopoldseder, L. Stadler, T. Würthinger, J. Eisl, D. Simon, and H. Mössenböck. 2018. Dominance-Based Duplication Simulation (DBDS): Code Duplication to Enable Compiler Optimizations *(CGO '18)*. ACM, New York, NY, USA, 126–137. https://doi.org/10.1145/3168811

[21] D. Leopoldseder, L. Stadler, T. Würthinger, J. Eisl, D. Simon, and H. Mössenböck. 2018. Dominance-based Duplication Simulation (DBDS): Code Duplication to Enable Compiler Optimizations *(CGO '18)*. ACM, New York, NY, USA, 126–137. https://doi.org/10.1145/3168811

[22] C. Mendis, A. Renda, S. Amarasinghe, and M. Carbin. 2019. Ithemal: Accurate, Portable and Fast Basic Block Throughput Estimation using Deep Neural Networks *(Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, 4505–4515. http://proceedings.mlr.press/v97/mendis19a.html

[23] A. Monsifrot, F. Bodin, and R. Quiniou. 2002. A Machine Learning Approach to Automatic Production of Compiler Heuristics *(AIMSA '02)*. Springer-Verlag, London, UK, UK, 41–50. http://dl.acm.org/citation.cfm?id=646053.677574

[24] A. Prokopec, A. Rosà, D. Leopoldseder, G. Duboscq, P. Tůma, M. Studener, L. Bulej, Y. Zheng, A. Villazón, D. Simon, T. Würthinger, and W. Binder. 2019. Renaissance: Benchmarking Suite for Parallel Applications on the JVM *(PLDI '19)*. ACM, New York, NY, USA, 31–47. https://doi.org/10.1145/3314221.3314637

[25] M. Rigger, M. Grimmer, and H. Mössenböck. 2016. Sulong - Execution of LLVM-Based Languages on the JVM: Position Paper *(ICOOOLPS '16)*. ACM, New York, NY, USA, Article 7, 4 pages. https://doi.org/10.1145/3012408.3012416

[26] V. Sarkar. 2000. Optimized Unrolling of Nested Loops *(ICS '00)*. ACM, New York, NY, USA, 153–166. https://doi.org/10.1145/335231.335246

[27] A. Sewe, M. Mezini, A. Sarimbekov, and W. Binder. 2011. Da Capo Con Scala: Design and Analysis of a Scala Benchmark Suite for the Java Virtual Machine *(OOPSLA '11)*. ACM, New York, NY, USA, 657–676. https://doi.org/10.1145/2048066.2048118

[28] D. Simon, J. Cavazos, C. Wimmer, and S. Kulkarni. 2013. Automatic Construction of Inlining Heuristics Using Machine Learning *(CGO '13)*. IEEE Computer Society, Washington, DC, USA, 1–12. https://doi.org/10.1109/CGO.2013.6495004

[29] L. Stadler, G. Duboscq, H. Mössenböck, and T. Würthinger. 2012. Compilation Queuing and Graph Caching for Dynamic Compilers *(VMIL '12)*. ACM, 49–58. https://doi.org/10.1145/2414740.2414750

[30] M. Stephenson and S. Amarasinghe. 2005. Predicting Unroll Factors Using Supervised Classification, Vol. 2005. 123– 134. https://doi.org/10.1109/CGO.2005.29

[31] M. Tartara and S. Crespi Reghizzi. 2013. Continuous Learning of Compiler Heuristics. *ACM Trans. Archit. Code Optim.* 9, 4, Article 46 (Jan. 2013), 25 pages. https://doi.org/10.1145/2400682.2400705

[32] Zheng W. and M. O'Boyle. 2018. Machine Learning in Compiler Optimization. *Proc. IEEE* 106, 11 (Nov 2018), 1879–1901. https://doi.org/10.1109/JPROC.2018.2817118

[33] C. Wimmer, V. Jovanovic, E. Eckstein, and T. Würthinger. 2017. One Compiler: Deoptimization to Optimized Code *(CC 2017)*. ACM, New York, NY, USA, 55–64. https://doi.org/10.1145/3033019.3033025

[34] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. 2013. One VM to Rule Them All *(Onward! '13)*. ACM, New York, NY, USA, 187–204. https://doi.org/10.1145/2509578.2509581

Chapter 6

Compilation Forking

This chapter includes a very fundamental paper of this thesis, which presents the core contribution of *compilation forking* and two case studies of how it can be applied to generate data for model training.

Paper: Raphael Mosaner, David Leopoldseder, Wolfgang Kisling, Lukas Stadler, Hanspeter Mössenböck. 2022. Compilation Forking: A Fast and Flexible Way of Generating Data for Compiler-Internal Machine Learning Tasks. In *The Art, Science, and Engineering of Programming*, 2023, Vol. 7, Issue 1, Article 3, pp. 1-29

Note: The last page, which contains a description of the paper's authors, was omitted for brevity.

Compilation Forking: A Fast and Flexible Way of Generating Data for Compiler-Internal Machine Learning Tasks

Raphael Mosaner[a], David Leopoldseder[b], Wolfgang Kisling[a], Lukas Stadler[c], and Hanspeter Mössenböck[a]

a Johannes Kepler University, Linz, Austria

b Oracle Labs, Vienna, Austria

c Oracle Labs, Linz, Austria

Abstract Compiler optimization decisions are often based on hand-crafted heuristics centered around a few established benchmark suites. Alternatively, they can be learned from feature and performance data produced during compilation.

However, data-driven compiler optimizations based on machine learning models require large sets of quality data for training in order to match or even outperform existing human-crafted heuristics. In static compilation setups, related work has addressed this problem with iterative compilation. However, a dynamic compiler may produce different data depending on dynamically-chosen compilation strategies, which aggravates the generation of comparable data.

We propose *compilation forking*, a technique for generating consistent feature and performance data from arbitrary, dynamically-compiled programs. Different versions of program parts with the same profiling and compilation history are executed within single program runs to minimize noise stemming from dynamic compilation and the runtime environment.

Our approach facilitates large-scale performance evaluations of compiler optimization decisions. Additionally, compilation forking supports creating domain-specific compilation strategies based on machine learning by providing the data for model training.

We implemented compilation forking in the GraalVM compiler in a programming-language-agnostic way. To assess the quality of the generated data, we trained several machine learning models to replace compiler heuristics for loop-related optimizations. The trained models perform equally well to the highly-tuned compiler heuristics when comparing the geometric means of benchmark suite performances. Larger impacts on few single benchmarks range from speedups of 20% to slowdowns of 17%.

The presented approach can be implemented in any dynamic compiler. We believe that it can help to analyze compilation decisions and leverage the use of machine learning into dynamic compilation.

ACM CCS 2012

- **Computing methodologies → Machine learning;**
- **Software and its engineering → Dynamic compilers; Just-in-time compilers;**
- *General and reference → Performance;*

Keywords Dynamic Compiler, Optimization, Performance, Data Generation, Neural Network

The Art, Science, and Engineering of Programming

Submitted May 2, 2022

Published June 15, 2022

DOI 10.22152/programming-journal.org/2023/7/3

In *The Art, Science, and Engineering of Programming*, vol. 7, no. 1, 2023, article 3; 30 pages.

1 Introduction

Dynamic optimizing compilers are complex software systems, requiring broad domain expertise to grasp the impacts of single optimization transformations on the overall program performance. Compiler experts typically fine-tune such optimization decisions based on (micro-)benchmarks, where the compilation process and the runtime performance are reproducible and stable. This results in a large set of compiler heuristics which guide the compilation process for real-world programs, by selecting the optimization parameter values to be used. For example, Leopoldseder et al. [29] introduce heuristics for estimating code size and performance impacts of compiler optimizations which are used to choose loop unrolling factors. These heuristics are typically one-size-fits-all and are optimized for code patterns found in the benchmarks used for evaluation. Therefore, different users compile their programs with the same heuristics, which might not be optimal for the particular domain or program. Manually creating heuristics for different domains or programs is often infeasible.

Data-driven approaches—often using machine learning—have been shown to outperform human-crafted heuristics for compiler optimizations [2, 27, 47]. However, the problem of generating appropriate data for training sophisticated models is often hard to solve [13]. First, compiler flags are typically not per compilation but global, which requires creating minimal programs to capture the impact of a compiler flag in isolation. Second, in a dynamic runtime, compilations are not deterministic and subject to profiling data or memory usage. Thus, it is often infeasible to create a setup for a dynamic compiler, where the impact of a single compilation parameter can be measured. Such measurements however, are required to train a machine learning model. Previous research has come up with ways to find optimal compilation plans for single functions in terms of peak performance [5, 24, 27]. However, these approaches are neither generally applicable nor suited to be used in large-scale data generation for dynamic compilers. For arbitrary methods, there are two noise factors which hamper the inference of compiler knowledge based on performance analysis: Firstly, *compilation noise*, where the same method can be compiled in different ways. This is often the case in a dynamic compiler, where compiler threads run in parallel to the executed program and profiling information is used by the compilation process. Secondly, *usage noise*, where—due to different usage scenarios, parameters or global values—a method's execution time might have high variance. While related work ignores one or both of these noise factors, we take both into account to get more reliable measurements.

We propose *compilation forking*, a technique for extracting optimization performance data in a dynamic compilation system for arbitrary programs. Its core idea is to fork method compilations at points of interest to ensure a common profiling and compilation history. A point of interest is a point in an ongoing compilation, for example right before loop unrolling is applied and the compiler has to choose one of multiple unroll factors. Starting from this common past, each compilation is completed with a different optimization parameter, and all resulting method versions are executed alternatingly. This eradicates compilation noise, up to the point of interest, and averages out usage noise in the long run. The generated data can be used to facilitate

quality analysis of compiler optimizations, which can be seen in Figure 1. Therein, the hand-crafted loop peeling heuristic in the GraalVM [51] compiler is analyzed using compilation forking. The x-axis shows bins for the relative impact on execution time

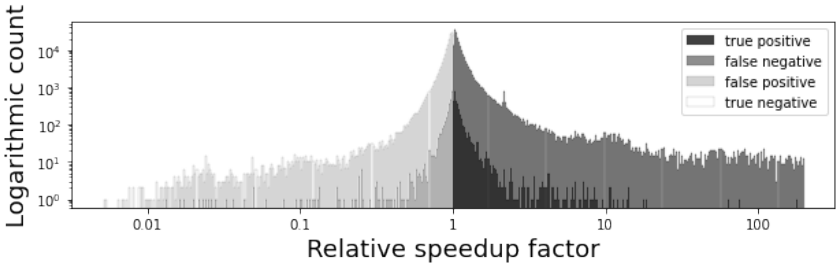


Figure 1 Loop peeling in the GraalVM compiler.

when applying the loop peeling transformation. It is scaled logarithmically. Therefore, values smaller than one indicate a slowdown caused by peeling and values larger than one a speedup. Outliers are cut off in both directions. The y-axis shows the number of loop peeling transformations in the respective bins e.g., the bin at the x-value 10 holds the number of peeling-transformations that led to a speedup of factor 10. The different grey-scales connect the measured performance impacts with the peeling decision that GraalVM would have made: True positive and true negative counts indicate that the GraalVM compiler correctly applied or neglected a loop peeling transformation. False negatives indicate that the compiler did not apply loop peeling although it would have produced a speedup. False positives indicate that the compiler applied loop peeling although it resulted in a slowdown. Large impacts of peeling result from interference with other optimizations (like vectorization) and rare patterns, where peeling enables removing whole loops. Compilation forking allows for such an assessment of compilation decisions for arbitrary programs.

Additionally, compilation forking can be used to train machine learning models in order to replace human-crafted heuristics, which we show in Section 5. In summary, this paper contributes the following:

- Compilation Forking: a novel approach for comparing local optimization decisions in a dynamic compiler under the same conditions on arbitrary programs.
- An elaborate performance measurement strategy that takes compilation noise, usage noise, and OS noise into account.
- A case study where compilation forking data is used to train machine learning models to predict loop optimization parameters.
- An evaluation in which a dynamic highly-optimizing production compiler is matched by learned models for loop optimizations.

The remainder of this paper is structured as follows. Section 2 gives an overview on machine learning in compilers and related work on data generation. Section 3 outlines the general process of compilation forking with Section 4 going into details on implementation specifics. Section 5 summarizes case studies where we trained machine learning models using data which is generated by compilation forking. Finally, in Section 6, we evaluate compilation forking in terms of performance and code size impact. Additionally, we evaluate our machine learning models, to show that compilation forking indeed produces high-quality data.

2 Background

In this section, we briefly provide an overview on machine learning in compilers in general. Subsequently, we point towards related work on data generation for machine learning problems in compilers.

2.1 Machine Learning in Compilers

Over the past decades, machine learning models have been shown to outperform hand-crafted compiler heuristics [2, 19, 27, 47]. Predominantly, learned models aim to improve the peak performance of compiled programs by guiding the compilation process. Other success metrics such as memory usage or code size have been optimized in the past as well [10, 11]. However, decreased memory pressure in modern hardware has led to them being neglected in more recent literature [2], apart from few exceptions [34].

Machine learning in compilers originates from iterative compilation [5, 27]. Its idea is to repeatedly compile programs with different sets of compiler parameters or a different order of compiler phases. There is extensive work on finding the best global compiler flag setup or phase plan for given programs by using iterative compilation [2, 47]. Furthermore, it can be used for creating a gold standard for evaluating other models or heuristics [19]. Auto-tuning frameworks, like OpenTuner [1], focus on reducing the state space for iterative compilations to converge more swiftly on a near-optimal program compilation. For establishing a more general relationship between source programs and beneficial compilation parameters, source code has been abstracted to descriptive features [2, 27, 47]. These features were then used in machine learning models, ranging from decision trees [33, 43], to genetic algorithms [7, 10, 44, 45], support vector machines [36, 41, 44] and neural networks [6, 12, 24, 32, 43]. Developments in the area of deep neural networks have reduced the need for extensive feature engineering and pre-processing by having these tasks taken over by the model itself [12, 27]. Thus, the traditional offline learning pipelines have adopted neural networks as their main instrument. Recently, research towards online learning in compilers has increased. Therein, trained models are improved at run time by rewarding advantageous decisions [21, 23].

Many different compiler optimizations have been investigated with machine learning models. There are models for performing inlining [7, 43] or vectorization decisions [21], for finding loop unrolling factors [33, 44] or for addressing the phase ordering [23] or skipping [24] problems. More general models aim towards predicting the performance impact of arbitrary compiler optimizations, rather than directly predicting the most beneficial optimization decision [14, 32].

In contrast to the reportedly good results in research, to the best of our knowledge, none of the optimizing compilers in HotSpot, JavaScript V8 [46] or GraalVM [51] are using machine learning to make decisions during dynamic compilation. We believe that compiler experts back off from employing machine learning black boxes in compilers due to the concern of degrading understandability and maintainability. Thus, despite successful implementations in research compilers such as Jikes RVM [7] or MILEPOST GCC [19], machine learning is not found in dynamic production compilers.

2.2 Related Work

In principle, our work is related to iterative compilation [5, 27] and multi-versioning [9, 26, 52]. However, the goal of compilation forking is not to produce near-optimal performance of methods by repeated compilation, but to infer impacts of local optimization decisions for later analysis. To the best of our knowledge, past work mainly optimized global compiler flags in iterative compilation for whole programs [19, 43, 44]. For instance, Stephenson et al. [44] re-run benchmarks multiple times with the loop unrolling factors set globally to a particular value. They instrument each loop and compare the performance of loops from different compilations to each other. We investigate compilation parameters locally and execute different versions alternatingly in the same program run for better stability.

Fursin et al. [18] created an approach for supporting iterative compilation by compiling multiple versions of each function, with differing compilation parameters. However, their work focuses more on finding performance stability patterns, which allows for making iterative compilation more feasible by reducing the evaluation time of different versions. They used the EKOPath compiler, which can be used for statically compiling C, C++ or Fortran programs. Our approach uses a dynamic compilation system with deoptimization [49] which enables an even more transparent usage by switching back to one favored version after data generation without interrupting the program.

Multi-versioning [9, 26, 52] is an approach related to iterative compilation, where multiple versions of a function or code snippet are deployed into an executable. At run time, the code which is best optimized towards the current input is selected. Our goal is neither to have multiple versions deployed into a production system, nor to create an optimal version during forking. Rather, we create different (non-optimal) versions to investigate the impact of local optimizations on the function performance in a fine-grained and consistent manner. Based on the gathered information, either improvements in the human-crafted heuristics can be deployed or machine learning models replacing the human-crafted heuristics.

Another research related to our work has been conducted by Sanchez et al. [41] in the IBM Testarossa JIT compiler. Starting from the conventional compilation plan, they successively remove optimization phases to compile different versions of a function. After a number of invocations of the function, re-compilation is triggered and another version is compiled with a different compilation plan. By measuring execution times based on processor timestamp counters they try to learn the best compilation plan for a given method. While their approach was successful for reducing start-up time, the overall throughput was reduced for most benchmarks. We hypothesize that their use of overall method execution time (including callees) instead of self time (excluding callees), the small number of training data points, and the goal of learning a whole phase plan for an already optimized compiler were the main reasons for not outperforming the baseline compiler. Thus, we aim to inspect differences arising from compilations in isolation, with compilation forking enabling us to start from a common past.

Lau et al. [26] also use multi-versioning in the IBM V9 compiler to determine the fastest of two versions of a function with statistical significance. They discovered that a large number of function invocations is necessary to correctly identify speedups between different versions: they argue that at least 1000 invocations are necessary to reason about differences in the 10% ranges. Second, they verified that operating system and usage noise indeed average out in the long run. We could confirm both findings in our own work, yet discovered that the longer an application runs and the more observations are recorded the better the overall data quality becomes. In contrast to their work, our system is capable of creating multiple forks per function for all compiled functions of a program run. This allows our approach to automatically extract observations, without any interaction or function pre-selection by the user. This holistic approach is - in part - enabled by our more fine-grained timestamping instrumentation. We extract self time instead of total time because we want to support all features of a dynamic execution system like the JVM, this includes deoptimization, native calls, garbage collections and transitive function calls. For all of the above "function exits" their total time does not contribute to the actual time spent in a single function with respect to function local optimization decisions. This means any knowledge system built upon this data would be biased towards total time, effectively prohibiting us to later learn any correlation between function local features and the performance of a single compiled, non-exited, piece of code. The use of total time limits the scalability of the approach shown in [26]. Additionally, we do not rely on background worker threads, which might have impacts on the measurements in a meta-circular environment.

More recently, Cummins et al. [13] addressed the problem of too few available data from a different angle compared to our approach. In essence, they use data augmentation by synthesizing programs to enlarge the set of benchmark data and thus the set of data points being usable for machine learning in compilers. They reported improvements of over 25% when training a predictive model with the extended, synthesized data. While they generate artificial benchmarks for collecting data, our approach enables executing arbitrary user programs for collecting data.

In their recently published work, Mpeis et al. [35] investigated the minimal state of a program to be captured online to later replay that programs under changed conditions offline. While our goals to conduct consistent performance measurements for changed optimization parameters are similar, the approaches are fundamentally different. Additionally, our work is not restricted to side-effect and I/O-free programs.

The span of related work, ranging back more than two decades is still present in recent research and shows that (1) *automated* generation of performance data in a (2) *dynamic compilation* system for (3) *local optimizations* is yet an unsolved problem.

3 Compilation Forking

We propose *compilation forking*, a technique which allows for flexible data generation during conventional program execution. Figure 2 shows an abstract depiction of our system architecture. It combines the process of compilation forking with the use case of training a machine learning model from the extracted data. The key idea of

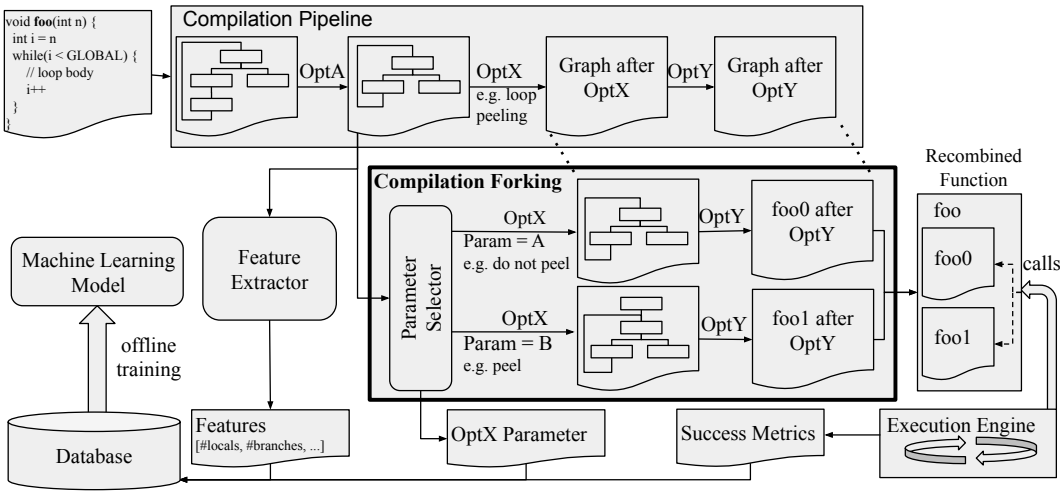


Figure 2 System architecture. Forking happens in the *loop peeling phase*.

compilation forking is to create multiple versions of a compilation, sharing the same compilation and profiling history, but going separate ways at a compilation decision under investigation. Figure 2 shows at the very top the compilation pipeline, where previous transformations such as *OptA* are applied identically to all forks. The center part of the figure depicts the creation of different versions, based on the number of options for *OptX* (e.g. loop peeling). In the center right, the different versions are recombined after they have passed through the remaining compilation pipeline. Each version is executed transparently until enough data is gathered for statistical analysis or machine learning (i.e. training or inference).

Executing multiple versions in a single program run has several advantages over traditional approaches which execute versions in separate runs. In the context of dynamic compilation, compiled functions depend on profiling information, timing and memory conditions. These dependencies and the whole compilation history are automatically taken into account in our approach and thus provide more comparable results compared to iterative compilation. Additionally, executing multiple versions in a single run—preferably alternating—reduces requirements regarding CPU stability and varying background tasks on the execution environment. Lastly, no additional post-processing of data from multiple runs is necessary; one program run suffices to produce consistent performance data. Compilation forking is neither restricted to being used with particular benchmarks, nor to mere execution runs for data generation.

In the following, we present a step-by-step overview of the compiler-agnostic compilation forking approach. As a running example, forking is applied in the *loop peeling* phase (c.f. Section 5.1) for function `foo`, shown in Figure 2. More complex steps and implementation details are explained in respective subsections of Section 4.

Forking Point Initially, an entry point for compilation forking has to be defined. This can be before any phase in the compilation pipeline where a compilation decision is to be made and its impact needs to be analyzed. Additionally, multiple forking points can be defined, resulting in a nested forking scheme. However, nested forking should

be applied with caution, i.e., only for strongly related optimizations to evaluate their interplay. Otherwise, it could lead to the initial problem of not being able to attribute performance changes to particular compilation parameters. A method is processed by all compiler phases preceding the phase at the next forking point, i.e. the forked phase, resulting in a so-called intermediate compilation.

Listing 1 Fork without peeling.

```
1 void foo_0(int n) {
2     int i = n
3     while(i < GLOBAL) {
4         // loop body
5         i++
6     }
7 }
```

Listing 2 Fork with peeling.

```
1  void foo_1(int n) {
2      int i = n
3      if(i < GLOBAL) {
4          // loop body
5          i++
6      }
7      while(i < GLOBAL) {
8          // loop body
9          i++
10     }
11 }
```

Listing 3 Recombined forks.

```
1  void foo(int n) {
2      switch(forkControl % nrForks) {
3          case 0:
4              int i = n
5              while(i < GLOBAL) {
6                  // loop body
7                  i++
8              }
9              break
10         case 1:
11             int i = n
12             if(i < GLOBAL) {
13                 // loop body
14                 i++
15             }
16             while(i < GLOBAL) {
17                 // loop body
18                 i++
19             }
20             break
21     }
22     forkControl++
23 }
```

Fork Creation At a forking point, the intermediate compilation is duplicated n times, with n being the number of compilation parameter values to be explored. In general, compilation parameter values can be either boolean, multi-class nominal or metric. Thus, the state space needs to be reduced to a manageable size. As loop peeling is a boolean decision (peel or not peel), we have to copy function `foo` only once to cover both scenarios for its `while` loop. In Section 5.3 we discuss how we used profiling information to further reduce the state space. After duplication, the forked phase is applied to each copy with the compilation parameter(s) enumerating the set of values to explore. This creates n versions of the intermediate compilation, which only differ in the parameter value for the current phase, but share the same past. For function `foo`, two forks—`foo_0` and `foo_1`—are created, which are shown in Listing 1 and Listing 2. In reality, intermediate compilations would be represented in a compiler-specific intermediate representation and not in source code.

Feature Extraction We use the term features as set of all *relevant* information we have about a compilation at a particular point in the pipeline. This includes both information on the compilation unit and on the optimization parameter values chosen for a fork. In a machine learning context, features are the input to a model, which produces a target value as its prediction. Feature extraction has to happen immediately before an optimization is applied.

Compilation Pipeline After forking, each fork is run through the remaining compilation pipeline. Parameter values chosen in the forked phase, might have large

impacts on subsequent phases. Also factors such as timing or memory usage might have an impact on the remaining phases, leading to noise in the generated data. Thus, we would interfere with the nature of a dynamic compiler if we set the remaining compilation pipeline to a fixed state.

Success Metric Instrumentation After compilation of the forked function has finished, success metrics for all versions need to be extracted for evaluation. Success metrics can either be the more prevalent execution time, but also memory usage or code size. For extracting run-time metrics, we instrument the compiled function to provide the success metrics during execution. A detailed description of how we extract execution time is given in Section 4.1.

Fork Recombination Eventually, all compiled forks are recombined to a self-contained function. It mimics the initially forked function by transparently dispatching to one of its versions during execution. For the running example, this is schematically depicted in Listing 3 on source code level, omitting any instrumentation for success metric or feature extraction. There are several advantages of recombining forks into a single function rather than having multiple functions at hand. Overall code size and call overhead are reduced and no undue patches to the deoptimizer and method call logic have to be made. The code size for forked functions might increase a lot, but as the control flows of different forks never merge again, no additional pressure is put on register allocation, which can use the same registers for all forks. A detailed depiction of the fork recombination process for compiler graphs is presented in Section 4.2.

Fork Execution The execution of forks in the recombined function can follow any kind of order, depending on the instrumentation parameters. In our reference implementation, the forks are executed alternately, as it is indicated in Listing 3. In contrast to our approach, Sanchez et al. [41] compile a new version of a function after enough (sequential) measurements have been taken. By referring to the work of Lau et al. [26], we believe that our approach will better average out CPU frequency jitter and impacts of varying parameters. Garbage collection (GC) caused by one fork execution might impact subsequent fork executions. However, we decided to exclude GC time at safepoints in our timestamp instrumentation, as GC is hard to be attributed to certain functions. After execution the information triplet of features, success metric and compilation decision can be used for training a machine learning model or to find the *best* optimization decision ad-hoc. Using deoptimization, this *best* version can then replace the instrumented, multi-version function initially created by our forking approach.

Requirements & Limitations The concept of compilation forking can be implemented in any compiler where optimizations are applied in a deterministic order. However, it is advisable that inlining and other optimizations which work across function boundaries precede the first forking point. Otherwise, the extracted success metric might be polluted by inlinees. For example, using method self time as presented in Section 4.1 would only work for optimizations after inlining. The types of possible optimizations comprise all those which can be decided based on a set of features observed at the point of an optimization. However, feature extraction and the forking point have to be defined manually for new optimizations.

Compilation forking is not a holistic approach, but rather allows for analyzing the impact of single or few optimization decisions, where an interplay is expected. For more holistic approaches we refer to recent related work tackling phase ordering [23] or skipping [24].

4 Implementation

We implemented compilation forking in the GraalVM [51] compiler, a highly optimizing dynamic compiler which is used in production on millions of devices. Our implementation is independent from source code as it directly uses Graal's graph-based intermediate program representation (IR) [15, 17]. The Graal IR consists of two directed graphs, one for control flow and the other for data flow. Nodes in the graph can either have a fixed position in the control flow (fixed nodes) or can be executed anywhere as long as data dependencies are met (floating nodes). By directly instrumenting Graal IR, we can profit from the polyglot features provided by GraalVM and its language implementation framework Truffle [50]. This enables extracting performance and feature data for any Truffle language, which facilitates training of language-specific machine learning models without additional effort. We now present more detailed implementation insights.

4.1 Timestamp Extraction

For most compiler optimizations, the impact on the execution time of the compiled function is of highest importance. In this section, we show our instrumentation for extracting *self time* of methods or code snippets. Self time is the time spent executing a method excluding time spent "outside" in calls or at safepoints[1] [28]. The basic idea behind this instrumentation is shown as pseudo code in Listing 4 and Listing 5. Therein, the self time of the current execution is first aggregated locally by excluding calls (or safepoints). At the end, the current execution time is aggregated to the global, thread safe storage.

Listing 4 Base function.

```
int bar(int n) {

    int m = n + 1


    int res = foo(n, m)

    res = res * 2



    return res
}
```

Listing 5 Pseudo instrumentation.

```
int bar(int n) {
    long t = 0, ts1 = getTime()
    int m = n + 1
    long ts2 = getTime()
    t += ts2 - ts1
    int res = foo(n, m)
    long ts3 = getTime()
    res = res * 2
    long ts4 = getTime()
    t += ts4 - ts3
    aggregate("bar", t)
    return res
}
```

[1] Points in the program execution where the GC can be safely run. See https://openjdk.java.net/groups/hotspot/docs/HotSpotGlossary.html, last accessed on 27 May 2022

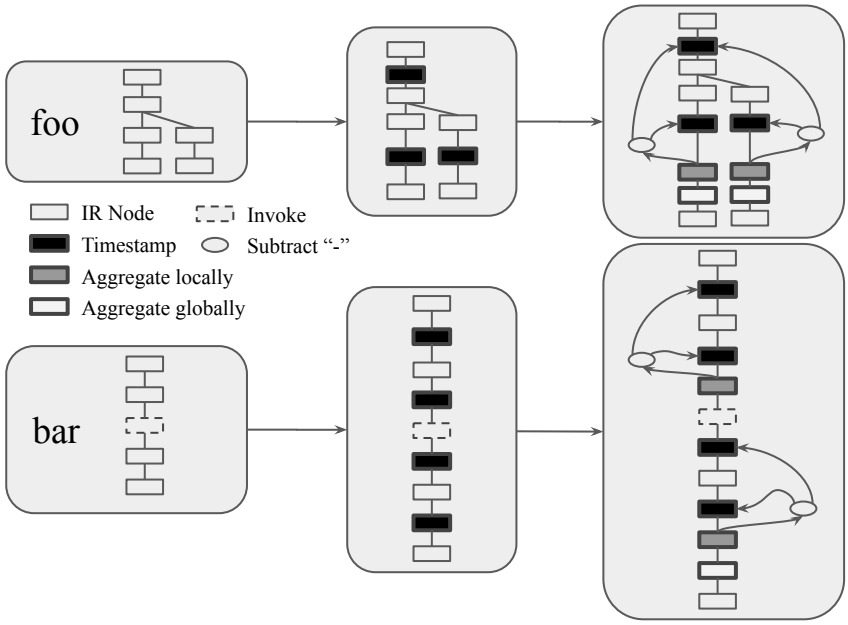


Figure 3 Timestamp instrumentation.

The actual instrumentation is added in the graph-based compiler IR, shown in Figure 3. Rectangular nodes are fixed nodes, in the sense that their order is preserved in a scheduling process. Circular nodes are nodes for subtraction, which can be executed out of order if data dependencies are met. Timestamps (black nodes) are added after each method entry (=first) node and before each method exit (=last) node, which is shown in the first instrumentation step in Figure 3 (foo). The difference between these timestamps would correspond to the *total* execution time of the method. To extract *self* time, we add additional timestamps before and after invocations of other methods and safepoints, which can be seen in Figure 3 (bar). While the instrumentation for measuring self time is more complex, we want to stress the importance of using it instead of total time. Otherwise, optimizations in callees would impact the *measured* performance of the caller. This could lead to an optimization being falsely identified as beneficial or harmful for the caller features. In case of across-function optimizations, like inlining, we can switch to total time to capture the time spent for a whole call.

Each time snippet is calculated by subtracting the start timestamp from the corresponding end timestamps (circular nodes). These measurements are summed up locally (dark grey nodes) for calculating the current method self time. Eventually, before each control flow sink, e.g. return or exception, the current execution time is added to the aggregated self time for this fork. This is captured in the white nodes which also handle the necessary synchronization of that operation. Storing all execution timestamps for a fork at run time would be infeasible with millions of invocations of each fork. However, by storing the aggregated run time of a fork together with its invocation count, we can calculate an average execution time for each fork, similar to [26]. We assume that—enough invocations provided—different method execution times due to different values of parameters or globals will average out as proven in [26]. The rationale behind this is the fact, that an optimization can only be considered beneficial, if the *average* execution time of the optimized code is improved.

Timestamps The timestamp values are extracted using Intel's `rdtscp`[2] instruction. This instruction ensures that preceding instructions are executed before acquiring a timestamp. Additionally, we emit `lfence` instructions after start timestamps and before end timestamps as shown in Figure 4. `lfence` instructions ensure that instructions scheduled after them will not be executed out-of-order before the `lfence` instruction is completed. Using the setup as shown in Figure 4 excludes the time for executing the

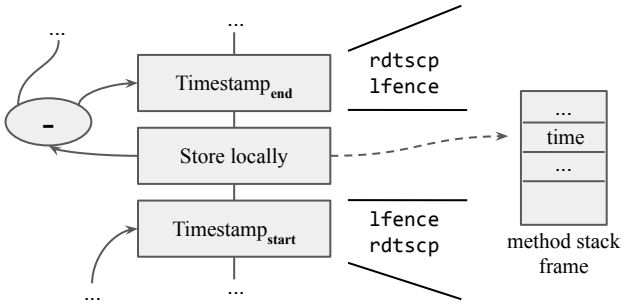


Figure 4 RDTSCP and memory fences.

instrumentation logic (e.g. when updating the aggregated local time). Thus, only the end timestamp call "pollutes" our measurement, which is the minimum unavoidable noise when timestamping. Nevertheless, the usage of `lfence` instructions will impact small or empty measurement regions a lot. In addition, if the performance of different forks is vastly affected by limiting out-of-order executions via `lfence` instructions, measurement noise might occur. Section 6.1 shows the overhead of the timestamp instrumentation for different types of functions. We use a data filtering to omit these data points from the result set.

Local Storage For the timestamp instrumentation a local slot in the method's stack frame is allocated via instrumentation to track self time across calls and safepoints. This ensures a thread-safe and recursion-supporting self time measurement. A measurement section can either end because of a control flow sink, e.g. return or exception, or an excluded operation such as a method call. Then, the method's stack slot value is updated by adding the difference of the timestamps from the current region.

Global Storage The global storage holds the aggregated time of all method executions, which are again subdivided into all forks. It resides in the compiler and is updated once before a control flow sink is encountered. The update is managed via GraalVM's `AtomicReadAndAdd` operation, which maps to an `xadd` on Intel and ensures synchronization.

Outlier Handling During feature extraction, we extract both static information and profiling information at the time of compilation. Other dynamic information such as

[2] https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf, last accessed on 27 May 2022

values of parameters or globals are not extracted in our approach. However, we argue that performance differences caused by different values of parameters or globals will cancel out in the long run [26]. However, noise from operating system (OS) interference still might occur and is hard to avoid in a real-world environment. The potential sources for OS noise include scheduling, context switches, memory usage and caching. We empirically checked that this noise is not introduced by our approach or instrumentation, by experimenting with native C programs, which exhibited similar OS noise. This experiment confirmed that while most execution times are stable, outliers from OS noise can exceed the average execution time by orders of magnitude. This can especially distort results for short-running methods. To counter OS outliers, we implemented an on-the-fly outlier removal as a transparent, yet aggressive addition to our timestamp instrumentation. It compares each locally aggregated execution time to the global average execution time for the fork. Depending on an outlier threshold local times may be omitted from being added to the global counter. Consistently, the invocation counter is not increased if an outlier is detected. With this measure, outliers accounting for sometimes 10% of the total execution time could be filtered out. However, we have to point out that it is impossible to distinguish between outliers caused by OS noise and outliers from extreme usage patterns. An single invocation of a fork with a exceptionally large parameter value would therefore likely be classified as an outlier. We are also experimenting with the use of a real-time OS to investigate the outlier behavior in a fully controlled environment, which might be put to use in the future.

4.2 Fork Recombination

Fork recombination happens at the very end of the compilation pipeline. All compiler

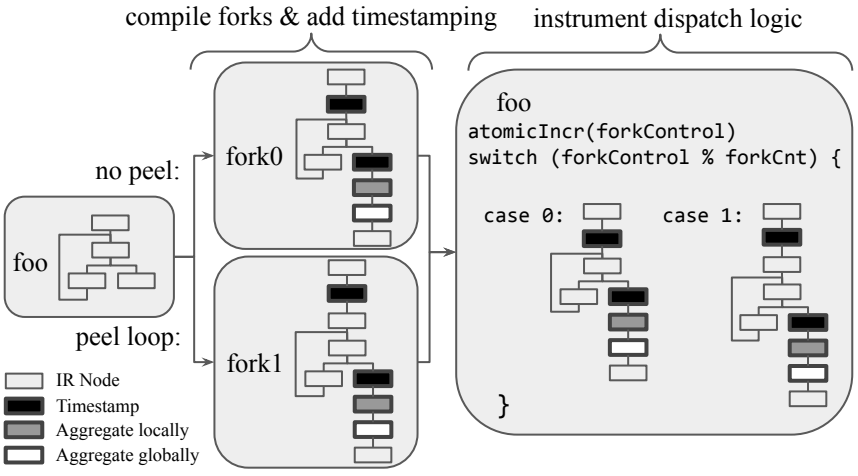


Figure 5 Fork recombination.

graphs originating from one or multiple forking events are merged into a combined graph resembling a pseudo-function. Figure 5 depicts the recombination process for a function where forking has been applied in the loop peeling phase. Initially, each fork is compiled on its own, including timestamp instrumentation or any other

success metric measurement. Eventually, the forks are recombined by copying their graphs into different branches of a switch. This switch—shown in source code for simplicity—controls the fork execution. In Figure 5 an alternating execution of the two forks is enforced. The *forkControl* variable is stored in the compiler, as discussed in Section 4.3. During code generation, it has to be ensured that all switch-cases are aligned identically. Otherwise, varying alignment can cause reproducible performance differences.

4.3 Data Structures

To avoid pressure on the garbage collector, which could impact the program execution, we store dynamically extracted performance information in a pre-allocated array of type long. This array is static, resides in the compiler and is persisted at the time of success metric collection or at program termination, at the latest. The storage format is shown in Figure 6. *C1* denotes the first forked compilation unit, with *C1F1* to *C1Fn*

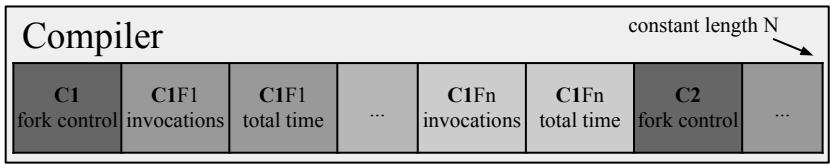


Figure 6 Storage format for performance data.

summarizing all its forks. The first field for each compilation unit is the *fork control*, which is used to choose the next fork to be executed (see Section 4.2). Right after the *fork control*, we store for each fork the number of invocations and the total execution time. All array indices are compiled as constants in the instrumentation code.

5 Case Studies: Loop Optimizations

In this section, we show case studies on how compilation forking can be used as a flexible data generation framework for machine learning in compiler optimizations.

5.1 Optimizations

Loop Peeling Loop peeling [3] is a transformation which moves a certain number of loop iterations in front of or behind the loop by copying the loop body. An example can be seen in Listing 1 and Listing 2. Peeling can eliminate null-checks within a loop, which are then performed only once outside the loop. In the GraalVM compiler, only the first iteration may be peeled. While the impact of loop peeling is generally considered low, our research with compilation forking showed that interference with other loop optimizations, especially vectorization, can have significant impacts which can be seen in Figure 1.

Table 1 Loop feature overview.

Category	Features
General	size; depth; isNested; #children; #backedges; #exits; isVectorizable
Execution	frequency; has[Exact/Max]TripCount; canOverflow
Nodes	#fixedNodes; #floatingNodes; #[IRNodeType]
Edges	#[EdgeType]IntoLoop; #[EdgeType]InLoop; #[EdgeType]OutOfLoop
Operands	#[object/int/float]Stamps; #[volatile/static]FieldAccesses
Parent	hasParent; parentSize
Graph	size; #loops; maxLoopDepth; #branches; #[IRNodeType]

Loop Unrolling Partial loop unrolling [3] is an optimization where the loop body is duplicated a certain number of times within the loop. Accordingly, the loop stride and the loop condition are adapted to fit the enlarged loop body and thus a smaller number of iterations. Loop unrolling can reduce the overhead of loop condition checks and can produce larger basic blocks, enabling more optimizations. The number of duplications is called *unroll factor*, the choice of which is the key of this optimization.

5.2 Features

In our case study, we focus on two loop-related optimizations — peeling and partial unrolling — introduced in Section 5.1. These optimizations use features that describe the loop to be optimized. We decided to base our features on the Graal intermediate program representation (IR) rather than on source code. Therefore, we can profit from the GraalVM's language implementation framework Truffle [50] which enables executing Java, JavaScript, Python or LLVM [40] programs. Also other approaches [6, 36] have used IRs—mostly LLVM IR—to support multiple source languages. Table 1 gives an overview of the features we extracted for loop-related optimizations. Other optimizations, for example duplication [30], would need a different set of features to describe the program parts to be optimized. The total number of potential features is approximately 1000 before applying any filters. This large number results from the fact that we use the node counts of the different IR node types (of which there are more than 450) in the loop and the enclosing graph as features.

5.3 Data Generation

When using compilation forking for loop-related optimizations, we work with the assumption that loops have no impact on each other when being optimized. This means that the speedup/slowdown resulting from transforming a loop l_1 is independent of a preceding or succeeding transformation of loop l_2. While this is a bold assumption and might not hold in some cases, interference should be low for non-nested loops. Our assumption results in a linear dependency between state space size and number of loops per function. Currently, we reduce the number of forks by selecting the most frequently executed loops as targets for creating forks. This information is provided as part of the profiling information provided by GraalVM [16]. Altogether, one fork for

each loop and for each optimization parameter value is created, along with a baseline fork without any optimized loops. For loop peeling, in each fork exactly one loop is peeled. For loop unrolling, each fork has exactly one loop unrolled with one specific unroll factor of 2, 4, 8, 16 or 32. The differences between the average execution times of fork and baseline can be considered as the success metric for the optimization. We used state-of-the-art industry and research benchmark suites such as *DaCapo* [4], *DaCapo Scala* [42], *Renaissance* [39] and *Octane*[3] for generating data, as well as a micro-benchmark suite including more recent JVM features such as lambdas and streams. To minimize noise, we disabled CPU frequency scaling, hyperthreading and vendor-specific features which impact performance.

5.4 Data Preprocessing

The implemented outlier removal handles large outliers. Nevertheless, small outliers still result in noise in the performance measurements. Thus, we apply filters to increase the consistency of the training dataset and consequently the trained model. Simple filters reduce noise susceptibility by removing compilations where forks either do not exceed a minimum number of invocations or a minimum average execution time. We experimented with filtering out data with very small speedups or slowdowns which may likely be overshadowed by noise. Additionally, small speedups or slowdowns indicate that the optimization decision is of not much importance, as the (performance) outcome is similar in any case. By removing such data, the trained model will be forced to focus on learning the more important decisions, where performance impacts are of higher significance. Apart from success metric filters, we also applied filters to reduce the feature space, i.e., the number of inputs to the model. There are many features with little information. A feature is deemed more informative the more different feature values are found throughout the dataset. Many IR nodes appear very rarely or not at all in whole benchmark suites leading to many IR node count features being zero for most data points. We conduct a sparsity check to remove such features, which allows us to heavily reduce the feature space. For loop peeling we reduced the number of features for model training to 299 and for unrolling to 257. All features are standardized before use by subtracting their mean and dividing by their standard deviation. The number of raw and filtered data points for each benchmark suite can be found in Table 2.

5.5 Model Training

We evaluated a multitude of models with different filtering and hyperparameter setups by using state-of-the-art machine learning frameworks for Python: PyTorch [37] for neural networks, scikit-learn [38] for dimensionality reduction, random forests (RF) and support vector machines (SVM) as well as the XGBoost [8] framework for gradient boosting. However, shallow models such as logistic regression, decision trees, SVMs

[3] https://github.com/chromium/octane, last accessed on 27 May 2022

Table 2 Data overview.

	peeling			unrolling		
Suite	**raw**	**filtered**	**[%]**	**raw**	**filtered**	**[%]**
DaCapo	28928	23697	81,9	2901	2083	71,8
DaCapo Scala	47138	40455	85,8	4832	3770	78,0
Renaissance	128541	109945	85,5	11009	8154	74,1
Micros	243434	207200	85,1	30359	22598	74,4
Octane	14079	12949	92,0	707	508	71,9

and RFs were insufficient in terms of prediction accuracy. This led us to different kinds of fully-connected neural networks (FCNNs). For the network layout, we tried small dense networks with up to 10 layers and a few million parameters to deeper residual networks with up to tens of millions of parameters. The latter have been shown to produce good results for tabular data [20]. Such residual networks with ten residual blocks and full pre-activation as described by Kaiming He et al. [31] produced the best results for peeling and unrolling. As optimizer we employed either plain Adam or AdamW, providing decoupled weight decay [22, 25]. We decided to train classification models in case of binary decisions such as peeling and a regressor to predict the speedups of different unroll factors. As loss function we used binary cross entropy (BCE) for classification and mean squared error (MSE) for regression. We scaled these losses by a function of the absolute logarithmic speedup and the aggregated execution time to give more importance to data points with a higher expected absolute speedup that have more impact on the overall benchmark time. In the models we use regularization layers and dropout to counter overfitting.

We used a cross-validation approach, where we randomly divided benchmarks into groups of five. A model is trained on all data except the five benchmarks from the corresponding evaluation group. Following this approach, we trained 28 models with identical hyperparameters for peeling and unrolling each. Therefore, each model is evaluated on truly unseen data in Section 6.2. The networks were trained for 2000 to 3000 epochs with a declining learning rate every 400 epochs. While training, we used a train-validate-split (90% train, 10% validate) to get insight into the training progress. The two resulting models are summarized in Figure 7.

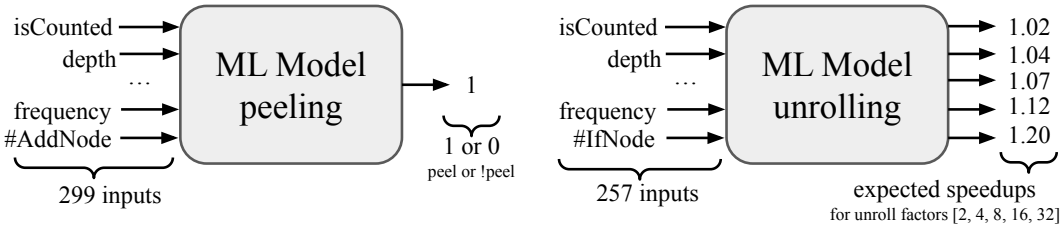


Figure 7 ML models for peeling and unrolling.

We also experimented with overfitting on single benchmarks, where we used gradient boosting due to its faster training. For this, we used XGBoost and allowed for an ample number of estimators (500), which led to an overfitting of up to 98 percent.

5.6 Model Evaluation

Evaluating multiple models in the compiler on all benchmarks is a time-consuming task. However, standard metrics such as accuracy, precision, recall or F1-score do not reflect the quality of the estimator in terms of total performance. For quickly selecting which models to test in the compiler, we estimated the performance impact on benchmark level by employing a custom heuristic.

Based on compilation forking data, this heuristic estimates the execution time $\widetilde{t}_m$ of a method m with predicted optimization parameters: We calculate the impacts of each compilation decision d in m compared to the baseline. Loop related optimizations yield one decision per loop and each decision can be represented by multiple forks. For example, in forked loop unrolling each unroll factor is mapped to one fork per loop. We select for each compilation decision d the fork where the parameter p under investigation matches the predicted value. Then, we extract the average execution time for the predicted decision $\overline{t}_{d}{}^{p}$ and calculate the difference to the baseline execution time $\overline{t}_b$. This difference denotes the expected speedup or slowdown for an optimization decision. The total execution time impact results from summing up the per-decision impacts. This total execution time impact is added to the baseline average execution time $\overline{t}_b$ and scaled by the total invocations i of the method, resulting in the expected absolute execution time $\widetilde{t}_m$. This is summarized in Equation (1).

$$\widetilde{t}_m = i\left(\left[\sum_d \overline{t}_{d}{}^{p} - \overline{t}_b\right] + \overline{t}_b\right) \tag{1}$$

$\widetilde{t}_m$: estimated average execution time of m
i : total invocations of method m
d : optimization decision
$\overline{t}_{d}{}^{p}$: Average execution time with parameter p in decision d
$\overline{t}_b$: Baseline of the method

Finally, we sum up all method execution times to estimate the per benchmark execution time when using a learned model. When exporting the default compiler decisions during forking, the same heuristic can be used to calculate the expected execution time in the current compiler. Additionally, an estimate of the best possible execution time can be made by using the minimum average execution time for each decision d. However, the heuristic relies on the assumption that decisions in the same method do not influence each other. It should therefore be taken as an estimator only.

6 Evaluation

We evaluated our approach twofold. First, we show the performance and code size impacts of compilation forking itself. Second, we show its applicability, by evaluating machine learning models, which are trained using the generated data.

```
int fib(int n) {
    if (n == 1 || n == 2) {
        return 1;
    } else {
        return fib(n-1) + fib(n-2);
    }
}
```

```
int sum(int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += i;
    }
    return sum;
}
```

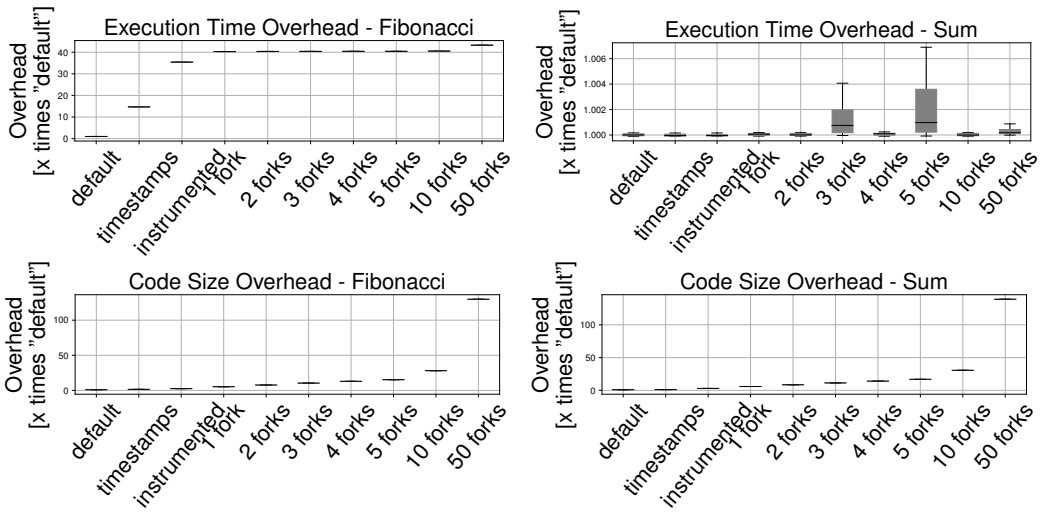


Figure 8 Performance and code size impact of forking. Lower is better.

6.1 Compilation Forking

As explained previously, compilation forking allows us to compare different optimizations based on the same compilation history. The performance of optimized parts is measured by instrumentation, which is excluded from time measurements. In the presented approach, the performance data is generated and processed offline, either by compiler experts analyzing compiler performance or when training machine learning models. Therefore, we consider compilation forking to be a non-performance-critical mode, where impacts on total execution time and code size are negligible. Nevertheless, we want to show how the impact on those metrics can vary depending on the compiled code and the number of forks. Note, that we now evaluate the *overall* performance *including* the instrumentation overhead; the *measured and extracted* performance numbers are not impacted by the instrumentation overhead.

Figure 8 shows two functions and the overhead in terms of execution time and code size. The following configurations were tested, which build on top of each other: *default*: GraalVM without any changes; *timestamping*: timestamp measurement; *instrumentation*: outlier removal and invocation counting; *forks*: a number of identical forks created from the original function. The first function recursively calculates the n-th Fibonacci number. A single invocation is executed very fast, but the recursive nature leads to many calls. The second function calculates a sum in a loop. An invocation has significantly higher workload, depending on n. Each measurement was executed 50 times with 10000 calls of the function.

Performance Figure 8 shows that most overhead is introduced by timestamping and instrumentation. The relative overhead of timestamping becomes more costly the smaller the function is and the more calls have to be excluded in the self time instrumentation. Thus, a slowdown factor of 16 is encountered for function *fib*, but no overhead for function *sum* with its long running loop. The instrumentation overhead for outlier removal and invocation counting happens only once per method and adds a constant overhead. Forking introduces a constant slowdown for the dispatch logic, which is relatively larger in smaller functions. This has also been encountered in related work [18] and is the reason why smaller functions are removed from the data set. Creating multiple forks does not impact the execution time. Only in the configuration with 50 forks a small slowdown is measured for both examples. We assume that this is because of the immensely increased code size which might impact caching.

Code Size The impact on code size depends on the number of inserted timestamps and the number of instrumented control flow sinks. Regarding the number of forks, it follows a linear pattern, as can be seen in Figure 8.

6.2 Learned Compiler Optimizations

We replaced the hand-crafted heuristics in the GraalVM compiler—currently one of the highest-optimizing Java compilers[4]—with our trained models to investigate two hypotheses: First, that compilation forking does not distort the program execution and that it yields consistent performance results. Second, to investigate how data-driven optimization approaches perform against hand-crafted heuristics with years of fine-tuning and compiler expertise.

Our evaluation compares the GraalVM compiler with a compiler version using the learned predictors. Each evaluation replaces only one loop optimization heuristic by a learned model to ensure better comparability. We measured run time, code size and compile time with the former being of most interest as it has been used as a success metric in model training.

Each optimization is evaluated on five benchmark suites, introduced in Section 5.3, executed for at least ten times per configuration. To ensure a separation of training and test data, we deployed multiple models, where each model is tested on five benchmarks not used in its training. Due to space limitations, we summarize the performance results per benchmark suite for peeling in Table 3 and for unrolling in Table 4. The tables contain the number of benchmarks in each suite, the number of benchmarks where a speedup or a slowdown has been encountered, along with a maximum speedup and slowdown. For calculating speedups and slowdowns, we aggregate benchmark runs by calculating geometric means. To ensure statistical significance, we present the number of significantly faster or slower benchmarks per suite (#sig) using a Wilcoxon signed-rank test [48] for unpaired data. We avoided a

[4] https://renaissance.dev/, last accessed on 27 May 2022

standard t-test as we cannot ensure that our data is normally distributed due to noise in the executions.

For most learned models we can see comparable or slightly worse performance than for the highly-tuned GraalVM heuristics in terms of geometric means for the whole benchmark suite run time. The learned peeling strategy summarized in Table 3 seems to peel more often, as the overall code size and compile time are increased. In single benchmarks, both peeling an unrolling achieve speedups of up to 20% but also slowdowns of up to 17% .

Table 3 Loop peeling evaluation. Lower is better for geometric means comparison.

Suite		Speedup			Slowdown			Geometric Means ML vs. Heuristics		
Name	#	#	#sig	max%	#	#sig	max%	RunTime	CodeSize	CompTime
DaCapo	9	2	1	0.698	7	5	6.165	1.020	1.203	1.336
DaCapo Scala	12	1	0	0	11	8	8.023	1.018	1.138	1.207
Renaissance	25	9	1	8.381	16	8	10.231	1.003	1.213	1.295
Micros	74	38	17	20.104	36	16	15.747	0.997	1.182	1.222
Octane	14	5	3	5.668	9	5	2.439	0.998	1.178	1.283

Table 4 Loop unrolling evaluation. Lower is better for geometric means comparison.

Suite		Speedup			Slowdown			Geometric Means ML vs. Heuristics		
Name	#	#	#sig	max%	#	#sig	max%	RunTime	CodeSize	CompTime
DaCapo	9	1	0	0	8	4	17.637	1.039	1.071	1.150
DaCapo Scala	12	5	2	4.467	7	3	5.890	1.003	1.036	1.078
Renaissance	25	9	1	18.337	16	4	11.598	1.008	1.023	1.081
Micros	74	33	3	9.718	41	6	8.463	1.001	1.061	1.104
Octane	14	4	1	0.765	10	6	5.940	1.011	1.071	1.138

We also analyzed the potential gain of optimizations by explicitly overfitting models on single benchmarks. Detailed results for loop peeling in the Octane benchmark suite are shown in Figure 9. It indicates that for many benchmarks, significant speedups

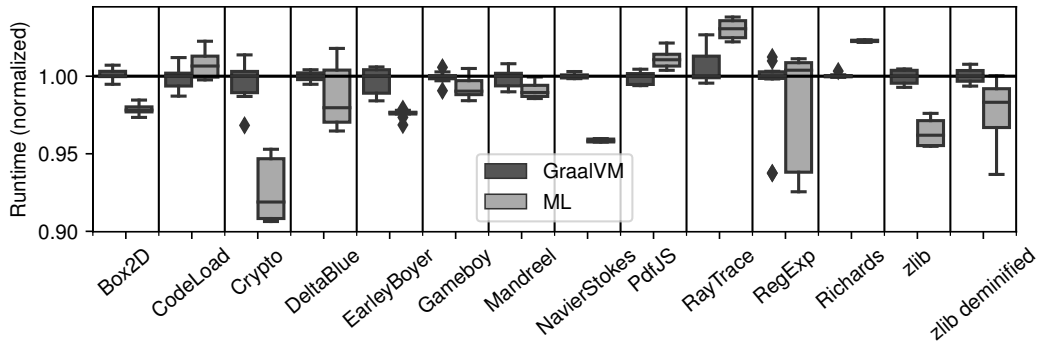


Figure 9 Octane peeling (overfitted). Lower is better.

can be achieved by optimizing loop peeling, which is often considered less important. However, benchmarks like Richards or Raytrace show slowdowns compared to the default GraalVM which indicates that the data is not clearly distinguishable using our features. This was already foreshadowed in the training process, where accuracy for Richards only converged at 73%.

Taking into account that the GraalVM heuristics are tuned towards these benchmarks, we argue that the performance of the trained models supports our major claim: High-quality performance data can be generated using compilation forking, which can facilitate creating machine learning models that match existing compiler heuristics. Additionally, we found flaws in the GraalVM heuristics for several benchmarks and could point compiler experts to them.

7 Conclusion

In this paper, we presented compilation forking - a method which brings back iterative compilation into modern times for dynamic compilers. It allows for comparing different optimization decisions in dynamic compilers based on a common compilation history for arbitrary programs. Instead of re-compiling individual functions and re-starting the surrounding benchmark program, different versions of functions are compiled and executed all in one run. We handle uncertainties in the dynamic compilation pipeline, by forking a compilation at a point of interest. We execute different forks alternatingly, to create a statistically representative average over function calls with different values of parameters or globals. Thus, we claim to be able to assess the impact of single compilation decisions accurately. Compilation forking itself is compiler-agnostic. However, our implementation within the GraalVM compiler allows for a novel level of programming-language-agnostic applicability. We use its graph-based compiler-internal program representation for extracting program features. This enables generating data for any language, supported by GraalVM's Truffle language implementation framework.

To verify the quality of our generated data, we trained several machine learning models for replacing compiler heuristics for loop peeling and unrolling. Achieving similar performance to one of the fastest JVMs supports our claim that compilation forking indeed can be used to extract quality performance data and to consistently compare different optimization decisions in a dynamic compiler Speedups of up to 20% for single benchmarks unveiled flaws in GraalVM's heuristics for particular code patterns. In future work we will shift the focus on improving our machine learning models further, by employing techniques such as graph neural networks to better capture graph-based IRs. Furthermore, compilation forking itself provides opportunities for further research: We plan to investigate the interplay of optimizations by employing nested forking. Besides that, deoptimization enables us to connect data generation by compilation forking with using learned or updated ML models in single runs.

Acknowledgements This research project is partially funded by Oracle Labs.

References

[1] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. "OpenTuner: An extensible framework for program autotuning". In: *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*. IEEE Computer Society, 2014, pages 303–315. DOI: 10.1145/2628071.2628092.

[2] Amir H. Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. "A Survey on Compiler Autotuning Using Machine Learning". In: *ACM Comput. Surv.* (2018), 96:1–96:42. ISSN: 0360-0300. DOI: 10.1145/3197978.

[3] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. "Compiler Transformations for High-Performance Computing". In: *ACM Comput. Surv.* (1994), pages 345–420. ISSN: 0360-0300. DOI: 10.1145/197405.197406.

[4] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. "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*. OOPSLA '06. Portland, Oregon, USA: Association for Computing Machinery, 2006, pages 169–190. ISBN: 1595933484. DOI: 10.1145/1167473.1167488.

[5] François Bodin, Toru Kisuki, Peter Knijnenburg, Mike OBoyle, and Erven Rohou. "Iterative compilation in a non-linear optimisation space". In: *Workshop on Profile and Feedback-Directed Compilation* (1998). URL: https://hal.inria.fr/inria-00475919/document (visited on 2022-05-27).

[6] Alexander Brauckmann, Andrés Goens, Sebastian Ertel, and Jeronimo Castrillon. "Compiler-Based Graph Representations for Deep Learning Models of Code". In: *Proceedings of the 29th International Conference on Compiler Construction*. CC 2020. San Diego, CA, USA: Association for Computing Machinery, 2020, pages 201–211. ISBN: 9781450371209. DOI: 10.1145/3377555.3377894.

[7] John Cavazos and Michael F. P. O'Boyle. "Automatic Tuning of Inlining Heuristics". In: *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*. SC '05. USA: IEEE Computer Society, 2005, page 14. ISBN: 1595930612. DOI: 10.1109/SC.2005.14.

[8] Tianqi Chen and Carlos Guestrin. "XGBoost: A Scalable Tree Boosting System". In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '16. San Francisco, California, USA: Association for Computing Machinery, 2016, pages 785–794. ISBN: 9781450342322. DOI: 10.1145/2939672.2939785.

[9] Peng-fei Chuang, Howard Chen, Gerolf F. Hoflehner, Daniel M. Lavery, and Wei-chung Hsu. "Dynamic profile driven code version selection". In: *the 11th Annual Workshop on the Interaction between Compilers and Computer Architecture*. 2007. URL: https://www.researchgate.net/publication/228952289_Dynamic_Profile_Driven_Code_Version_Selection (visited on 2022-05-27).

[10] Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. "Optimizing for Reduced Code Space Using Genetic Algorithms". In: *SIGPLAN Not.* (1999), pages 1–9. ISSN: 0362-1340. DOI: 10.1145/315253.314414.

[11] Keith D. Cooper, Devika Subramanian, and Linda Torczon. "Adaptive Optimizing Compilers for the 21st Century". In: *J. Supercomput.* (2002), pages 7–22. ISSN: 0920-8542. DOI: 10.1023/A:1015729001611.

[12] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. "End-to-End Deep Learning of Optimization Heuristics". In: *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2017, pages 219–232. DOI: 10.1109/PACT.2017.24.

[13] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. "Synthesizing Benchmarks for Predictive Modeling". In: *Proceedings of the 2017 International Symposium on Code Generation and Optimization*. CGO '17. Austin, USA: IEEE Press, 2017, pages 86–99. ISBN: 9781509049318. DOI: 10.1109/CGO.2017.7863731.

[14] Christophe Dubach, John Cavazos, Björn Franke, Grigori Fursin, Michael F.P. O'Boyle, and Olivier Temam. "Fast Compiler Optimisation Evaluation Using Code-Feature Based Performance Prediction". In: *Proceedings of the 4th International Conference on Computing Frontiers*. CF '07. Ischia, Italy: Association for Computing Machinery, 2007, pages 131–142. ISBN: 9781595936837. DOI: 10.1145/1242531.1242553.

[15] Gilles Duboscq, Lukas Stadler, Thomas Würthinger, Doug Simon, Christian Wimmer, and Hanspeter Mössenböck. "Graal IR: An Extensible Declarative Intermediate Representation". In: *Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop*. 2013, pages 1–9. URL: https://ssw.jku.at/General/Staff/GD/APPLC-2013-paper_12.pdf (visited on 2022-05-27).

[16] Gilles Duboscq, Thomas Würthinger, and Hanspeter Mössenböck. "Speculation without Regret: Reducing Deoptimization Meta-Data in the Graal Compiler". In: *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. PPPJ '14. Cracow, Poland: Association for Computing Machinery, 2014, pages 187–193. ISBN: 9781450329262. DOI: 10.1145/2647508.2647521.

[17] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. "An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler". In: *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages*. VMIL '13. Indianapolis, Indiana, USA: Association for Computing Machinery, 2013, pages 1–10. ISBN: 9781450326018. DOI: 10.1145/2542142.2542143.

[18] Grigori Fursin, Albert Cohen, Michael O'Boyle, and Olivier Temam. "A Practical Method for Quickly Evaluating Program Optimizations". In: *High Performance Embedded Architectures and Compilers*. Edited by Nacho Conte Tomband Navarro, Wen-mei W. Hwu, Mateo Valero, and Theo Ungerer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pages 29–46. ISBN: 978-3-540-32272-6. DOI: 10.1007/11587514_4.

[19] Grigori Fursin, Cupertino Miranda, Olivier Temam, Mircea Namolaru, Ayal Zaks, Bilha Mendelson, Edwin Bonilla, John Thomson, Hugh Leather, Chris Williams, Michael O'Boyle, Phil Barnard, Elton Ashton, Eric Courtois, and François Bodin. "MILEPOST GCC: machine learning based research compiler". In: *Proceedings of the GCC Developers' Summit 2008*. 2008. URL: https://hal.inria.fr/inria-00294704 (visited on 2022-05-27).

[20] Yury Gorishniy, Ivan Rubachev, Valentin Khrulkov, and Artem Babenko. "Revisiting Deep Learning Models for Tabular Data". In: *CoRR* (2021). arXiv: 2106.11959. URL: https://arxiv.org/abs/2106.11959 (visited on 2022-05-27).

[21] Ameer Haj-Ali, Nesreen K. Ahmed, Ted Willke, Yakun Sophia Shao, Krste Asanovic, and Ion Stoica. "NeuroVectorizer: End-to-End Vectorization with Deep Reinforcement Learning". In: *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*. CGO 2020. San Diego, CA, USA: Association for Computing Machinery, 2020, pages 242–255. ISBN: 9781450370479. DOI: 10.1145/3368826.3377928.

[22] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Identity Mappings in Deep Residual Networks". In: *Computer Vision – ECCV 2016*. Edited by Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling. Cham: Springer International Publishing, 2016, pages 630–645. ISBN: 978-3-319-46493-0. DOI: 10.1007/978-3-319-46493-0_38.

[23] Q. Huang, A. Haj-Ali, W. Moses, J. Xiang, I. Stoica, K. Asanovic, and J. Wawrzynek. "AutoPhase: Compiler Phase-Ordering for HLS with Deep Reinforcement Learning". In: *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. Los Alamitos, CA, USA: IEEE Computer Society, 2019, pages 308–308. DOI: 10.1109/FCCM.2019.00049.

[24] Tarindu Jayatilaka, Hideto Ueno, Giorgis Georgakoudis, EunJung Park, and Johannes Doerfert. "Towards Compile-Time-Reducing Compiler Optimization Selection via Machine Learning". In: *50th International Conference on Parallel Processing Workshop*. ICPP Workshops '21. Lemont, IL, USA: Association for Computing Machinery, 2021. ISBN: 9781450384414. DOI: 10.1145/3458744.3473355.

[25] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: arXiv, 2014. DOI: 10.48550/ARXIV.1412.6980. (Visited on 2022-05-27).

[26] Jeremy Lau, Matthew Arnold, Michael Hind, and Brad Calder. "Online Performance Auditing: Using Hot Optimizations without Getting Burned". In: *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '06. Ottawa, Ontario, Canada: Association

for Computing Machinery, 2006, pages 239–251. ISBN: 1595933204. DOI: 10.1145/1133981.1134010.

[27] Hugh Leather and Chris Cummins. "Machine Learning in Compilers: Past, Present and Future". In: *2020 Forum for Specification and Design Languages (FDL)*. IEEE Computer Society, 2020, pages 1–8. DOI: 10.1109/FDL50818.2020.9232934.

[28] David Leopoldseder, Roland Schatz, Lukas Stadler, Manuel Rigger, Thomas Würthinger, and Hanspeter Mössenböck. "Fast-Path Loop Unrolling of Non-Counted Loops to Enable Subsequent Compiler Optimizations". In: *Proceedings of the 15th International Conference on Managed Languages & Runtimes*. ManLang '18. Linz, Austria: Association for Computing Machinery, 2018. ISBN: 9781450364249. DOI: 10.1145/3237009.3237013.

[29] David Leopoldseder, Lukas Stadler, Thomas Würthinger, Josef Eisl, Doug Simon, and Hanspeter Mössenböck. "Dominance-Based Duplication Simulation (DBDS): Code Duplication to Enable Compiler Optimizations". In: *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. CGO 2018. Vienna, Austria: Association for Computing Machinery, 2018, pages 126–137. ISBN: 9781450356176. DOI: 10.1145/3168811.

[30] David Leopoldseder, Lukas Stadler, Thomas Würthinger, Josef Eisl, Doug Simon, and Hanspeter Mössenböck. "Dominance-based Duplication Simulation (DBDS): Code Duplication to Enable Compiler Optimizations". In: *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. CGO 2018. Vienna, Austria: Association for Computing Machinery, 2018, pages 126–137. ISBN: 978-1-4503-5617-6. DOI: 10.1145/3168811.

[31] Ilya Loshchilov and Frank Hutter. "Decoupled Weight Decay Regularization". In: arXiv, 2017. DOI: 10.48550/ARXIV.1711.05101. (Visited on 2022-05-27).

[32] Charith Mendis, Alex Renda, Dr.Saman Amarasinghe, and Michael Carbin. "Ithemal: Accurate, Portable and Fast Basic Block Throughput Estimation using Deep Neural Networks". In: *Proceedings of the 36th International Conference on Machine Learning*. Edited by Kamalika Chaudhuri and Ruslan Salakhutdinov. Proceedings of Machine Learning Research. PMLR, 2019, pages 4505–4515. URL: http://proceedings.mlr.press/v97/mendis19a.html (visited on 2022-05-27).

[33] Antoine Monsifrot, François Bodin, and Rene Quiniou. "A Machine Learning Approach to Automatic Production of Compiler Heuristics". In: *Proceedings of the 10th International Conference on Artificial Intelligence: Methodology, Systems, and Applications*. AIMSA '02. London, UK, UK: Springer-Verlag, 2002, pages 41–50. ISBN: 3-540-44127-1. URL: http://dl.acm.org/citation.cfm?id=646053.677574 (visited on 2022-05-27).

[34] Raphael Mosaner, David Leopoldseder, Lukas Stadler, and Hanspeter Mössenböck. "Using Machine Learning to Predict the Code Size Impact of Duplication Heuristics in a Dynamic Compiler". In: *Proceedings of the 18th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*.

MPLR '21. Association for Computing Machinery, 2021, pages 127–135. ISBN: 978-1-4503-8675-3/21/09. DOI: 10.1145/3475738.3480943.

[35] Paschalis Mpeis, Pavlos Petoumenos, Kim Hazelwood, and Hugh Leather. "Developer and User-Transparent Compiler Optimization for Interactive Applications". In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2021. Virtual, Canada: Association for Computing Machinery, 2021, pages 268–281. ISBN: 9781450383912. DOI: 10.1145/3453483.3454043.

[36] Eunjung Park, John Cavazos, and Marco A. Alvarez. "Using Graph-Based Program Characterization for Predictive Modeling". In: *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. CGO '12. San Jose, California: Association for Computing Machinery, 2012, pages 196–206. ISBN: 9781450312066. DOI: 10.1145/2259016.2259042.

[37] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. "Pytorch: An imperative style, high-performance deep learning library". In: *Advances in neural information processing systems* (2019), pages 8026–8037. URL: https://dl.acm.org/doi/10.5555/3454287.3455008 (visited on 2022-05-27).

[38] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Peter Prettenhofer, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Ron Weiss, Vincent Dubourg, et al. "Scikit-learn: Machine learning in Python". In: *the Journal of machine Learning research* (2011), pages 2825–2830. URL: https://www.jmlr.org/papers/volume12/pedregosa11a/pedregosa11a.pdf (visited on 2022-05-27).

[39] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. "Renaissance: Benchmarking Suite for Parallel Applications on the JVM". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pages 31–47. ISBN: 9781450367127. DOI: 10.1145/3314221.3314637.

[40] Manuel Rigger, Matthias Grimmer, and Hanspeter Mössenböck. "Sulong - Execution of LLVM-Based Languages on the JVM: Position Paper". In: *Proceedings of the 11th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*. ICOOOLPS '16. Rome, Italy: Association for Computing Machinery, 2016. ISBN: 9781450348379. DOI: 10.1145/3012408.3012416.

[41] Ricardo Nabinger Sanchez, Jose Nelson Amaral, Duane Szafron, Marius Pirvu, and Mark Stoodley. "Using Machines to Learn Method-Specific Compilation Strategies". In: *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. CGO '11. USA: IEEE Computer Society, 2011, pages 257–266. ISBN: 9781612843568. DOI: 10.1109/CGO.2011.5764693.

[42] Andreas Sewe, Mira Mezini, Aibek Sarimbekov, and Walter Binder. "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*. OOPSLA '11. Portland, Oregon, USA: Association for Computing Machinery, 2011, pages 657–676. ISBN: 9781450309400. DOI: 10.1145/2048066.2048118.

[43] Douglas Simon, John Cavazos, Christian Wimmer, and Sameer Kulkarni. "Automatic Construction of Inlining Heuristics Using Machine Learning". In: *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization*. CGO '13. Washington, DC, USA: IEEE Computer Society, 2013, pages 1–12. ISBN: 978-1-4673-5524-7. DOI: 10.1109/CGO.2013.6495004.

[44] M. Stephenson and S. Amarasinghe. "Predicting unroll factors using supervised classification". In: *International Symposium on Code Generation and Optimization*. IEEE Computer Society, 2005, pages 123–134. DOI: 10.1109/CGO.2005.29.

[45] Michele Tartara and Stefano Crespi Reghizzi. "Continuous Learning of Compiler Heuristics". In: *ACM Trans. Archit. Code Optim.* (2013). ISSN: 1544-3566. DOI: 10.1145/2400682.2400705.

[46] *V8 JavaScript Compiler*. 2021. URL: https://github.com/v8/v8 (visited on 2022-05-27).

[47] Zheng Wang and Michael O'Boyle. "Machine Learning in Compiler Optimization". In: *Proceedings of the IEEE* (2018), pages 1879–1901. ISSN: 0018-9219. DOI: 10.1109/JPROC.2018.2817118.

[48] Frank Wilcoxon. "Individual Comparisons by Ranking Methods". In: *Biometrics Bulletin* (1945), pages 80–83. ISSN: 00994987. URL: http://www.jstor.org/stable/3001968 (visited on 2022-05-27).

[49] Christian Wimmer, Vojin Jovanovic, Erik Eckstein, and Thomas Würthinger. "One Compiler: Deoptimization to Optimized Code". In: *Proceedings of the 26th International Conference on Compiler Construction*. CC 2017. Austin, TX, USA: Association for Computing Machinery, 2017, pages 55–64. ISBN: 9781450352338. DOI: 10.1145/3033019.3033025.

[50] Christian Wimmer and Thomas Würthinger. "Truffle: A Self-Optimizing Runtime System". In: *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*. SPLASH '12. Tucson, Arizona, USA: Association for Computing Machinery, 2012, pages 13–14. ISBN: 9781450315630. DOI: 10.1145/2384716.2384723.

[51] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. "One VM to Rule Them All". In: *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. Onward! 2013. Indianapolis, Indiana, USA: Association for Computing Machinery, 2013, pages 187–204. ISBN: 978-1-4503-2472-4. DOI: 10.1145/2509578.2509581.

[52] Mingzhou Zhou, Xipeng Shen, Yaoqing Gao, and Graham Yiu. “Space-Efficient Multi-Versioning for Input-Adaptive Feedback-Driven Program Optimizations”. In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*. OOPSLA ’14. Portland, Oregon, USA: Association for Computing Machinery, 2014, pages 763–776. ISBN: 9781450325851. DOI: 10.1145/2660193.2660229.

Chapter 7

Self-optimizing Heuristics

This chapter includes the follow-up paper to the previously introduced *compilation forking*. It presents, how compilation forking and de-optimization can be used to optimize machine-learning-based heuristics at user site.

Paper: Raphael Mosaner, David Leopoldseder, Wolfgang Kisling, Lukas Stadler, and Hanspeter Mössenböck. 2022. Machine-Learning-Based Self-Optimizing Compiler Heuristics. In *Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes* (MPLR 2022). Association for Computing Machinery, New York, NY, USA, 98–111. https://doi.org/10.1145/3546918.3546921

Machine-Learning-Based Self-Optimizing Compiler Heuristics*

Raphael Mosaner
raphael.mosaner@jku.at
Johannes Kepler University
Linz, Austria

David Leopoldseder
david.leopoldseder@oracle.com
Oracle Labs
Vienna, Austria

Wolfgang Kisling
wolfgang.kisling@jku.at
Johannes Kepler University
Linz, Austria

Lukas Stadler
lukas.stadler@oracle.com
Oracle Labs
Linz, Austria

Hanspeter Mössenböck
hanspeter.moessenboeck@jku.at
Johannes Kepler University
Linz, Austria

ABSTRACT

Compiler optimizations are often based on hand-crafted heuristics to guide the optimization process. These heuristics are designed to benefit the average program and are otherwise static or only customized by profiling information. We propose *machine-learning-based self-optimizing compiler heuristics*, a novel approach for fitting optimization decisions in a dynamic compiler to specific environments. This is done by updating a machine learning model with extracted performance data at run time. Related work—which primarily targets static compilers—has already shown that machine learning can outperform hand-crafted heuristics. Our approach is specifically designed for dynamic compilation and uses concepts such as deoptimization for transparently switching between generating data and performing machine learning decisions in single program runs. We implemented our approach in the GraalVM, a high-performance production VM for dynamic compilation. When evaluating our approach by replacing loop peeling heuristics with learned models we encountered speedups larger than 30% for several benchmarks and only few slowdowns of up to 7%.

CCS CONCEPTS

• **General and reference** → *Performance*; *Empirical studies*; • **Software and its engineering** → **Just-in-time compilers**; **Dynamic compilers**; • **Computing methodologies** → **Neural networks**.

KEYWORDS

Dynamic Compilation, Optimization, Heuristics, Loop Peeling, Performance, Machine Learning, Neural Networks

ACM Reference Format:
Raphael Mosaner, David Leopoldseder, Wolfgang Kisling, Lukas Stadler, and Hanspeter Mössenböck. 2022. Machine-Learning-Based Self-Optimizing Compiler Heuristics. In *Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes (MPLR '22), September 14–15, 2022, Brussels, Belgium.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3546918.3546921

*This research project is partially funded by Oracle Labs.

MPLR '22, September 14–15, 2022, Brussels, Belgium

1 INTRODUCTION

Dynamic compilation [3] has surpassed static compilation when it comes to aggressiveness of optimizations and input-specific compilation strategies. This is facilitated by profiling-based speculation [3, 14], where optimization decisions are based on profiling information which is gathered prior to compilation. For example, conditional branches can be omitted under the assumption that they are never entered, if this is indicated by the branch probabilities measured during profiling [27]. If assumptions are invalidated during execution, deoptimization [21, 44] followed by a re-compilation can produce a new compilation with updated assumptions.

The benefits of such data-driven approaches [27, 44, 46] are evident but they are still not widely used. State-of-the-art dynamic compilers [47] still rely heavily on human-crafted heuristics to guide the optimization process. These heuristics are based on many years of development effort and compiler expertise to perform well on the *average* program. Nevertheless, they mainly reflect the benchmarks which were used by compiler experts for performance optimizations and fine-tuning. Tailoring heuristics for specific workloads or hardware environments would be infeasible. Thus, profiling information is often the only knob in these otherwise static and one-size-fits-all heuristics.

Another challenge in compiler construction is that optimizations can have impacts on each other and the trade-offs to be made are not always clear. For example, a loop peeling transformation [4] can prevent loop vectorization [4] in a later optimization stage. Modeling such circumstances in heuristics is hard, and so are decisions which do not negatively affect more important optimizations later on. The unsolved phase-ordering problem [1, 24] indicates that optimal holistic compilation decisions are still sought. In the meantime, carefully designed heuristics consider at least some interactions. For example, the GraalVM compiler [47] checks if a loop can be vectorized and if so, omits the application of partial loop unrolling [27]. Again, the assumption that vectorization is more beneficial than partial unrolling holds only for the *average* program and might be invalid for a particularly important user application.

In static compilation, machine learning has already been shown to outperform human-crafted compiler heuristics [2, 7, 31, 43]. However, there is significantly less research regarding the usage of machine learning in dynamic compilation and we are not aware of any state-of-the-art dynamic compiler that is learning compilation decisions at run time to fit the current environment.

In order to solve above problems we propose *machine-learning-based self-optimizing compiler heuristics*: an end-to-end approach to

learn compilation decisions at run time from dynamically extracted performance metrics. We tune our model in recurring phases, with the variably sized batch of data collected since the last update or program start: In the *data generation phase* we are using a recently established technique called *compilation forking* [33], which produces comparable performance data to determine the impact of optimization decisions based on observed code features. This data is used in the *learning phase* to train a machine learning model, which can predict the best optimization decision for a given piece of code. In the *prediction phase*, this model is deployed and previously compiled functions are deoptimized and re-compiled using the model decisions. Our approach happens dynamically while the program is being executed. This allows us to customize the compiler to specific programs or hardware without the assistance of compiler engineers. Furthermore, our approach can be used to assist compiler engineers to improve existing heuristics, as proposed in [32].

We implemented our approach in the GraalVM compiler, to evaluate it in one of the most highly optimizing Java compilers. Improving GraalVM's performance provides confidence that our approach is beneficial for real-world production systems. Furthermore, GraalVM's polyglot nature [45] is an optimal target for tuning generic, language-agnostic optimizations towards specific languages, such as JavaScript. This is possible, by directly working with GraalVM's graph-based intermediate representation (IR) [13, 15] rather than source code. As a sample optimization for this paper we chose *loop peeling* because of its interplay with other optimizations such as loop inversion, vectorization or guard optimizations. This interplay often makes designing heuristics for when to apply peeling a non-trivial task. Our approach is similarly applicable to any other optimization. For example, we are also experimenting with optimizations such as unrolling and vectorization which—for brevity—are not part of this paper. We learned models for each benchmark in the *DaCapo* [5], *DaCapo Scala* [38], *JetStream* [36] and *Octane*[1] [9] suites. Especially on the *JetStream* suite, towards which the GraalVM compiler was not tuned, our approach discovered significant speedups of more than 30% for multiple benchmarks. The largest slowdown over all benchmarks was 7%. Our research contributes

- a novel approach for learning optimization heuristics at run time in a dynamic compiler,
- an implementation of this approach in a dynamic compiler that is among the most highly optimizing Java compilers on the market
- a quantitative experiment where a loop peeling decision is learned at run time, which outperforms heuristics by up to 30+% on well-known benchmark suites
- a qualitative experiment where a machine learning model is improved with new data at run time

The remainder of this paper is structured as follows. Section 2 gives an overview of related work and briefly explains compilation forking and loop peeling. Section 3 provides an outline of our approach. Thereafter, Section 4 explains implementation details in the GraalVM whereas Section 5 summarizes our machine learning pipeline. Section 6 shows our evaluation methodology and results. Finally, Section 7 discusses limitations and future work.

[1] https://github.com/chromium/octane

2 BACKGROUND

We first give an overview of a recently introduced technique called *compilation forking* which we use for generating performance data of different compilation decisions in single program runs. Then, we discuss related work in the area of machine learning in compilers. Finally, we give insight into *loop peeling* which was used as a sample optimization for evaluating our approach.

2.1 Compilation Forking

Compilation forking [33] is a novel approach for evaluating the performance impact of local optimization decisions in a dynamic compiler. It requires only a single program run to evaluate mutually exclusive optimization decisions and can therefore be used transparently for generating data.

Concepts, such as iterative compilation [6], can hardly be applied in dynamic compilation where profiling, deoptimization or memory and timing thresholds may lead to different compilations for the same function in different runs. Compilation forking faces these problems by creating copies (called *forks*) of the state of an intermediate compilation right before the optimization—whose impact has to be measured—is applied. These forks are compiled with different optimization parameter values and are instrumented for performance measurements. This ensures that forks share the same compilation history, up to the point where the measured optimization is applied. All forks are then recombined into a dispatch function which transparently executes one fork per invocation. Therefore, compilation forking is also related to multi-versioning [48], as multiple versions of the same code are executed in the same run. These versions are executed alternatingly or in a random order to average out measurement noise caused by the environment. This reduces the CPU and OS stability requirements and allows for making consistent measurements without full control of the surrounding system. For the remainder of this paper we will use the term *fork* as one version of a code produced by compilation forking.

2.2 Related Work

There is an extensive set of research in the domain of machine learning for compilers [2, 43]. However, our approach combines multiple aspects which we are not aware of being found together in a sole research. It (1) *learns* or (2) *updates*—(3) *at run time*—a machine learning model which replaces an (4) *optimization heuristic* in a (5) *dynamic compiler*. We therefore address related work which is similar in one of these aspects to our approach.

Our approach is related to iterative compilation [6, 17, 18, 26] and multi-versioning [16, 25, 48]. In iterative compilation, functions are compiled multiple times with different sets of optimization parameters to converge on a near optimal compilation in terms of execution performance [6, 17]. This is not the goal of our approach which employs compilation forking [33] to create multiple non-optimal versions of a function to infer speedups or slowdowns of local optimization decisions, e.g. peeling of a particular loop. The knowledge of local optimization decisions is then used to create a machine learning model. Multi-versioning [16, 25, 48] is an approach related to iterative compilation, where multiple versions of a function or code snippet are deployed into an executable. At run time, the code which is best optimized towards the current

input is selected. Our approach differs from multi-versioning as the versions—*forks* in our notation—are only alive temporarily during data generation before re-compiling a function using learned decisions.

Tartara and Crespi Reghizzi [40] proposed *continuous learning of compiler heuristics*, which is a holistic approach for finding a set of optimization heuristics in a static compiler. They defined a grammar from which new heuristics can be inferred based on a pre-defined set of program features. For the composition of heuristics and a particular compilation plan they used genetic algorithms [8, 10, 40]. Their approach outperformed GCC O3 on the selected benchmarks which is impressive keeping its holistic nature in mind. However, this approach needs a controlled environment and multiple program runs to compare the performances and update heuristics in the static compiler. By utilizing compilation forking [33], our approach is capable of updating a learned model within a single program run for any dynamically compiled program. Furthermore, we replace heuristics by neural networks which are universal approximators to arbitrary functions compared to a limited search space spanned by the grammar as defined in [40].

Sanchez et al. [37] used machine learning in the IBM Testarossa JIT compiler to predict an optimal compiler phase plan. They use deoptimization to support data generation by re-compiling methods with different phase plans after a measurement interval has passed. Our approach executes different method versions alternatingly to average out measurement noise if the execution environment or program usage changes over time. Furthermore, Sanchez et al. [37] followed a traditional approach with a clear distinction between data generation and model usage, which both happen transparently in a single program run in our approach. They used support vector machines[11] in contrast to our research where we propose updating neural networks incrementally.

Improving loop related compiler optimizations has been the subject of various research in the past [19, 29, 30, 34, 39]. In a recent study, Mammadli et al. [30] investigated source-to-source transformations of loops prior to compilation to improve the compilation stability and the performance of the compiled programs. They are using a neural network for predicting the performance impact of source-to-source transformations on a subsequent compilation, which outperforms the used baseline significantly. However, their approach happens fully offline and provides yet another static heuristic specific to the compiler configuration which was used for creating the training data.

Wang et al. [42] present *SuperSonic*, a tool for automatically tuning hyper-parameters to optimize reinforcement learning (RL) [22] architectures for the domain of code optimization. After deployment, the reinforcement learning client can be further refined with unseen data. However, this task requires the storage of execution data across multiple runs, compared to the fully transparent model update in single runs as we propose in our work. *SuperSonic* outperforms existing auto-tuners, such as *OpenTuner* or *CompilerGym*, but it is not evident if these frameworks could work in a dynamic compilation environment.

An additional area of application where we envision our approach to be useful is in compiler optimization construction and tuning. Therein, self-optimizing compiler heuristics can be used offline by compiler engineers to evaluate heuristics under development and find weakpoints, without deploying any machine learning in the final product. This has already been proposed in the past [32, 41], however, without taking highly domain specific models into account. Recently, Cummins et al. [12] proposed *CompilerGym*, a framework opens up compiler research to machine learning experts. Their framework makes compiler tasks, such as phase ordering for LLVM or GCC flag tuning, available to performing AI research in an easily accessible way. This includes automatically obtaining benchmark data and providing AI algorithms or APIs for hyper-parameter tuning.

2.3 Loop Peeling

Loop peeling [4] is a transformation which moves the first or last few loop iterations out of the loop. Listing 1 shows a loop which when peeling the first iteration results in the code shown in Listing 2.

```
for (int i = 0 ; i < limit; i++ ) {
    // loop body
}
```

Listing 1: Loop before peeling the first iteration.

```
if ( 0 < limit ) {
    // loop body
}

for (int i = 1 ; i < limit; i++ ) {
    // loop body
}
```

Listing 2: Loop after peeling the first iteration.

The `if`-check might be removed since we know that `i==0` at this point. Depending on the loop body, further optimizations can be enabled using the knowledge about `i`. Thus, loop peeling is called an *enabling optimization*, as performance improvements are not directly caused by peeling but indirectly by enabled follow-up optimizations. For example, loop peeling may allow for removing redundant checks or assignments caused by special cases in first loop iterations. This is shown in Listing 3, where the variable `redundant` is assigned in each loop iteration, although being always `i-1` after the first iteration.

```
int redundant = 0
for (int i = 0 ; i < 100; i++ ) {
    dst[i] = src[i] + redundant;
    redundant = i;
}
```

Listing 3: Loop with redundant variable.

After peeling the loop, the redundant variable can be omitted, as it can be statically inferred that for `i >= 1` its value is always `i-1`. The resulting code is shown in Listing 4.

```
dst[0] = src[0] + 0;
for (int i = 1 ; i < 100; i++ ) {
    dst[i] = src[i] + (i-1);
}
```

Listing 4: Peeled loop with redundant variable removed.

In a dynamic compiler, profiling information and assumptions can enable more aggressive optimizations after peeling. Listing 5 contains two nested loops, where `limit1` and `limit2` might be subject to assumptions.

```
for( int i = 0; i < limit1; i++ ) {
    int result = 0;
    for( int j = 0; j < limit2; j++ ) {
        // side-effect-free computations
        result += ...
    }
    if( i != 0 ) process(result);
}
```

Listing 5: Nested loop.

If `limit1` is assumed to be 1, peeling the outer loop would lead to the whole code being never executed.

```
if (0 < 1) {
    int result = 0;
    for ( int j = 0; j < limit2; j++ ) {
        // side-effect-free computations
        result += ...
    }
    if ( 0 != 0 ) process(result);
}
for( int i = 1; i < 1; i++ ){
    int result = 0;
    for ( int j = 0; j < limit2; j++ ) {
        // side-effect-free computations
        result += ...
    }
    if ( i != 0 ) process(result);
}
```

Listing 6: Peeled outer loop before further optimizations.

In the peeled code in Listing 6, the compiler can see that the result computed by the inner loop is never used and can remove the code that computes it. The remaining loop, starting with `i == 1`, can be removed as well, since the upper bound is already reached. Such cases can have tremendous positive impacts on overall program performance, but they rare in practice and incorporating them in static heuristics is difficult. On the other side, loop peeling may also hinder other optimizations. For example, loop vectorization might be only applied if there is a certain number of loop iterations in a counted loop. If this number is decreased by peeling, the beneficial vectorization of the loop can be prevented. In contrast, peeling a loop may also enable vectorization. Loop peeling, despite being a seemingly small transformation, can have large impacts on program performance due to its nature as an enabling optimization.

In the GraalVM compiler, loop peeling can only remove the first iteration of a loop. Therefore, for the remainder of this paper, we will be using *peeling* synonymously to *peeling the first loop iteration*. This implies that peeling decisions are boolean decisions indicating whether the first iteration should be peeled (=true) or not (=false).

3 APPROACH

In this section, we present *machine-learning-based self-optimizing compiler heuristics*. This novel approach facilitates replacing heuristics in a dynamic compiler with learned models which are tuned with actual data at run time. It therefore automatically considers peculiarities of the user domain including different hardware or different types of programs. For small yet static domains, overfitting can be exploited to make optimal decisions, similarly to iterative compilation [6, 17]. However, there are multiple advantages compared to iterative compilation: First, peculiarities of dynamic compilation are taken into account by considering the compilation history when measuring the impact of a compilation decision. Second, our approach enables learning local compiler optimization decisions rather than optimizing compiler flags used for whole programs. Third, by automatically storing learned decisions in a model, this model can be re-used for similar domains on the fly. Using a machine learning model as knowledge base facilitates both using a pre-trained model and refining the model if new data is acquired. This is a significant advancement over the state-of-the-art, where machine learning models are deployed as unchangeable, static heuristics. Exceptions are found in recent research regarding reinforcement learning in static compilation [12, 41, 42].

Our approach consists of three phases, two of which correspond directly to how the dynamic compilation is performed. These are the *data generation phase*, the *learning phase* and *prediction phase*. They can be iterated multiple times (see Figure 1), which enables iterative refinement or adjustment to new data or circumstances. Figure 1 shows the life-cycle of a method *foo* throughout these phases. It starts with exploring the impact of different optimization decisions by employing compilation forking [33] in the data generation phase. After a learning phase, where either a new model is created or an existing model is updated, *foo* is deoptimized and re-compiled with the model decision replacing the human-crafted heuristic. We now discuss these phases in detail.

3.1 Data Generation Phase

In the data generation phase, feature data and performance metrics of program snippets are collected. The performance metrics determine how a code, which is described by the feature data, needs to be optimized. Features have to be extracted at compile time whereas performance metrics need to be measured at run time.

Feature Extraction. In a machine learning context, features are the input to a model. They describe the code snippet for which an optimization decision has to be made. Examples of features for the loop peeling model are the loop depth and the number of branches in the loop (see Section 5.3). It is essential to extract feature data at compile time as close to the monitored compilation decision as possible. For example, when compiling a function with multiple loops which can be peeled, the feature extraction for `loopB` needs to take place after `loopA` has been processed. Otherwise, the extracted features for `loopB` would not account for changes made by peeling `loopA`. In related work, feature data is often extracted either before compilation or before the optimization phase is started. We extract features during fork creation at compile time and write them to a shared storage.

Performance Data Extraction. The performance data which is required to identify beneficial or disadvantageous decisions needs to be extracted at run time. For extracting comparable performance measurements from dynamically compiled programs, several challenges need to be taken into account: (A) The outcome of multiple decisions needs to be measured in a single program run, (B) the

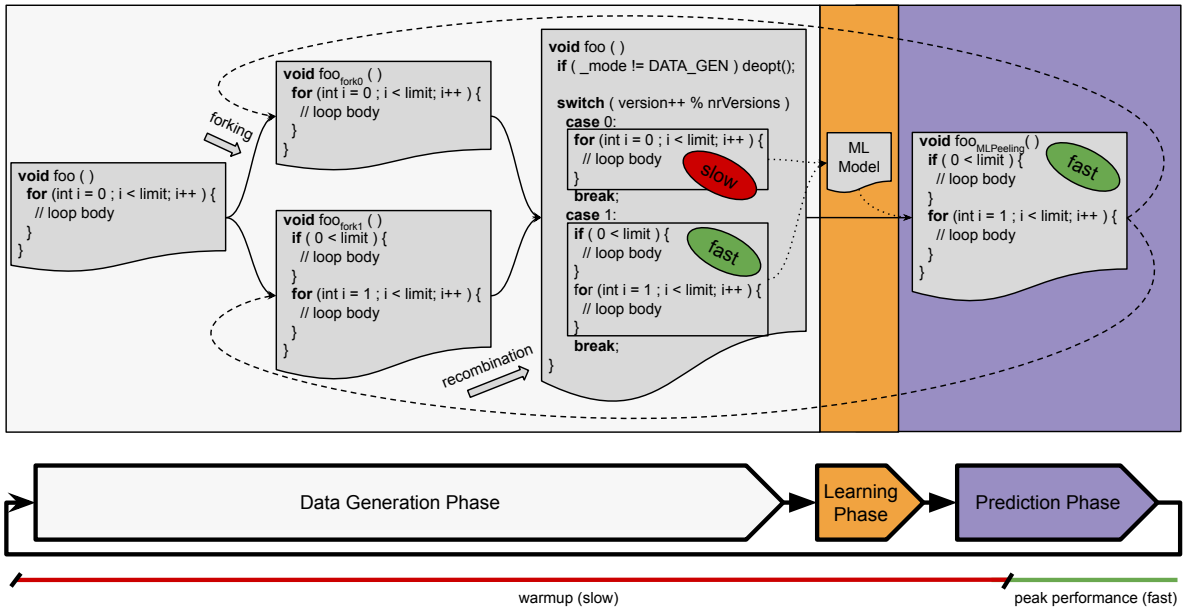


Figure 1: Overview of our approach applied to an example function.

impact of different decision outcomes needs to be measured based on the same compilation history prior to the decision, and (C) noise introduced by different program states or function parameters has to be handled. These requirements are met by using *compilation forking* [33] which we discussed in Section 2.1. It enables extracting aggregated performance data for multiple versions (*forks*) of a code with different compilation decisions and outputs tuples of kind

$$(ID, run\ time, invocations, min, max)$$

They contain how often a fork has been executed (*invocations*) and the total execution time (*run time*) aggregated from all invocations. *min* and *max* are the minimum and maximum execution time from all invocations of a fork. Aggregated performance data reduces the statistical capabilities which would be possible if performance data were stored per invocation. However, forks might be executed millions of times in one program run. Storing performance data for each invocation would introduce a huge overhead. The aggregated data is stored and updated locally in the dynamic runtime until the data generation phase is ended. The data generation phase ends after a specified period of time or after enough data has been collected. The dynamic runtime disables further data generation and persists the aggregated performance data in a shared storage.

3.2 Learning Phase

In the learning phase, the machine learning pipeline is invoked to either create a new a machine learning model from the gathered data or to refine an existing model. When training a new model, overfitting will likely occur as only data from one program run is used for training. We discuss overfitting in Section 7.2. The phases in the machine learning pipeline can be subdivided into data pre-processing, data filtering and model training. Data pre-processing associates the feature data with the performance data and creates a labelled data set. For example, in a loop peeling scenario the label for each set of features would be either `true|1` or `false|0` depending on whether peeling the loop described by the given features reduced the function's execution time or not. The created data set can be filtered, to remove data points which are likely subject to measurement noise or to remove features which should be excluded from the training process. This is discussed in greater detail in Section 5.2. If few data points remain after filtering, a data augmentation phase creates additional data to have enough data for later training. The model training phase will fit a model of pre-defined type and structure to the labeled data (see Section 5.4). Depending on the problem at hand, the machine learning model produces one or multiple predicted values from the input features. For example, in a loop peeling scenario the model would output either a `1` or a `0` as prediction whether to apply peeling or not. In Section 5.4 we describe the structure and hyper-parameters of the neural networks which were trained for each benchmark.

A serialized version of this model is written to a shared storage along with an ordered list of features which need to be used as its input. The model and the definition of the input features can then be fetched by the dynamic compiler which switches to *prediction mode*. This triggers deoptimization of all functions which were compiled and instrumented in the data generation phase during their next execution.

3.3 Prediction Phase

In the prediction phase, the previously trained or refined machine learning model is deployed in the dynamic compiler. The dynamic compiler in *prediction mode* uses the model to replace human-crafted compilation heuristics or decisions. All functions which were compiled and instrumented in the data generation phase are deoptimized and re-compiled using the model. This can be seen in

Figure 1. After another—deferred—warm-up period, the program reaches a stable state of peak performance, where certain optimizations were subject of learned decisions. The prediction phase can be used without a preceding model training, if an already trained model is available before program start.

4 IMPLEMENTATION

We will now present the details of our reference implementation in the GraalVM [47] and discuss the machine learning pipeline in Section 5. GraalVM uses a graph-based intermediate representation (GraalIR) [13, 15] which is a superposition of data flow graph and control flow graph. By directly operating on the IR graph and by extracting features from the graph rather than source code, our implementation can optimize programs from any programming language which is supported by GraalVM's polyglot framework *Truffle* [45].

4.1 Architecture

Our system architecture for implementing *machine-learning-based self-optimizing compiler heuristics* in the GraalVM is shown in Figure 2. We decided to use a client-server model to retain flexibility of

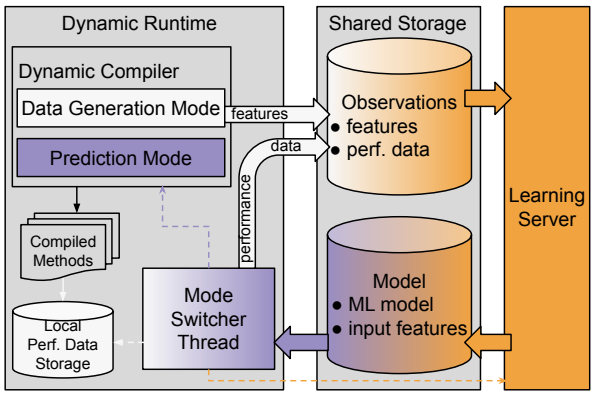


Figure 2: System architecture.

where the potentially GPU-supported training process is executed. The *dynamic runtime* is capable of executing dynamically compiled programs. This includes starting the execution in an interpreter, invoking the dynamic compiler for hot methods and switching from interpreted to compiled methods after compilation or vice-versa after deoptimization. The *dynamic compiler* applies several optimizations in fixed order to the compiled method before emitting code. While we were using a method-based compiler, any compiler which applies optimizations in a deterministic order is suitable for implementing our approach. In *data generation mode*, the compiler emits feature data of program parts that are subject to an optimization. Furthermore, it explores multiple optimization variants by employing compilation forking [33] and extracts performance measurements for each variant. This performance data is stored locally in the dynamic runtime where total execution time and number of invocations can be updated efficiently during data generation. In *prediction mode*, the compiler uses a machine learning model to make compilation decisions. The *mode switcher thread* is a background thread in the dynamic runtime which triggers the transition between the two compiler modes. This includes communicating the feature and performance data to the learning server, awaiting its response, providing the trained model to the compiler and changing the compiler mode flag to *prediction mode*.

We use a shared storage for passing data between the dynamic runtime and the learning server. If the learning server runs locally, this is more efficient than sending large files with features or models. One section in the shared storage holds the feature and the performance data which was created in the data generation phase. The other section holds the machine learning model after training has finished, along with a description of the input features.

The learning server contains a pipeline (Section 5) for training or updating machine learning models.

4.2 Compilation Forking

In order to reduce the state space when creating forks for peeled loops, we configured compilation forking to process loops independently. This means that for a function with three loops four forks will be created. One fork has no loop peeled and is considered as the baseline. In all other forks exactly one loop is peeled. If a fork outperforms the baseline it is inferred that peeling the respective loop was beneficial. Peeling of nested loops might violate the assumption of independence and produce inaccurate data points for training. We accepted this as trade-off to make our approach more applicable by keeping the state space feasible.

4.3 Deopt Instrumentation

In addition to the instrumentation added for performance measurement and fork recombination, we introduced a check of the compiler mode flag at the start of each recombined function. This conditional is only added by the compiler in the *data generation mode*. If the compiler mode is set to *prediction mode*, the function is deoptimized and re-compiled at its next invocation. This instrumentation is added to the compiler IR graph - Figure 1 depicts it in pseudo-code.

4.4 Mode Switching

In the current implementation, the data generation phase ends after a fixed period of time, which is specified as dynamic runtime parameter. As part of future work, we plan to make this fixed time interval dynamic, based on the progress of the program warm-up. This can be solved by tracing the compilation frequency, which is already implemented in the GraalVM compiler. After the data generation phase has ended, the aggregated performance data is written into a as `json`-file and moved to the shared storage. The feature data has already been written into the shared storage at compile time.

The learning server is invoked via a socket connection by sending either a `learn` or an `update` request. These requests also include the paths to the feature and performance data. The response contains the path to the model in the shared storage or a forwarded error message if training was not successful. After changing the compiler mode to *prediction mode*, the dynamic compiler replaces the optimization phase, which supported forking with a version which fetches the ML model from the shared storage location.

5 MACHINE LEARNING FRAMEWORK

According to the state-of-the-art, we implemented our machine learning pipeline in Python and used PyTorch [35] for training neural networks. We created an extensible framework to adapt to new optimizations by configuring the filters and learning pipeline like a plug-in system. The model architecture and hyper-parameters have been chosen empirically for the experiments conducted in the paper and might be refined in the future.

5.1 Data Pre-processing

Data Merging. The feature data and the performance data are written to the shared storage at compile time and at program execution, respectively. Figure 3 depicts the feature data for a sample function on the right and the performance data of the same function and its two forks on the left. During compilation forking an artificial identifier is introduced for each fork by attaching the fork number to the original compilation identifier. This is necessary to distinguish multiple compilations of the same function. The first step in the pipeline is to merge the feature and the performance data using this compilation identifier as a key.


```
{"method": "Clazz.method(Clazz.java:123)",
"compID": "Compilation-10027_Fork0",
"invocations": "171571",
"time": "36553828",
"min": "40",
"max": "16816"}
,
{"method": "Clazz.method(Clazz.java:123)",
"compID": "Compilation-10027_Fork1",
"invokations": "171562",
"time": "36945844",
"min": "40",
"max": "19416"}
```

```
{"method": "Clazz.method(Clazz.java:123)",
"compID": "Compilation-10027_Fork0",
"context": "peeling",
"features": {
    "size": "26",
    "depth": "1",
    "nrChildren": "0",
    "hasParent": "false",
    "nrBackedges": "1",
    "nrExits": "1",
    "counted": "true",
    "isVectorizable": "true",
    (...)}
```


Figure 3: Performance data (left) and feature data (right).

Shape Unification. Typical features are the counts of specific nodes in Graal's IR, e.g. `#AddNodes` or `#IfNodesInLoop`. To reduce the memory footprint, feature extraction only dumps non-zero node counts. During data pre-processing, however, the feature space needs to be expanded to a uniform shape, including also the features with zero counts.

Labeling. The output produced by compilation forking (see Figure 3) consists of aggregated execution times for each function. This success metric has to be turned into labels for training the machine learning model. For loop peeling, this label is created using the logarithmic average speedup compared to a baseline where peeling is disabled for the particular loop. This is shown in the following equation.

$$peel = \begin{cases} 1 & logSpeedup \geq \log(1+\epsilon) \\ 0 & logSpeedup < \log(1+\epsilon) \end{cases}$$

$$logSpeedup = \log(\frac{avgTime_{noPeel}}{avgTime_{peel}})$$

The ϵ value can be used to label peeling decisions with only minor performance benefits as *no peel* to avoid a code size increase for very small performance gains. This label strategy implies that we see loop peeling as a classification problem with only `1` (= peel) or `0` (= no peel) as outputs. It is also possible to model the task as a regression problem and use the `avgSpeedup` as label. This would require a threshold for the predicted speedup in the compiler above which a peeling is applied. While part of the data pre-processing, labeling happens after the data filtering. This allows applying filters based on the measured performance values.

5.2 Data Filtering

Data pre-processing turns the data into a format that is understandable for a machine learning model. Data filtering manipulates the data set to reduce noise and improve the overall data quality and feature relevance. First, we apply filters which remove observations, i.e. features and respective labels, as a whole. Then, we apply filters which remove feature columns for all observations and reduce the dimensionality of the model input. After all filters have been applied, data augmentation will increase the remaining data set size if necessary.

AvgRuntimeFilter. This filter removes data points if the average run time is below or above a specified threshold. Functions with a very small run time are more easily subject to noise and are therefore excluded from training. We did not set an upper limit for the average run time, as especially long-running functions are desirable to be optimized.

MinInvocationsFilter. The premise of compilation forking is that, when executing different optimization variants in one program run, differences in execution time caused by the environment or parameters will cancel out across many invocations. Therefore, functions with only few invocations are removed as their measurements are not stable enough.

AbsoluteDifferenceFilter. This filter addresses the label ambiguity caused by measurement noise. If the absolute difference of the average execution time of the baseline and the optimized version are closer than the assumed measurement inaccuracy, the observation is removed as the classification label cannot be identified correctly.

SkewednessFilter. In the absence of separate performance measurements per execution, we implemented this filter to remove data points with few large outliers. If removing the maximum execution time has a noticeable impact on the average execution time, the filter will remove the data point.

$$SkewednessFilter = \frac{avgTime}{(\frac{totalTime-max}{invocations-1})} > 1+\epsilon$$

FeatureDiversityFilter. This filter removes features with little information. A feature is the more informative, the more different values are found in all observations. Therefore, the filter computes a histogram of all values for each feature. If the most frequent value occurs in more than 95% of the data, say, the feature is removed. This has an especially high impact on rare node types, whose frequencies are zero in most functions. However, the filter can only be applied when training a new model because the number of features for an existing model is fixed.

Data Augmentation. Data augmentation is the process of creating new data points from existing ones. It can be useful if little data is available in order to reduce overfitting. If our approach is used to train new models from single program runs, data augmentation will automatically be applied if the number of data points is below a threshold. In our domain, the implications of changing feature

values are unclear and therefore unsuited for creating correct data points. Thus, we perform data augmentation by adding data points with identical features but slightly changed performance values. This can result in data points with very small performance differences to produce new data points with opposite classification labels.

5.3 Features

Table 1 shows a list of all extracted features for loop peeling wich are based on the loop features presented in [33]. The features are either boolean features—with the suffix "?"—or integer features and are divided into seven categories. Values in brackets are placeholders and summarize multiple features.

As GraalVM uses a graph-based intermediate representation (IR) [13, 15] many features are graph-related. For example, the *size* of the loop corresponds to the number of its IR nodes and the *node cost* [28] is a GraalVM heuristic for estimating the execution time for a set of nodes. There are six types of edges in the IR which, together with their origin and destination, lead to 18 edge features. Due to the large number of different node types, there can be up to 1000 features before reduction. However, many node types appear very rarely or never in certain phases of the compilation. Thus, the number of selected features in our experiments varied between 150 and 200, depending on the *FeatureDiversityFilter*.

Table 1: Features for loop peeling, based on [33]

Loop General	**Loop Nodes**	**Loop Operands**
size	#fixedNode	#objectStamps
depth	#floatingNodes	#intStamps
node cost	#PhiNodes	#floatStamps
#children	#ProxyNodes	#volatileFieldAccess
#backedges	#IfNodes	#staticFieldAccess
#exits	#[IRNodeType]	**Loop Edges**
counted?	**Graph**	#[EdgeType]IntoLoop
can ends safepoint?	node cost	#[EdgeType]InLoop
vectorizable?	#loops	#[EdgeType]OutOfLoop
Loop Parent	max loop depth	**Loop Execution**
hasParent?	#branches	frequency
parent size	#[IRNodeType]	constant max trip count?
parent node cost		has exact trip count?
		can overflow?

5.4 Model Training

There are many different types of machine learning models and hyper-parameters for configuring them. In our approach we are using neural networks, which are easy to update if the *Data Generation Phase* and the *Learning Phase* are executed repeatedly or a pre-trained model needs to be refined. For the loop peeling models we used residual neural networks [20] with full pre-activation residual blocks. Residual networks include skip-connections, which improves training large networks with little data. Figure 4 depicts the three types of layers which were used in the networks. The input block consists of three linear layers followed by rectified linear units (ReLU). Its number of inputs depends on the feature reduction process. To counter overfitting, a batch normalization and a dropout layer (probability = 0.2) are added. The output block uses four linear layers and produces in case of loop peeling exactly one output which indicates whether to apply the transformation or not. Between input and output blocks there are five residual blocks with full pre-activation, as shown in the top center of Figure 4. The resulting deep residual neural network and its skip-connections are summarized in the bottom of Figure 4. While this architecture has provided good results, we assume that other network structures could perform similarly. We used Adam [23] as optimizer with a learning rate of 3e−3 and a weight decay of 5e−5. As loss function we used binary cross entropy (BCE).

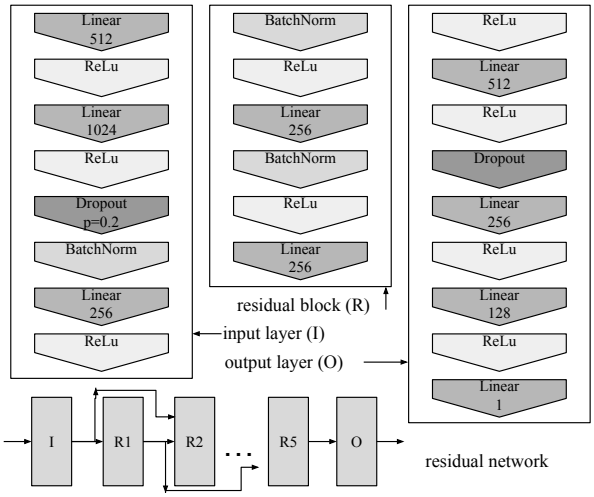


Figure 4: Residual network blocks and network structure.

Loss Scaling. When training a classifier, the actual performance impact of an optimization is lost after labeling data points with either *peel* or *noPeel*. Thus, equal emphasis is put on learning less impactful and more impactful decisions, during training. We implemented a scaled version of the binary cross-entropy loss, which reduces the loss for less important data points and assigns a higher loss for data points with large performance impacts. This way, we shift the focus of the trained model towards predicting more impactful decisions correctly, at the cost of incorrectly predicting less impactful decisions. The scaled BCE loss was implemented using a double Gaussian curve as a filter function.

6 EVALUATION

We established two major claims regarding our approach, which are manifested in two hypotheses that need to be tested.

Hypothesis 1. Machine-learning-based self-optimizing compiler heuristics can improve the peak performance of dynamically compiled programs by replacing a compiler heuristic with a learned model at run time.

Hypothesis 2. Machine-learning-based self-optimizing compiler heuristics can be used to refine a pre-trained machine learning model and tune it towards a specific environment during dynamic compilation.

To test these hypotheses, we implemented our approach in the GraalVM compiler [47], which is among the most highly optimizing Java compilers on the market[2]. We replaced the loop peeling optimization with a learned model and evaluated the two hypotheses using benchmarks from the well-known benchmark suites *DaCapo* [5], *DaCapo Scala* [38], *JetStream* [36] and *Octane* [9]. *Hypothesis 1* is addressed in a quantitative experiment in Section 6.2 whereas *Hypothesis 2* is addressed in a qualitative experiment in Section 6.3.

6.1 Experimental Setup

All experiments where executed on an Intel I7-4790K at 4.4GHz with 20GB main memory and two GeForce GTX 1070, which were used for model training. Hyper-threading, frequency scaling and network access were disabled.

Warm-up. The experiments in Section 6.2 and Section 6.3 report the impact on the benchmark peak performance which excludes the preceding warm-up time. The total warm-up time is the sum of 1) the data generation time including forking, 2) the model training time and 3) the warm-up time for re-compiling previously forked functions using the learned model. Therefore, traditional metrics for evaluating the program warm-up time, such as the number of warm-up iterations, are not applicable for our approach. The data generation time and the model training time are based on hyperparameters which can be freely chosen. For example, we defined the data generation time to be 5 minutes for each *DaCapo* and *DaCapo Scala* benchmark, 7.5 minutes for each *Octane* benchmark and 10 minutes for each *JetStream* benchmark. These numbers were conservative estimates to maximize the number of methods which could be compiled with forking and to aggregate plenty of performance measurements for each fork. Automatically minimizing the data generation time for particular programs based on the program warm-up is subject to future work. In addition, we empirically evaluated that model training would take less than a minute on our system. Thus, the number of benchmark warm-up iterations was increased to fit the data generation time, the model training time and the default warm-up time for the re-compilations. For reproducibility, Table 2 shows the factors by which the default GraalVM warm-up was increased in our experiments. For example, the *JetStream hash-map* benchmark is very short running and had to be increased by a factor of 120 to fit the selected data generation time. Finding a distinct way for evaluating the warm-up and further minimizing it will be an interesting part of future work, which is discussed in Section 7.4. Subsequently shown performance numbers refer to the peak performance of the program which is measured after the warm-up.

6.2 Training New Models

We investigated *Hypothesis 1* with a quantitative experiment using all benchmarks from suites *DaCapo* [5], *DaCapo Scala* [38], *JetStream* [36] and *Octane* [9]. For each benchmark execution we created a new model for replacing the loop peeling heuristic using the approach as described in Section 3. Despite measures, such as batch normalization, the small amount of training data caused some overfitting. However, for achieving the best performance this can be desirable.

[2]https://renaissance.dev/

Table 2: Factors, by which the initial benchmark warm-ups are increased.

DaCapo		D. Scala		JetStream		Octane			
avrora	12	apparat	30	bigfib	45	box2	25	raytrace	25
fop	30	factorie	4	container	12	code-load	25	regexp	4
h2	3	kiama	30	dry	50	crypto	12	richards	25
jython	4	scalac	7	float-mm	80	deltablue	25	splay	25
luindex	25	scaladoc	14	gcc-loops	35	earley-b.	25	typescript	4
lusearch	15	scalap	30	hash-map	120	gbemu	25	zlib	8
pmd	15	scalariform	35	n-body	18	mandreel	12	zlib-dim.	8
sunflow	6	scalatest	18	quicksort	33	navier-st.	12		
xalan	18	scalaxb	30	towers	75	pdfjs	8		
		tmt	10						

Figures 5 (*JetStream*), 6 (*DaCapo*), 7 (*DaCapo Scala*) and 8 (*Octane*) show the performance impact of our approach (abbreviated as *GraalML*) compared to the default GraalVM heuristics. Each benchmark has been executed 10 times, creating 10 different models in the process. All benchmark results are normalized to the median of the default GraalVM performance and the medians are displayed in the center of the boxplots.

For the *JetStream* benchmarks (see Figure 5), six out of nine benchmarks show significant speedups for the majority of trained models, with the median speedup for *gcc-loops* being close to a factor of two. Figure 5 indicates that the performance of benchmarks which are run with our machine-learning-based approach often have high variance. There are multiple reasons for this instability when training models with little data. Before training, the extracted data is randomly split into a training data set and a validation data set. This random split can affect the model if important data points are moved to the validation data set and are thus omitted during training. Overfitting can also cause problems in small data sets if code is compiled differently in multiple runs. If dynamic compilation produces different data in the data generation phase and in the prediction phase, an overfitted model can be confused by the non-fitting data.

The *DaCapo* benchmarks (see Figure 6) show similar or slightly worse performance (up to 4%) for the GraalML configuration compared to the default heuristics. We identified two major reasons for this. First, *DaCapo* is one of the benchmark suites which was used for optimizing the GraalVM heuristics. Thus, the GraalVM heuristics are already tuned towards the DaCapo benchmarks. Second, as mentioned in Section 4.2, compilation forking is implemented in a way where loops are assumed to be independent of each other. For nested loops this assumption might fail and can produce misleading performance results which is not the case for the default GraalVM heuristics.

For *DaCapo Scala* (see Figure 7) one large speedup could be measured (*scalatest*) - most other benchmarks perform similarly to the default heuristics. The slowdown for *tmt* is caused by the deoptimization before switching to *prediction mode*; using the ML model from the start would lead to similar results as with the default GraalVM heuristics.

For the *Octane* suite (see Figure 8) multiple speedups of more than 5% were measured and few minor slowdowns of less than 2%, with only *typescript* having a more significant slowdown of 4%.

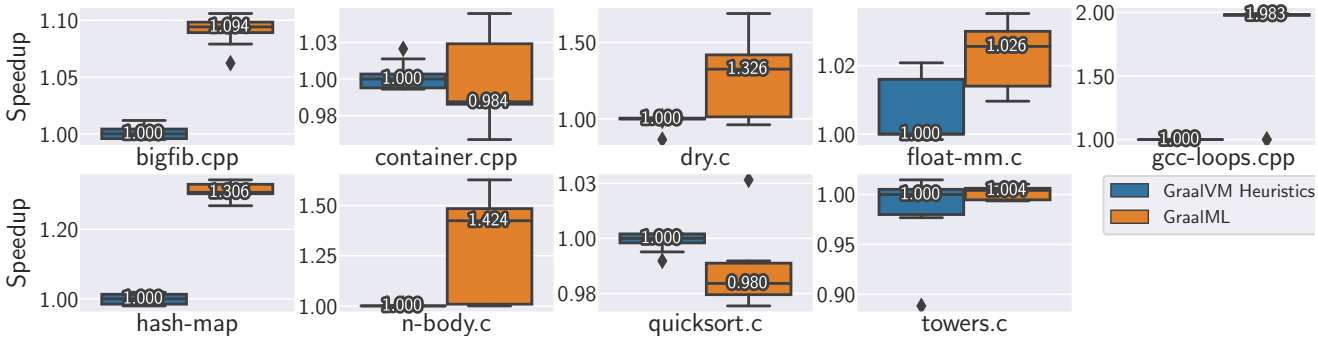


Figure 5: JetStream peak performance. Higher is better.

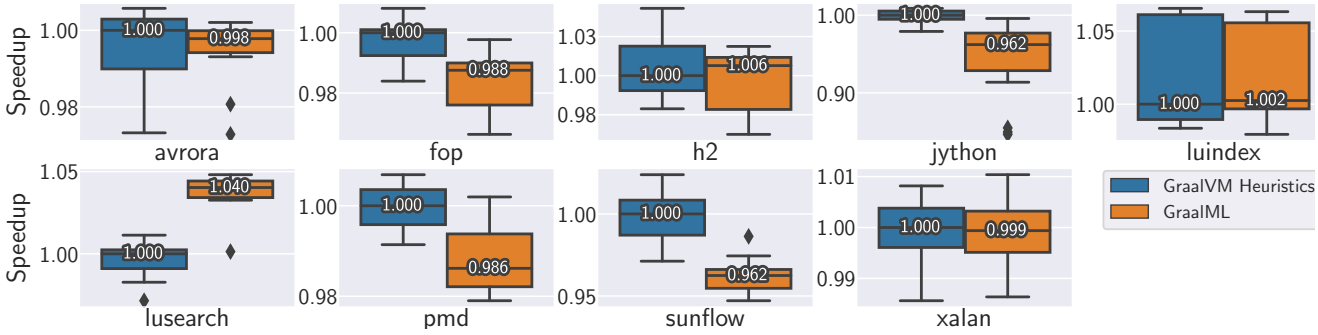


Figure 6: DaCapo peak performance. Higher is better.

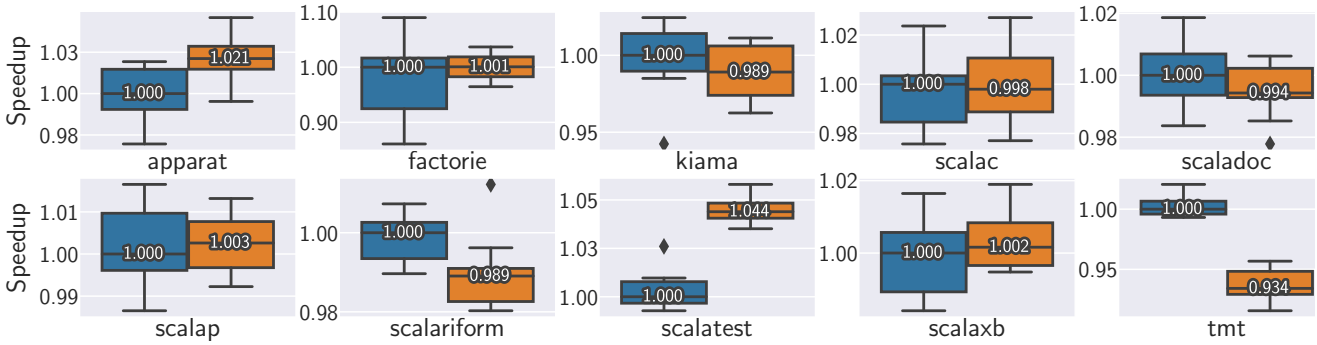


Figure 7: DaCapo Scala peak performance. Higher is better.

The presented quantitative experiments have shown that our approach can outperform existing heuristics with multiple speedups of more than 30% compared to few regressions of up to 7%. This supports *Hypothesis 1*. Especially, if models are trained for single programs overfitting can produce extremely good results. However, it also increases the performance variance and reduces generalization. This is further discussed in Section 7.2. Our approach is able to to compete with of one of the most highly optimizing compilers when it comes to benchmarks towards which its heuristics were specifically tuned. However, automatically learning heuristics with similar performance for new domains, programs or hardware without additional engineering effort can be considered a huge advantage over hand-crafted heuristics.

6.3 Self-optimizing Model

An advantage of our approach over static heuristics—human-crafted or learned—is the continuous evolution of the model to fit the current environment or data. We conducted a qualitative experiment to test *Hypothesis 2* by showing how a pre-trained model for one benchmark optimizes itself to fit another benchmark. We hand-picked two benchmarks from different suites: *xalan* (*DaCapo*) and *gcc-loops* (*JetStream*). *DaCapo* benchmarks are Java programs whereas *JetStream* benchmarks are JavaScript programs. Thus, we expected that a model trained on the one would perform poorly on the other. All configurations contain 20 measurement runs which are normalized to the median of the default GraalVM performance for the respective benchmark.

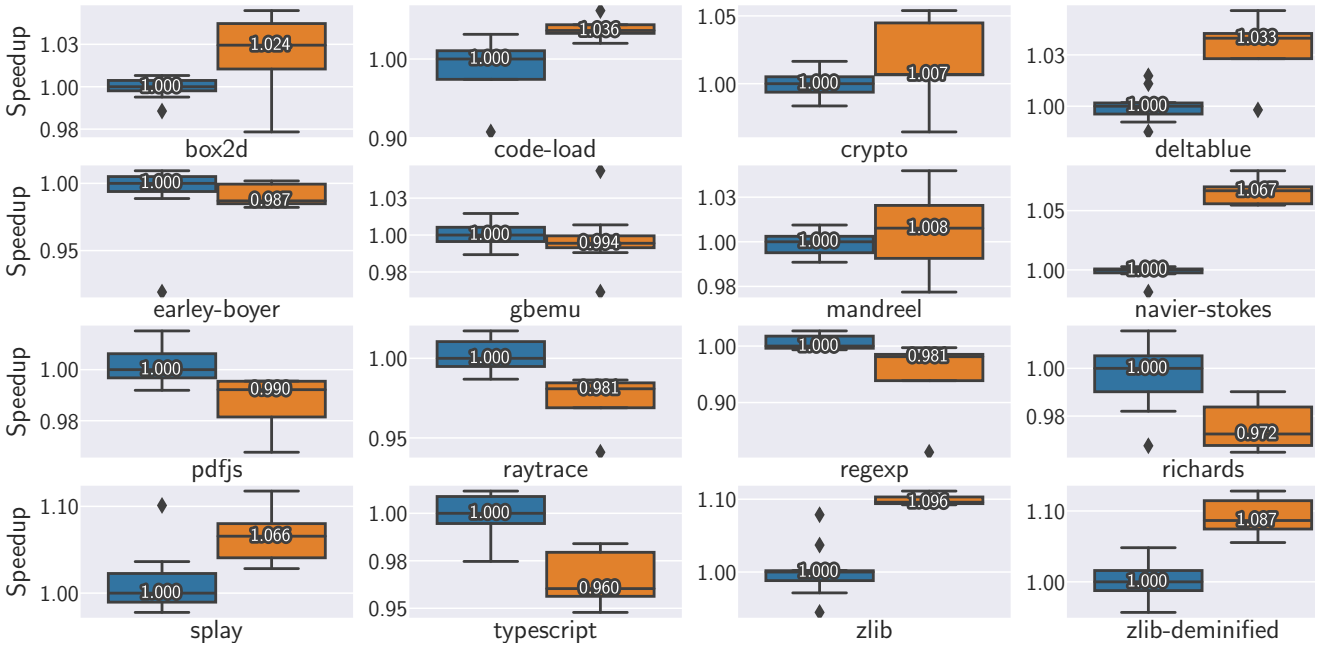


Figure 8: Octane peak performance. Higher is better.

The left part of Figure 9 compares the *xalan* benchmark with the default GraalVM configuration with an implementation of our approach in Graal (abbreviated as GraalML). For each GraalML measurement a new new model was created. GraalML produces similar results for the *xalan* benchmark compared to the default GraalVM heuristics. However, the variance of the *xalan* performance is also increased because the models are trained in a slightly different way depending on the extracted data. Figure 10 shows the performance

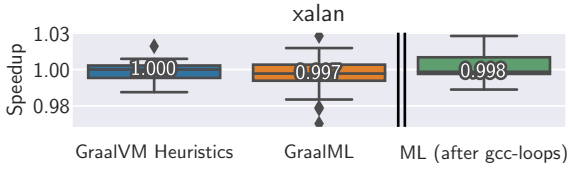


Figure 9: xalan (DaCapo) peak performance. Higher is better.

of the *gcc-loops* benchmark. For the *ML (xalan)* configuration the *gcc-loops* benchmark was executed in the prediction phase solely, using each previously trained *xalan* model in a separate run. This shows that the *xalan* models achieve similar performance to the default GraalVM heuristics for the *gcc-loops* benchmark. Some of the *xalan* models performed significantly better. This can be due to lucky "guessing" because the *gcc-loops* data is unknown to the model. The third configuration (GraalML) shows the *gcc-loops* performance when refining the *xalan* models at run time using our approach. The performance improvements for the *gcc-loops* benchmark are significant but slightly worse as when training a new model with data from *gcc-loops* only (c.f. Figure 5). This suggests that *Hypothesis 2* holds in that machine-learning-based self-optimizing compiler

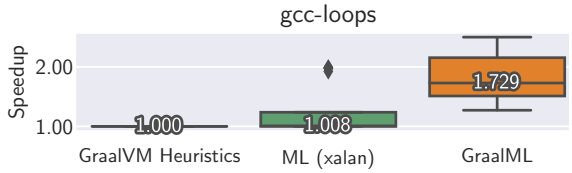


Figure 10: gcc-loops (JetStream) peak performance. Higher is better.

heuristics can be used to tune pre-trained models to new benchmarks. For completeness, the model, after being updated with data from *gcc-loops*, is tested for the *xalan* benchmark which is shown in the right part of Figure 9. The performance of *xalan* has slightly increased and the performance variance has decreased. This make sense as more data was used to train the model. Some data points extracted from the *gcc-loops* benchmark may have been useful for the *xalan* benchmark as well.

7 DISCUSSION

We have shown that machine-learning-based self-optimizing compiler heuristics can improve compiler optimizations in one of the most highly optimizing Java compilers on the market. This section addresses limitations which—if not inherent—will be subject to future work in order to further improve our approach.

7.1 Limitations of Forking

Compilation forking [33] has some limitations. It is not supposed to be used for exhaustive explorations of large optimization spaces. For example, a function with 10 consecutive loops would produce $2^{10} = 1024$ forks if all combinations of peeling decisions were taken

into account. By considering loops in isolation [33] the number of forks can be reduced to 11 but at the cost of ignoring potential impacts between multiple peeled loops. Depending on the number of forks, the enormously increased compile time can be a limiting factor for our approach as well: If the compile time of a forked function exceeds the time allocated for the data generation phase, no performance data is produced for this function. In general, the program's run time has to be sufficiently large to profit from our approach, which makes it especially suited for long-running server applications.

7.2 Overfitting

To reduce overfitting, our neural networks employ batch normalization and dropout layers. Nevertheless, when training a new model with data from only one program run, overfitting is very likely to occur. This can be deliberately taken into account, to create an optimization strategy tailored to a specific program, similar to iterative compilation. However, the potentially overfitted model can be re-used in future runs of this program, in contrast to iterative compilation. The more different the data is when incrementally updating a model, the more general the model becomes with a potential degradation for some programs compared to an overfitted model. This is seen when comparing the performance of the *gcc-loops* benchmark with a model that was only trained with this benchmark (median speedup factor 1.983, c.f. Figure 5) versus a model that was trained with *xalan* data before (median speedup factor 1.729, c.f. Figure 10). Overfitting is also likely to produce high performance variance if dynamic compilation compiles functions differently which results in predicting decisions for unknown data.

7.3 Updating a Model

Section 6.3 shows how an existing model can be updated with the newly fetched data as shown in Section 6.3. Long-running programs can also contain multiple learning phases to adapt to a changing environment. The new data is automatically pre-processed to match the model's feature set, which is fixed after the first training phase. This can lead to important features in the new data being ignored. It might therefore be beneficial to evaluate the importance of the features to be omitted and to automatically train a new model if necessary.

The more data an existing model has seen, the less it changes with new data. For updating a model more aggressively, the server can be configured to adapt the learning rate in order to escape (local) optima derived from old data.

7.4 Warm-up

As discussed in Section 6.1, evaluating the warm-up of our approach is different from traditional work in compilers. The total warm-up time is the sum of 1) the data generation time including forking, 2) the model training time and 3) the warm-up time for re-compiling previously forked functions using the learned model. The warm-up time of forking highly depends on how many forks need to be created for each function in a program and can vary a lot [33]. For generating data it is also not necessary to compile all functions with forking. This is controlled by the data generation time hyperparameter which can be chosen to end the data generation in the midst of program warm-up and just use the data collected up to that point. Automatically evaluating the progress of the program warm-up during forking and setting the data generation time accordingly is subject to future work. Similar trade-offs can be made to impact the model training time. Longer training time will fit a model more towards the recently provided data. This produces better results for the currently compiled programs at the cost of larger warm-up due to increased training time.

7.5 End-to-end Approach

Our approach does not require human interaction after deployment. However, there are some steps necessary prior to deployment: Compilation forking needs to be implemented for the optimization to be learned. This includes defining the features to be extracted. Currently, we have various sets of features which can be re-used if the domain is similar (e.g. loop-related optimizations). Additionally, hyper-parameters for the machine learning models and pre-processing steps have to be defined which are suitable for the predictive task. The learning framework provides a set of configuration options, which simplifies this setup. However, if the predictive task is very different from existing tasks, manual additions to the framework might be necessary. Lastly, the data generation time has to be set in accordance to the program run time and warm-up time. As part of future work, we will add an automated inference of smallest sufficient data generation time, based on an automated detection of the program's warm-up state.

7.6 Holistic Approach

Compilation forking can analyze the interplay of optimizations by employing nested forking which creates versions according to a grid search over multiple compilation decisions. In our approach, we only addressed learning single optimization decisions at run time. An offline approach would only require *some* data to be produced per data generation run, as there will be numerous programs executed which produce much data for creating a model with good generalization. In our approach, where data from only one program run can be used for creating a new model, investigating the interplay of multiple optimizations (i.e. >3) would be infeasible due to the limitations of compilation forking when it comes to an increased state space.

8 CONCLUSION

We have presented *machine-learning-based self-optimizing compiler heuristics*: an end-to-end approach to learn compilation decisions at run time from dynamically extracted performance metrics. It uses neural networks as knowledge base to update the learned compilation decisions at run time with new data. We showed in quantitative experiments that our approach can outperform human-crafted heuristics, especially for programs towards which these heuristic were not tuned. This eases deployment of compilers to new environments without investing additional engineering effort. Furthermore, our approach can be used to assist compiler experts when creating or evaluating new heuristics. Future work will address the discussed limitations and explore concepts such as "compilation-as-a-service" or "prediction-as-a-service" which are facilitated by the client-server architecture we proposed.

REFERENCES

[1] L. Almagor, Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven W. Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. 2004. Finding Effective Compilation Sequences. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems* (Washington, DC, USA) *(LCTES '04)*. Association for Computing Machinery, New York, NY, USA, 231–239. https://doi.org/10.1145/997163.997196

[2] Amir H. Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. 2018. A Survey on Compiler Autotuning Using Machine Learning. *ACM Comput. Surv.* 51, 5, Article 96 (Sept. 2018), 42 pages. https://doi.org/10.1145/3197978

[3] Joel Auslander, Matthai Philipose, Craig Chambers, Susan J. Eggers, and Brian N. Bershad. 1996. Fast, Effective Dynamic Compilation. *SIGPLAN Not.* 31, 5 (may 1996), 149–159. https://doi.org/10.1145/249069.231409

[4] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. 1994. Compiler Transformations for High-Performance Computing. *ACM Comput. Surv.* 26, 4 (Dec. 1994), 345–420. https://doi.org/10.1145/197405.197406

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

[6] François Bodin, Toru Kisuki, Peter Knijnenburg, Mike OBoyle, and Erven Rohou. 1998. Iterative compilation in a non-linear optimisation space. *Workshop on Profile and Feedback-Directed Compilation* (03 1998). https://hal.inria.fr/inria-00475919/document

[7] Alexander Brauckmann, Andrés Goens, Sebastian Ertel, and Jeronimo Castrillon. 2020. Compiler-Based Graph Representations for Deep Learning Models of Code. In *Proceedings of the 29th International Conference on Compiler Construction* (San Diego, CA, USA) *(CC 2020)*. Association for Computing Machinery, New York, NY, USA, 201–211. https://doi.org/10.1145/3377555.3377894

[8] John Cavazos and Michael F. P. O'Boyle. 2005. Automatic Tuning of Inlining Heuristics. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing (SC '05)*. IEEE Computer Society, USA, 14. https://doi.org/10.1109/SC.2005.14

[9] Stefano Cazzulani. 2012. Octane: The JavaScript benchmark suite for the modern web. https://blog.chromium.org/2012/08/octane-javascript-benchmark-suite-for.html retrieved May 25 2022.

[10] Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. 1999. Optimizing for Reduced Code Space Using Genetic Algorithms. *SIGPLAN Not.* 34, 7 (May 1999), 1–9. https://doi.org/10.1145/315253.314414

[11] Corinna Cortes and Vladimir Vapnik. 1995. Support-Vector Networks. *Mach. Learn.* 20, 3 (sep 1995), 273–297. https://doi.org/10.1023/A:1022627411411

[12] Chris Cummins, Bram Wasti, Jiadong Guo, Brandon Cui, Jason Ansel, Sahir Gomez, Somya Jain, Jia Liu, Olivier Teytaud, Benoit Steiner, Yuandong Tian, and Hugh Leather. 2022. CompilerGym: Robust, Performant Compiler Optimization Environments for AI Research. In *Proceedings of the 20th IEEE/ACM International Symposium on Code Generation and Optimization* (Virtual Event, Republic of Korea) *(CGO '22)*. IEEE Press, 92–105. https://doi.org/10.1109/CGO53902.2022.9741258

[13] Gilles Duboscq, Lukas Stadler, Thomas Würthinger, Doug Simon, Christian Wimmer, and Hanspeter Mössenböck. 2013. Graal IR: An Extensible Declarative Intermediate Representation. In *Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop*. 1–9. https://ssw.jku.at/General/Staff/GD/APPLC-2013-paper_12.pdf

[14] Gilles Duboscq, Thomas Würthinger, and Hanspeter Mössenböck. 2014. Speculation without Regret: Reducing Deoptimization Meta-Data in the Graal Compiler. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools* (Cracow, Poland) *(PPPJ '14)*. Association for Computing Machinery, New York, NY, USA, 187–193. https://doi.org/10.1145/2647508.2647521

[15] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. 2013. An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler. In *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages* (Indianapolis, Indiana, USA) *(VMIL '13)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/2542142.2542143

[16] Peng fei Chuang, Howard Chen, Gerolf F. Hoflehner, Daniel M. Lavery, and Wei chung Hsu. 2007. Dynamic profile driven code version selection. In *the 11th Annual Workshop on the Interaction between Compilers and Computer Architecture*. https://www.researchgate.net/publication/228952289_Dynamic_Profile_Driven_Code_Version_Selection

[17] Grigori Fursin, Albert Cohen, Michael O'Boyle, and Olivier Temam. 2005. A Practical Method for Quickly Evaluating Program Optimizations. In *High Performance Embedded Architectures and Compilers*, Nacho Conte, Tomband Navarro, Wenmei W. Hwu, Mateo Valero, and Theo Ungerer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 29–46. https://doi.org/10.1007/11587514_4

[18] Grigori Fursin, Cupertino Miranda, Olivier Temam, Mircea Namolaru, Ayal Zaks, Bilha Mendelson, Edwin Bonilla, John Thomson, Hugh Leather, Chris Williams, Michael O'Boyle, Phil Barnard, Elton Ashton, Eric Courtois, and François Bodin. 2008. MILEPOST GCC: machine learning based research compiler. In *Proceedings of the GCC Developers' Summit 2008*. https://hal.inria.fr/inria-00294704

[19] Ameer Haj-Ali, Nesreen K. Ahmed, Ted Willke, Yakun Sophia Shao, Krste Asanovic, and Ion Stoica. 2020. NeuroVectorizer: End-to-End Vectorization with Deep Reinforcement Learning. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization* (San Diego, CA, USA) *(CGO 2020)*. Association for Computing Machinery, New York, NY, USA, 242–255. https://doi.org/10.1145/3368826.3377928

[20] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Identity Mappings in Deep Residual Networks. In *Computer Vision – ECCV 2016*, Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling (Eds.). Springer International Publishing, Cham, 630–645. https://doi.org/10.1007/978-3-319-46493-0_38

[21] Urs Hölzle, Craig Chambers, and David Ungar. 1992. Debugging Optimized Code with Dynamic Deoptimization. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation* (San Francisco, California, USA) *(PLDI '92)*. Association for Computing Machinery, New York, NY, USA, 32–43. https://doi.org/10.1145/143095.143114

[22] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. 1996. Reinforcement Learning: A Survey. *J. Artif. Int. Res.* 4, 1 (May 1996), 237–285. https://doi.org/10.1613/jair.301

[23] Diederik Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. *International Conference on Learning Representations* (12 2014).

[24] P.A. Kulkarni, D.B. Whalley, G.S. Tyson, and J.W. Davidson. 2006. Exhaustive optimization phase order space exploration. In *International Symposium on Code Generation and Optimization (CGO'06)*. IEEE Computer Society, 13 pp.–318. https://doi.org/10.1109/CGO.2006.15

[25] Jeremy Lau, Matthew Arnold, Michael Hind, and Brad Calder. 2006. Online Performance Auditing: Using Hot Optimizations without Getting Burned. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Ottawa, Ontario, Canada) *(PLDI '06)*. Association for Computing Machinery, New York, NY, USA, 239–251. https://doi.org/10.1145/1133981.1134010

[26] Hugh Leather and Chris Cummins. 2020. Machine Learning in Compilers: Past, Present and Future. In *2020 Forum for Specification and Design Languages (FDL)*. IEEE Computer Society, 1–8. https://doi.org/10.1109/FDL50818.2020.9232934

[27] David Leopoldseder, Roland Schatz, Lukas Stadler, Manuel Rigger, Thomas Würthinger, and Hanspeter Mössenböck. 2018. Fast-Path Loop Unrolling of Non-Counted Loops to Enable Subsequent Compiler Optimizations. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes* (Linz, Austria) *(ManLang '18)*. Association for Computing Machinery, New York, NY, USA, Article 2, 13 pages. https://doi.org/10.1145/3237009.3237013

[28] David Leopoldseder, Lukas Stadler, Thomas Würthinger, Josef Eisl, Doug Simon, and Hanspeter Mössenböck. 2018. Dominance-Based Duplication Simulation (DBDS): Code Duplication to Enable Compiler Optimizations. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization* (Vienna, Austria) *(CGO 2018)*. Association for Computing Machinery, New York, NY, USA, 126–137. https://doi.org/10.1145/3168811

[29] Shun Long and Michael O'Boyle. 2004. Adaptive Java Optimisation Using Instance-Based Learning. In *Proceedings of the 18th Annual International Conference on Supercomputing* (Malo, France) *(ICS '04)*. Association for Computing Machinery, New York, NY, USA, 237–246. https://doi.org/10.1145/1006209.1006243

[30] Rahim Mammadli, Marija Selakovic, Felix Wolf, and Michael Pradel. 2021. Learning to Make Compiler Optimizations More Effective. In *Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming* (Virtual, Canada) *(MAPS 2021)*. Association for Computing Machinery, New York, NY, USA, 9–20. https://doi.org/10.1145/3460945.3464952

[31] Charith Mendis, Alex Renda, Dr.Saman Amarasinghe, and Michael Carbin. 2019. Ithemal: Accurate, Portable and Fast Basic Block Throughput Estimation using Deep Neural Networks. In *Proceedings of the 36th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, 4505–4515. http://proceedings.mlr.press/v97/mendis19a.html

[32] Raphael Mosaner. 2020. Machine Learning to Ease Understanding of Data Driven Compiler Optimizations. In *Companion Proceedings of the 2020 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity* (Virtual, USA) *(SPLASH Companion 2020)*. Association for Computing Machinery, New York, NY, USA, 4–6. https://doi.org/10.1145/3426430.3429451

[33] Raphael Mosaner, David Leopoldseder, Wolfgang Kisling, Lukas Stadler, and Hanspeter Mössenböck. 2022. Compilation Forking: A Fast and Flexible Way of Generating Data for Compiler-Internal Machine Learning Tasks. *The Art, Science, and Engineering of Programming* 7 (06 2022). https://doi.org/10.22152/

programming-journal.org/2023/7/3

[34] Eunjung Park, John Cavazos, and Marco A. Alvarez. 2012. Using Graph-Based Program Characterization for Predictive Modeling. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization* (San Jose, California) *(CGO '12)*. Association for Computing Machinery, New York, NY, USA, 196–206. https://doi.org/10.1145/2259016.2259042

[35] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019), 8026–8037. https://dl.acm.org/doi/10.5555/3454287.3455008

[36] Filip Pizlo. 2014. JetStream Benchmark Suite. http://browserbench.org/JetStream/ retrieved May 25 2022.

[37] Ricardo Nabinger Sanchez, Jose Nelson Amaral, Duane Szafron, Marius Pirvu, and Mark Stoodley. 2011. Using Machines to Learn Method-Specific Compilation Strategies. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '11)*. IEEE Computer Society, USA, 257–266. https://doi.org/10.1109/CGO.2011.5764693

[38] Andreas Sewe, Mira Mezini, Aibek Sarimbekov, and Walter Binder. 2011. Da Capo Con Scala: Design and Analysis of a Scala Benchmark Suite for the Java Virtual Machine. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Portland, Oregon, USA) *(OOPSLA '11)*. Association for Computing Machinery, New York, NY, USA, 657–676. https://doi.org/10.1145/2048066.2048118

[39] Mark Stephenson and Saman Amarasinghe. 2005. Predicting unroll factors using supervised classification. In *International Symposium on Code Generation and Optimization*. IEEE Computer Society, 123–134. https://doi.org/10.1109/CGO.2005.29

[40] Michele Tartara and Stefano Crespi Reghizzi. 2013. Continuous Learning of Compiler Heuristics. *ACM Trans. Archit. Code Optim.* 9, 4, Article 46 (Jan. 2013), 25 pages. https://doi.org/10.1145/2400682.2400705

[41] Mircea Trofin, Yundi Qian, Eugene Brevdo, Zinan Lin, Krzysztof Choromanski, and David Li. 2021. MLGO: a Machine Learning Guided Compiler Optimizations Framework. *CoRR* abs/2101.04808 (2021). arXiv:2101.04808 https://arxiv.org/abs/2101.04808

[42] Huanting Wang, Zhanyong Tang, Cheng Zhang, Jiaqi Zhao, Chris Cummins, Hugh Leather, and Zheng Wang. 2022. Automating Reinforcement Learning Architecture Design for Code Optimization. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction* (Seoul, South Korea) *(CC 2022)*. Association for Computing Machinery, New York, NY, USA, 129–143. https://doi.org/10.1145/3497776.3517769

[43] Zheng Wang and Michael O'Boyle. 2018. Machine Learning in Compiler Optimization. *Proc. IEEE* 106, 11 (Nov 2018), 1879–1901. https://doi.org/10.1109/JPROC.2018.2817118

[44] Christian Wimmer, Vojin Jovanovic, Erik Eckstein, and Thomas Würthinger. 2017. One Compiler: Deoptimization to Optimized Code. In *Proceedings of the 26th International Conference on Compiler Construction* (Austin, TX, USA) *(CC 2017)*. Association for Computing Machinery, New York, NY, USA, 55–64. https://doi.org/10.1145/3033019.3033025

[45] Christian Wimmer and Thomas Würthinger. 2012. Truffle: A Self-Optimizing Runtime System. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity* (Tucson, Arizona, USA) *(SPLASH '12)*. Association for Computing Machinery, New York, NY, USA, 13–14. https://doi.org/10.1145/2384716.2384723

[46] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. 2017. Practical Partial Evaluation for High-performance Dynamic Language Runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 662–676. https://doi.org/10.1145/3062341.3062381

[47] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to Rule Them All. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (Indianapolis, Indiana, USA) *(Onward! 2013)*. Association for Computing Machinery, New York, NY, USA, 187–204. https://doi.org/10.1145/2509578.2509581

[48] Mingzhou Zhou, Xipeng Shen, Yaoqing Gao, and Graham Yiu. 2014. Space-Efficient Multi-Versioning for Input-Adaptive Feedback-Driven Program Optimizations. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (Portland, Oregon, USA) *(OOPSLA '14)*. Association for Computing Machinery, New York, NY, USA, 763–776. https://doi.org/10.1145/2660193.2660229

Chapter 8

Learned Vector Unrolling

This chapter includes the paper about our case study on learning how to unroll vectorized loops for obtaining the best performance. It revisits the process of assisting compiler engineers during the heuristic design process, as outlined in the initial paper in Chapter 4.

Paper: Raphael Mosaner, Gergö Barany, David Leopoldseder, and Hanspeter Mössenböck. 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* (VMIL 2022). Association for Computing Machinery, New York, NY, USA, 36–47. https://doi.org/10.1145/3563838.3567679

Improving Vectorization Heuristics in a Dynamic Compiler with Machine Learning Models

Raphael Mosaner
raphael.mosaner@jku.at
Johannes Kepler University
Linz, Austria

Gergö Barany
gergo.barany@oracle.com
Oracle Labs
Vienna, Austria

David Leopoldseder
david.leopoldseder@oracle.com
Oracle Labs
Vienna, Austria

Hanspeter Mössenböck
hanspeter.moessenboeck@jku.at
Johannes Kepler University
Linz, Austria

Abstract

Optimizing compilers rely on many hand-crafted heuristics to guide the optimization process. However, the interactions between different optimizations makes their design a difficult task. We propose using machine learning models to either replace such heuristics or to support their development process, for example, by identifying important code features. Especially in static compilation, machine learning has been shown to outperform hand-crafted heuristics. We applied our approach in a state-of-the-art dynamic compiler, the GraalVM compiler. Our models predict an unroll factor for vectorized loops for which the GraalVM compiler developers have not been able to design satisfactory heuristics. Thereby, we identified features to describe vectorized loops and empirically evaluated the impact of different training data, features or model parameters on the accuracy of the learned models. When deployed in the GraalVM dynamic compiler, our models produce significant speedups of 8-11%, on average. Furthermore, large speedups unveiled a performance bug in the compiler which was fixed after our report. Our work shows that machine learning can be used to improve a dynamic compiler directly by replacing existing vectorization heuristics or indirectly by helping compiler developers to design better hand-crafted heuristics.

CCS Concepts: • **General and reference** → *Performance*; • **Software and its engineering** → **Just-in-time compilers**; **Dynamic compilers**; • **Computing methodologies** → **Classification and regression trees**; *Cross-validation.*

Keywords: Dynamic Compilation, Optimization, Heuristics, Loop Vectorization, Unrolling, Performance, Machine Learning, Random Forests

VMIL '22, December 05, 2022, Auckland, New Zealand

ACM Reference Format:
Raphael Mosaner, Gergö Barany, David Leopoldseder, and Hanspeter Mössenböck. 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 (VMIL '22), December 05, 2022, Auckland, New Zealand*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3563838.3567679

1 Introduction

Dynamic compilation [3] can use more input-specific and aggressive optimization strategies than static compilation. This is facilitated by profiling-based speculation [3, 9], where optimization decisions are based on profiling information which is gathered prior to compilation. The GraalVM compiler [33] uses profile-guided optimization (PGO) extensively in its more than 100 compiler phases and in countless optimizations. However, most of these optimizations impact each other which often requires trade-offs. Therefore, compiler heuristics are employed for finding a compilation setup which results in the highest peak performance for compiled functions. Designing theses heuristics is an iterative process, which often takes years of engineering effort and compiler expertise. Some compilation parameters are still set statically, because compiler engineers have not found dynamic heuristics which yield significant improvements.

One example for such a parameter in the GraalVM compiler is the `VectorUnroll` parameter. After the loop vectorization phase, the vectorized loop can be unrolled by using a custom unroll factor (`VectorUnroll`). This global compiler parameter is set to 1 by default, as until now, compiler engineers have not found a satisfying way to set it for each vectorized loop individually, because interactions between vectorization and unrolling are hard to formalize and the resulting code size increase might affect subsequent optimizations in turn. Additionally, heuristics for vectorization strongly depend on the underlying hardware and need to be optimized for each system by hand. Figure 1 summarizes the impact of changing the `VectorUnroll` parameter (globally) from the default value 1 to 8 for the GraalVM

vectorization micro-benchmarks. The histogram groups all 231 benchmarks according to the relative speedup obtained by using `VectorUnroll=8` instead of `VectorUnroll=1`. The

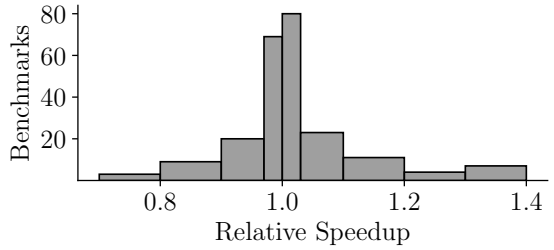


Figure 1. Relative speedup when using `VectorUnroll=8` instead of `VectorUnroll=1` for 231 micro-benchmarks.

x-axis depicts the speedup-bins and the y-axis the number of benchmarks in the respective bin. For example, the bar between 1.1-1.2 shows that about 10 benchmarks encounter speedups of 10-20% when using `VectorUnroll=8` instead of `VectorUnroll=1`. While a large number of benchmarks (>150) show similar performance for both parameter values, one in three benchmarks has a significant speedup or a significant slowdown of more than 3%. Changing the `VectorUnroll` parameter globally benefits certain benchmarks but degrades the performance of other benchmarks in turn. In addition, running the same experiment on another hardware or architecture might produce different results.

We propose a purely data-driven approach, where machine learning is used to model the relationship between vectorized loops and the optimal `VectorUnroll` parameter to use. Such an approach can automatically adapt compiler heuristics to specific inputs or to specific vectorization hardware. It can also support engineers by providing baselines to which human-crafted heuristics under development can be compared to. We evaluated our approach by training and comparing multiple random forest predictors with different training data, feature sets and model parameters.

Our predictive models were created in the context of the GraalVM [33], which is a high-performance, polyglot virtual machine. Its compiler is one of the most highly-optimizing dynamic compilers for the Java ecosystem[1]. The GraalVM compiler uses a graph-based intermediate representation [8, 10] which is based on a sea of nodes. Nodes in the graph are connected by edges that model control flow and data flow. There is a small fixed number of edge types, of which the most important are `Value` (data dependency) and `Memory` (memory access order dependency). Nodes that represent computations which produce a value are associated with a type, e.g., `int32` or `float64`. Nodes can also have associated metadata. For example, `LoopBegin` nodes have an estimated number of loop iterations derived from profiling information. We tested our machine learning models on the vectorization micro-benchmark suite, which is used internally for optimizing vectorization workloads in the GraalVM. Our models outperform the static heuristics which are currently deployed in the GraalVM compiler by 8% - 11% on average, taking into account all benchmarks where significant performance differences of more than 3% were measured. Outliers with speedups of 4–6 × were not included in these averages as they uncovered a performance bug in GraalVM (see Section 6.1). Our research contributes

- a set of features for describing vectorized loops
- a comprehensive evaluation of differently trained machine learning models for predicting unroll factors for vectorized loops
- an integration of trained models in a dynamic compiler that is among the most-highly optimizing Java compilers on the market
- a quantitative experiment which shows that these models outperform existing compiler heuristics by 8% - 11% on average
- a comparison of the deployed models in terms of impacts on compile time and peak performance of executed benchmarks

The remainder of this paper is structured as follows. Section 2 starts by describing how we generated data for training our machine learning models. Section 3 discusses the extracted program features and Section 4 analyzes the generated data and presents pre-processing decisions. Section 5 discusses our model training configurations and compares their impacts on the trained models. Section 6 evaluates the deployed models in the GraalVM compiler in terms of peak performance and compile time.

2 Data Generation

We decided to investigate five values of the `VectorUnroll` parameter for each vectorized loop: 1, 2, 4, 8 and 16, where `VectorUnroll=1` is the default setting in the GraalVM compiler. When using supervised learning, a machine learning model is trained with $\langle feature, label \rangle$ pairs, where features describe the vectorized loop and labels denote the "correct" (most performant) `VectorUnroll` parameter value. In the following, we describe how the required data is created.

To evaluate which parameter value is most beneficial for a vectorized loop, multiple versions must be compiled, executed and measured. Typically, other approaches apply a process called *iterative compilation* [5]. On a high granularity level, the same program is be compiled multiple times with different parameters and the parameter setup which yields the highest peak performance is remembered for future compilations of the program. Similarly, functions can be recompiled with different parameters and their individual performance can be measured [29]. This allows for fine-grained program optimizations, where functions can use different optimization parameters rather than program-wide and static flags. In a dynamic compiler, however, re-compiling individual functions influences other compilations and the program performance as a whole. Dynamically compiling a program

[1] https://renaissance.dev/

multiple times often produces different optimization decisions, due to varying profiles, timings and interleavings of program and compiler threads, which run in parallel. This aggravates measuring and comparing the impact of different `VectorUnroll` parameter values on a single loop. A recently established technique, called *compilation forking* [23], addresses these problems by creating and executing multiple versions of the same function in a single program execution. Compilation forking creates these versions based on a common compilation history and with identical profiles. This ensures that until creating versions for different *VectorUnroll* values, every compilation decision has been made in exactly the same way. The different versions are then executed alternatingly, to cancel out measurement noise and different usages of the functions in the long run.

2.1 Compilation Forking

We implemented compilation forking in the GraalVM compiler phase which handles loop vectorization. A function can have multiple vectorized loops L, which, together with the number of selected versions for the `VectorUnroll` parameter (5), would result in $5^{|L|}$ versions, where $|L|$ is the number of vectorized loops in the function. However, since vectorized loops can be considered independent from each other, the number of versions can be reduced to grow linearly with the number of vectorized loops. This is shown in

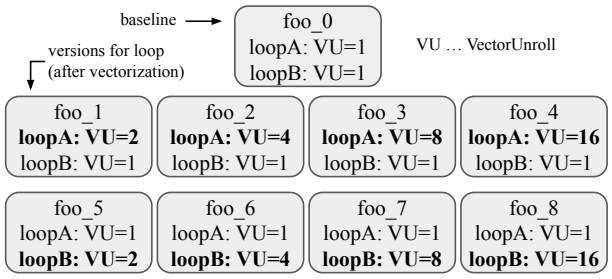


Figure 2. Compilation forking [23] with two loops.

Figure 2 for a function `foo` with two vectorized loops. Each function has a baseline version where all vectorized loops use `VectorUnroll=1`. Then, for each vectorized loop, four additional versions are created—one for each `VectorUnroll` ϵ {2, 4, 8, 16}—where all other vectorized loops use the baseline configuration (`VectorUnroll=1`). The total number of versions is $1 + 4 * |L|$. This would result in 9 versions of a function with two vectorized loops rather than 25. All versions are compiled independently starting from the common past after forking. Additionally, each version is instrumented to measure the number of executions and the total execution time. After the compilation and instrumentation of all versions has finished, they are recombined into one function. This includes an internal dispatch to execute these versions alternatingly upon invocation, which happens transparently to the caller. Executing versions alternatingly helps averaging out noisy measurements in the long run. During the program execution, the performance measurements for each version are aggregated as shown in Table 1.

Table 1. Extracted performance measurements for each vectorized loop (*VectorID*) in a compiled function (*CompID*)

CompID	VectorID	$Time_1$	$Invo_1$	$Time_2$	$Invo_2$	$Time_4$	$Invo_4$	…
Comp-1234	Vec-1	882330	336	624341	336	596885	335	…
Comp-1234	Vec-2	882330	335	872432	335	885231	335	…
Comp-6789	Vec-1	308261	721	71901	721	71821	721	…

2.2 Observer Effect

Compilation forking [23] creates comparable performance measurements for different optimization decisions in a dynamic compiler. Nevertheless, an observer effect can still occur and change how the program as a whole is compiled. For example, functions take longer to compile when compilation forking is enabled. This can impact the compilation order and in further consequence profiles or even inlining decisions. The created versions are consistent, as they correctly reflect the impact of the changed `VectorUnroll` parameter, but might not be compiled in the same way as without forking. The problem may become apparent after deployment but only if the encountered data is inherently different from the training data because forking is disabled.

2.3 Feature Extraction

In GraalVM, vectorized loops are processed one after another. Unrolling a vectorized loop changes "global" features—such as the number of total nodes in the graph representation of the function—for subsequent loops. Hence, the order in which otherwise identical vectorized loops are unrolled could interfere with the extracted features. Thus, we extract the features from a snapshot of the compilation before any vectorized loop is unrolled. This process is identical during data generation and when using a deployed model in the compiler.

2.4 Collected Data

We collected feature and performance data from 231 microbenchmarks which were designed to cover different use cases of vectorization and therefore provided measurable differences when changing vectorization parameters. These benchmarks have been used to design and improve vectorization-related heuristics in the GraalVM. We collect data from five executions of each benchmark, to account for differences due to dynamic compilation and measurement noise. In total, 142699 data points were collected which corresponds to 142699 vectorized loops. Each data point is described by 190 program features and holds the performance measurements for all different `VectorUnroll` values used during compilation. The actual number of data points and features used for training is discussed in Section 5.

3 Features

In a machine learning context, features describe the environment for which a prediction is made. We use several

features to describe the state of an intermediate compilation at the point where a loop is vectorized. As the GraalVM compiler uses a graph-based intermediate representation (GraalIR) [8, 10], most features are graph-related. Listing 1 shows a function `bar`, which maps negative values of an input array to zero and keeps positive values. The results are written to an array `res`, which is returned in the end.

```
1 double[] bar(double[] src, double[] res) {
2   for (int i = 0; i < result.length; i++) {
3     res[i] = Math.max(src[i], 0D);
4   }
5   return result;
6 }
```

Listing 1. Vector example.

When compiling this code, the opportunity of vectorization arises, which results in the following features.

3.1 Vector Operation Features

Vectorized loops have a main vector operation kind, which is a vectorized *write* in case of Listing 1. The main kinds are *map* (transform arrays element-by-element), *fold* (reduce arrays to a scalar with an arithmetic operation), and *write* (write a vector to memory). Thus, the *VectorKind* feature is a categorical feature with a small number of possible values. Vector operations also have a data type which includes a bit width. As the bit width only lies within a small set of values, we combined the type and its bit width by creating a categorical feature *VectorType* with values $\{void\} \cup \{float, int\} x \{16, 32, 64\}$. The third feature, which is given for any vector operation, is the average number of loop iterations *LoopIterations* which was measured during profiling.

3.2 Vector Component Features

Vector operations in the GraalVM compiler can consist of a varying number of components. As the majority of machine learning models takes a fixed number of features as input, we decided to aggregate information across such components. As the computation inside the loop may involve arbitrarily many operations, we use histograms to capture information about these operations and their dependencies.

VectorInputs. For the *vectorInput* histogram all occurrences of [*VectorKind* × *VectorType*] are counted recursively. In the example loop, the operation which precedes the vectorized *write* is a mapping from `src` array elements to `res` array elements which is represented by a node with *VectorKind* `VectorMap` and *VectorType* `float_64`. These features are similar to the vector operation counts presented by Stock et al. [31], but with added type information. As the number of vector operation kinds is small, adding the type information does not increase the number of features dramatically.

Vector arithmetic. Arithmetic operations inside the vectorized loop are encoded as features by creating a *VectorNodes* histogram. The `Math.max` operation in the example loop is represented by a `Max` node. Type information for nodes is not as important as for vector operations. Therefore, the number of features is reduced by storing only the node kinds without type information. Similarly to the node information, the *VectorEdges* histogram stores all edge type counts.

VectorGroups. For loops which contain multiple vectorizable operations that do not depend on each other, the GraalVM compiler creates a 'vector group'. Such vector operations inside a group need to be processed together, albeit being potentially unrelated. We aggregate all grouped vector operations inside a group in a *VectorGroup* histogram.

3.3 Feature Summary

Table 2 summarizes all features which were available after data collection. Column *Count* contains the number of features for histograms. In addition to vector operation and vector component features, we captured histograms for all node types (*GraphNode* histogram) and edge types (*GraphEdge* histogram) of the whole graph the vectorized loop is part of. These features give an estimate of the complexity of the function as a whole which also correlated with the resulting code size [24]. Values that are not encountered during data collection are not added to the histograms. For example, 109 different node types were encountered in all benchmarks, but within vector arithmetic only 36 node types were found. While the total number of extracted features is 190, the actual number of features used in model training might be reduced as discussed in Section 5.

Table 2. Extracted features.

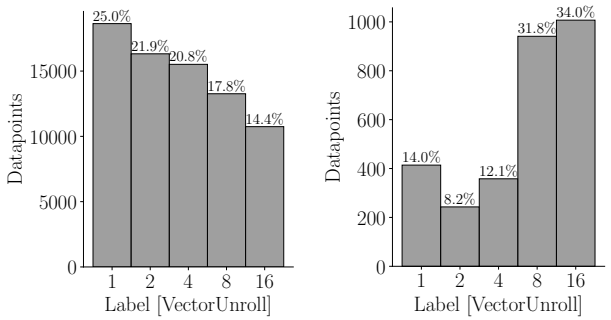
Feature(s)	Type	Count	Example
VectorKind	cat	1	"vectorWrite"
VectorType	cat	1	"int_32"
LoopIterations	float	1	1024.125
VectorInputs	hist	25	"MapVectorNode_int16: 2"
VectorNodes	hist	36	"MaxNode: 1"
VectorEdges	hist	2	"Value: 2"
VectorGroup	hist	3	"FoldVectorNode_int64"
GraphNodes	hist	109	"AddNode: 7"
GraphEdges	hist	7	"Association: 32"
GraphAggregates	hist	5	"FixedNodes: 175"

4 Data Analysis and Pre-processing

Our initial data set consisted of 142699 data points and 190 feature values collected from 231 benchmarks which were executed 5 times each. We analyzed the generated data and selected pre-processing steps to improve the data quality.

Labels. In order to train a classification model, the labels (i.e. "correct" `VectorUnroll` parameters) for each data point need to be derived from the performance measurements. The labels are calculated from the average execution times as

$$label = \underset{i \in \{1,2,4,8,16\}}{\arg\min} \left(\frac{totalTime_i}{invocations_i} \right)$$



(a) Including standard library functions. **(b)** Excluding standard library functions.

Figure 3. Distribution of the extracted most beneficial `VectorUnroll` parameter in the benchmarks.

which calculates the unroll factor yielding the lowest average execution time. If the difference of average speedups for different values of `VectorUnroll` is below the level of measurement noise, data points described by similar features can have different labels assigned. To reduce this phenomenon, we excluded measurements where the average execution time is very small or the number of invocations is below 100. This reduced the number of data points to 74485, which is about 52% of the initial data size.

Feature Reduction. We built sparse histograms, meaning that they only contain features which are encountered in at least one data point in the generated data. For example, from about 500 different nodes in the GraalIR only 109 nodes are found during the vectorization phase in one of our benchmarks. To reduce the number of features even further, we removed features which are equal for at least 99% of the data points. This heuristic reduced the number of histogram features to 139 (see Table 3) or 142 features in total.

Table 3. Histograms after feature reduction.

Histogram Features	before filtering	after filtering
VectorInputs	25	16
VectorNodes	36	14
VectorEdges	2	2
VectorGroup	3	2
GraphNodes	109	94
GraphEdges	7	7
GraphAggregates	5	4

Data from Standard Libraries. Most programs use standard library functions, such as `java.util.ArrayList.add`. This can lead to similar data points across different benchmarks. While the profiling information gathered prior to compilation often causes differences between the same library functions used in different benchmarks, we analyzed the impact of removing standard library functions from the data. Figure 3 shows the distribution of labels for all data (Figure 3a) and for the data with standard library functions removed (Figure 3b). In both cases, the previously mentioned filtering has already been applied to the data. As expected, there are significant differences in the distributions. When taking a look at Figure 3a, which summarizes all data including standard library functions, a tilt towards lower `VectorUnroll` values can be seen but the distribution as a whole is fairly balanced. When removing the data points stemming from library functions, the distribution becomes highly imbalanced in favor of large values for `VectorUnroll`. This imbalance can be explained by the nature of the micro-benchmarks which are designed with long-running loops to be optimized with vectorization. We decided to analyze both configurations. In one configuration, we trained and deployed our models to work with all data. In the other configuration, we trained our models without the standard library data and used the models only when compiling non-standard library data.

5 Model Training

After initial experiments we decided against using deep-learning algorithms due to the small data set and its expected low versatility. Instead, we employed random forests [15], which build multiple decision trees and combine their local outputs to derive its global decision. In case of a classification problem, this global decision is a majority vote over all decision trees which outputs the most frequently predicted class. Decision trees have also been used by Kulkarni et al. [16] as their human-readable format allows to derive information about important features and also Papadimitriou et al. [26] recently chose a tree-based learning algorithm in their work.

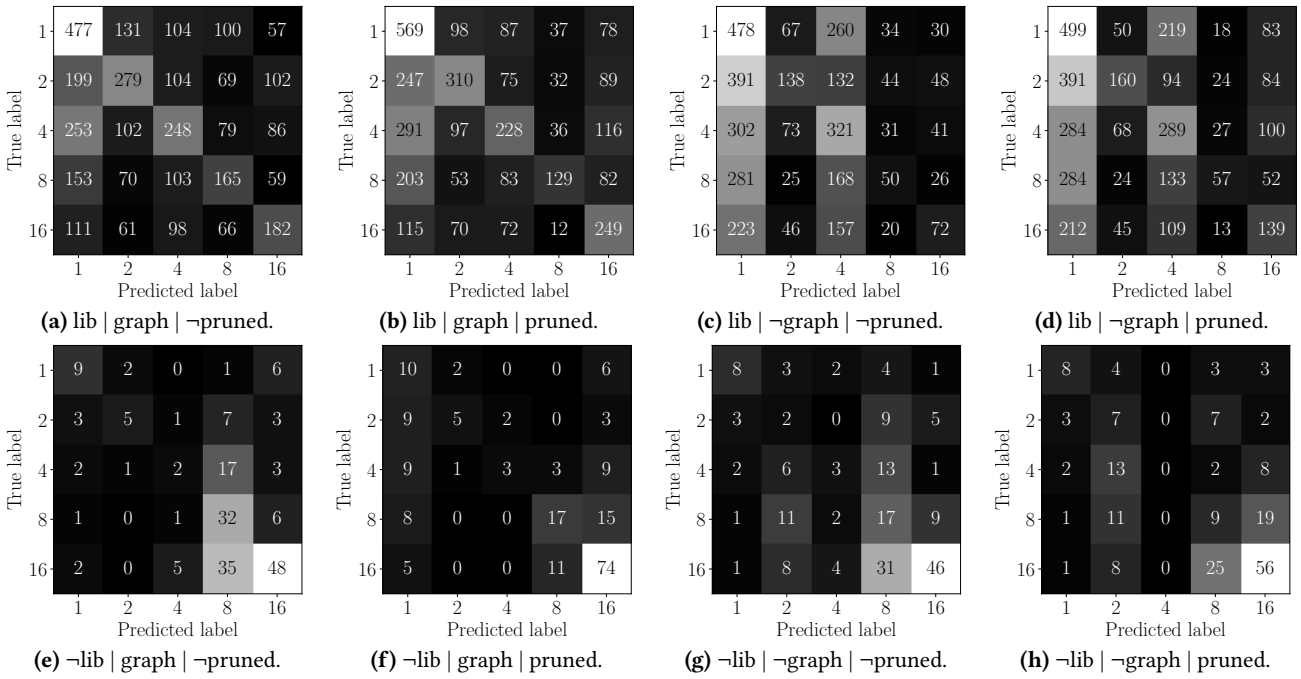
5.1 Training Setup

We conducted a cross-validation to distinct training data and test data on benchmark level. For the training setup we randomly split the set of benchmarks B into n groups $B_0, B_1, ..., B_n$ of 10 to 11 benchmarks. Each of these groups is used as test data for one sub-model $M_0, M_1, ..., M_n$. Sub-model M_i is trained with the data from benchmarks $B \backslash B_i$ and tested on the data of B_i. This way, there is no overlap of training and test data on the level of benchmarks. On the level of functions, there can still be overlaps for standard library methods, if they are compiled in a similar way.

5.2 Training Configurations

We evaluated the impact of three disjoint configuration parameters each addressing one dimension in the training.

Data: Standard Library Functions. Figure 3 shows that the label distribution differs between the whole data and benchmark-specific data without standard library functions. We therefore compare models which were trained and tested on data which either includes or excludes the library data. Due to our cross-validation setup, the number of data points which were used for training and testing differ for each sub-model. Including the library data yields 71245 data points for training and 3237 data points for testing, on average. Excluding the library data reduces the number of data points for training to 2798 and for testing to 163, on average.



(a) lib | graph | ¬pruned. **(b)** lib | graph | pruned. **(c)** lib | ¬graph | ¬pruned. **(d)** lib | ¬graph | pruned.

(e) ¬lib | graph | ¬pruned. **(f)** ¬lib | graph | pruned. **(g)** ¬lib | ¬graph | ¬pruned. **(h)** ¬lib | ¬graph | pruned.

Figure 4. Confusion matrices for one sub-model for each configuration.
lib ... includes standard library data | graph ... includes graph features | pruned ... random forest pruned to depth 10

Features: Graph Features. After feature reduction, 37 features describe the vectorized loop, and 105 features describe the compilation graph as a whole. To evaluate the impact of these graph features, we used configurations with all (142) features and ones without the graph features.

Model: Tree Pruning. Overfitting and generalization are crucial aspects during model training. Overfitting is a phenomenon where the training data with all its peculiarities (and noise) is perfectly captured by the model. This gives good prediction accuracy on the training data but at the cost of bad performance on the test data. Due to overfitting, the generalization to new data can degrade. The default setting when training random forests allows decision trees of arbitrary depth, which can cause overfitting. Therefore, we added a configuration setup where we pruned the decision trees to a maximum depth of 10. This leads to omitting data splits in the deeper levels of the tree which are more likely to be based on training data peculiarities.

5.3 Training Results

The above configuration parameters lead to 8 (= 2 x 2 x 2) configurations in total. For each configuration a set of random forests—each with 100 trees—was trained, according to our cross-validation setup (see Section 5.1). The sub-models were then tested with the data from the benchmarks which were excluded from the training. Figure 4 shows the confusion matrices for the first sub-model from each configuration. The x-axis shows the predicted value whereas the y-axis the actual label. The main diagonal holds the correct predictions. Sub-models (a) and (b) show especially good accuracy over all classes, however, with a stronger tendency to predict `VectorUnroll=1` than in the distribution of the training data (Figure 3a). When removing the graph features for the training, a further shift towards smaller values for `VectorUnroll` can be seen for (c) and (d). This is an indicator that the size or complexity of the graph influences the prediction and pushes it towards larger unroll factors. For sub-models (e)-(h) which were trained and tested without the data from standard library functions, `VectorUnroll=8` and `VectorUnroll=16` are predicted most of the time. Many predictions are correct, or just off-by-one, which correlates with being the second best label.

Table 4 summarizes the trained models for each configuration, where all values correspond to geometric means across all sub-models. *Pred 1* to *Pred 16* summarize how often the particular class was predicted for the test data. The best performance when using the model on the *training* data is achieved when using the graph features and without pruning the trees. Without the library functions, the model can be overfitted to the training data with an accuracy of over 90%. When the library functions are included, the overfitted model has below 60% accuracy. This can be explained by contradicting data, where similar or even identical data points are labeled differently. Such data is especially found in the standard library data, where different `VectorUnroll` values often produce similar performance. For all configurations, the accuracy on the test data can be improved by pruning the trees. This is expected, as pruning counters overfitting.

Table 4. Summary of trained models. *Pred 1* to *Pred 16* summarize the distribution of predicted classes for the test data. All values are geometric means across all sub-models trained for cross-validation.

Model Config			Features	Data train	Data test	ACC train	ACC test	Pred 1 test	Pred 2 test	Pred 4 test	Pred 8 test	Pred16 test
with library data	graph features	large RF	142	71245	3237	**59.2%**	39.7%	35.2%	17.9%	19.3%	12.8%	14.8%
		pruned	142	71245	3237	43.9%	**42.7%**	40.6%	17.4%	16.0%	7.6%	18.4%
	no graph features	large RF	37	71245	3237	41.5%	31.0%	49.0%	9.8%	29.0%	6.0%	6.0%
		pruned	37	71245	3237	34.2%	33.1%	48.1%	12.4%	24.2%	4.4%	10.6%
without library data	graph features	large RF	142	2798	163	**90.5%**	50.1%	14.9%	1.7%	7.5%	32.9%	33.5%
		pruned	142	2798	163	73.5%	**53.6%**	14.9%	1.3%	2.2%	27.2%	43.7%
	no graph features	large RF	37	2798	163	82.4%	46.7%	11.0%	3.9%	4.5%	33.5%	27.9%
		pruned	37	2798	163	66.0%	50.5%	8.2%	0.1%	0.3%	26.9%	49.1%

Using the graph features improves the accuracy on the training data significantly. As the accuracy for the test data is improved as well, it is indicated that the graph features do not solely contribute to overfitting.

The distributions of the predictions for the test data show a tendency of amplifying the underlying training data distribution the more measures against overfitting are in place. Forests which use the graph features and are not pruned predict similarly to the label distribution in the training data, shown in Figure 3. This lets us assume that even after splitting the data correctly into training and test data, many data points may be similar across these splits: We can reason that the benchmarks are written in similar ways. When graph features are omitted and trees are pruned, more frequent classes in the training data are predicted with even higher frequency. This can especially be seen for the models which are not using the library data. There, more than 75% of the test data either produces `VectorUnroll=8` or `VectorUnroll=16` as output. The underlying distribution in the training data (Figure 3b) has only 66% in those two classes.

For both data sets, the best models in terms of test data accuracy are obtained when using the graph features and pruning the trees. However, the accuracy of the models does not reflect the performance of the model when deployed. The training data labels are based on measurements. Therefore, the label for vectorized loops where unrolling does not have any impact are determined only by the measurement noise. Thus, the underlying model cannot learn a correct way to predict these data points and has likely a low accuracy. However, "wrong" predictions for such data have no impact when deploying the model in the compiler.

5.4 Learning from Models

An additional application of machine learning in compilers is to support compiler engineers when designing heuristics. When using decision trees or random forests, the data is classified in a human-readable way. In their work on predicting inlining decisions, Kulkarni et al. [16] transformed their neural network to a decision tree to infer knowledge about the model's decision making. Figure 5 shows a snippet from a single decision tree taken from one of our random forest classifiers. The first node contains all data points and

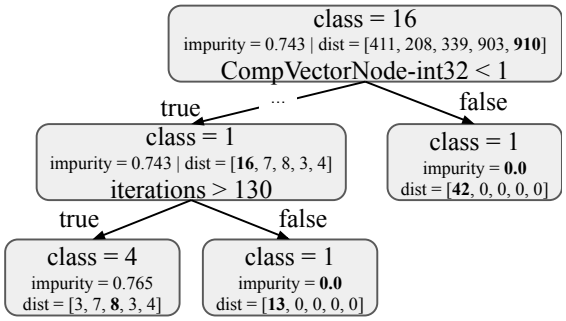


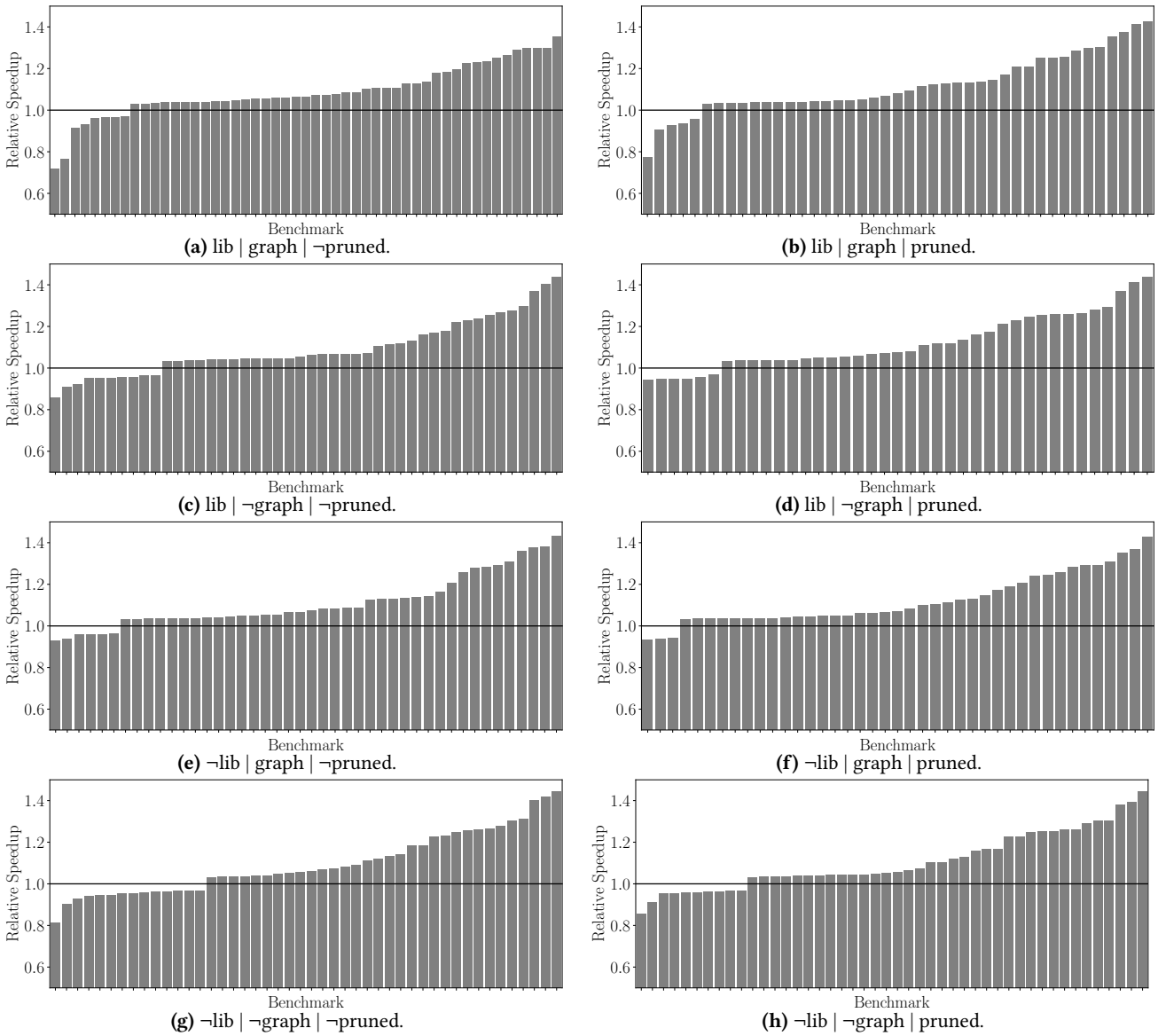
Figure 5. Decision tree snippet.

if not further subdivided would predict `VectorUnroll=16` for every input, because this is the most frequent class in the node's distribution. A lower impurity means that more data points share the same class. The most information can be inferred from nodes with an impurity of 0 at shallow levels of the tree. For example, from the first split we can infer that the presence of `CompareVectorNodes` indicates that unrolling is not beneficial. Similarly, in a deeper nesting level, vectorized loops with less than 130 iterations should best use `VectorUnroll=1`. Across all random forests, the loop iteration feature has the highest importance. While the actual cut-off values may be of less relevance, important features and their rough value ranges can be used when designing heuristics, especially when aggregated across all decision trees of a random forest.

6 Evaluation

We have shown the performance of the trained models in terms of prediction accuracy on the test data. However, the initial hypothesis, that predicting the `VectorUnroll` parameter can improve the current static heuristics in the GraalVM, needs to be tested by deploying our models in the compiler.

Experimental Setup. We used the OnnxRuntime [21] for Java to load the previously trained models into the GraalVM compiler. Furthermore, we configured our benchmark harness to use sub-model M_i which was trained with benchmarks $B \backslash B_i$ when executing benchmarks B_i, according to



(a) lib | graph | ¬pruned. **(b)** lib | graph | pruned.
(c) lib | ¬graph | ¬pruned. **(d)** lib | ¬graph | pruned.
(e) ¬lib | graph | ¬pruned. **(f)** ¬lib | graph | pruned.
(g) ¬lib | ¬graph | ¬pruned. **(h)** ¬lib | ¬graph | pruned.

Figure 6. Relative speedup compared to default GraalVM. Higher is better. Each tick on the x-axis represents one benchmark, the y-axis the corresponding speedup (>1) or slowdown (<1). Benchmarks with speedups or slowdowns <3% are not shown.

our cross-validation setup. Models which were trained without the standard library data are only used when compiling non-standard library methods. Each benchmark was executed 5 times and the results are summarized as geometric means. We evaluated our approach in terms of peak performance, compile time and disk space requirements. Table 5 summarizes the geometric means for these metrics.

6.1 Peak Performance

In accordance to Figure 1, most benchmarks do not show measurable differences in terms of peak performance when changing the `VectorUnroll` parameter. We decided to remove benchmarks where the peak performance impact is

Table 5. Geometric means for speedup (higher is better) and compile time (lower is better) normalized to default GraalVM.

Model Config			Speedup	Compile Time	Size [MB]
with library data	graph features	large RF	1.081	2.386	118.9
		pruned	1.109	1.103	4.2
	no graph features	large RF	1.087	1.726	64.1
		pruned	1.117	1.078	3.9
without library data	graph features	large RF	1.106	1.072	9.7
		pruned	1.117	1.078	2.6
	no graph features	large RF	1.086	1.062	7.8
		pruned	1.099	1.070	1.8

less than 3% from the plots and from the speedup means in Table 5. Figure 6 summarizes the remaining 40-60 benchmarks, where significant impacts were encountered, for each model. The benchmark results for each model are sorted according to the relative speedup compared to the default GraalVM compiler. Each tick on the x-axis corresponds to one benchmark and the y-axis depicts the relative speedup normalized to the baseline. Higher is better.

All models significantly improve the average peak performance compared to the GraalVM compiler without machine learning. Models (a) and (b) which were trained on all data and use the graph features show the largest slowdowns on single benchmarks. Removing the graph features ((c) and (d)) removes large negative outliers. Further pruning the trees (model (d)) has the best results with 11.7% average speedup for the plotted benchmarks. For models (e) to (h) which were trained and executed on non-standard library data, removing the graph features (models (g) and (h)) has larger negative outliers and worse results. Table 5 indicates that pruning the trees produces consistently better results. Using the graph features produces mixed results, depending on the data set used for training and testing. In total, all models achieve similar performance results. As models (e) to (h) are not executed for standard library functions, it is evident that most speedups in the (micro-)benchmarks were from the custom benchmark functions rather than standard library functions. This indicates that models (a)–(d), albeit being trained with data stemming to more than 95% from standard library functions, correctly predict the most important data points which were seen less frequently during training.

Performance Bug. Our machine learning models found a performance issue in the GraalVM compiler. Certain vectorized loops contained redundant pairs of integer narrow/extend operations. The compiler only recognized these as redundant when the vectorized loop was unrolled. This caused an artificial improvement through vector unrolling of about 4–6 × on 6 benchmarks. This issue was fixed after our report, and the compiler no longer generates the redundant operations. We removed these benchmarks in the plots and the summary in Table 5 to avoid distorting the evaluation.

6.2 Compile Time and Model Size

Table 5 reports the total compile time impact when using models during compilation. Without pruning the models, significant compile time increases by factors of 1.7 to 2.4 are encountered. We found that the overall compile time increase strongly correlates with the size of the model. Table 7 shows a more nuanced analysis of the compile time.

Column *Model Loading* contains the relative portion of compile time which is spent for model loading. The model loading for the largest model consumes 60.8% of the total compile time, which makes model loading the most expensive sub-task. Models are loaded and stored once for each compiler thread. Therefore, the impact of model loading is significant for short running benchmarks. However, models with smaller size, either because less data was used for training or because of pruning are loaded significantly faster.

Column *Decision* contains the relative portion of compile time which is spent for collecting features and invoking the model. It can take up to 3.1% of the total compile time. Column *Prediction* shows that only up to 0.5% of the total compile time is spent for invoking the model. The prediction time makes up at most 21.6% of the decision time (column *Prediction / Decision*) which shows that feature extraction is far more expensive than invoking the model. The last column shows the decision time (feature extraction + model invocation) in comparison to the average compile time of executing benchmarks without any models in place. Except for model loading, the compile time increases are moderate with 2-3.4% in comparison to the default GraalVM compiler without our models. Models which are not executed on standard library data produce a much smaller overhead.

6.3 DaCapo and Scala-DaCapo

We decided to evaluate our approach on (micro-)benchmarks specifically designed for benchmarking vectorization. This way, the impact of changed vectorization heuristic is clearly measurable and less side effects of dynamic compilation can interfere with the measurements. However, we also tested two of our models on very different data from well-known benchmark suites. Both models are trained including the standard library data and use pruned trees. One model excludes the graph features and the other uses them. Table 6 summarizes the speedup and compile time for the DaCapo and Scala-DaCapo suites. The average speedup is slightly worse than the default GraalVM configuration. However, most speedups for single benchmarks are below measurement noise. This is, because only small portions of these benchmarks contain long running, vectorizable loops which benefit most from unrolling. For DaCapo, only the *luindex* benchmark showed a significant slowdown of 4.5% for the model including the graph features. For Scala-DaCapo, the *scalatest* benchmark exhibits a speedup of 7.8% for both models. However, for the model without graph features *kiama*, *scaladoc* and *scalaxb* also exhibit slowdowns of 3.3-4.8%.

Table 6. DaCapo and Scala-DaCapo performance.

Suite	Graph Features	Speedup	Compile Time
DaCapo	yes	0.993	1.184
DaCapo	no	0.996	1.219
Scala-DaCapo	yes	0.995	1.187
Scala-DaCapo	no	0.991	1.210

Despite being trained only with data from Java micro-benchmarks, our models perform similarly to the highly tuned GraalVM compiler on larger benchmarks. We could also conclude, that our models can not perfectly generalize to the vastly different data from only little (micro-)benchmark

Table 7. Model impact on compile time

Model Config			Model Loading	Decision	Prediction	Prediction / Decision	Decision / CompTime (default)
with library data	graph features	large RF	60.8%	1.1%	0.2%	18.0%	2.7%
		pruned	5.5%	3.1%	0.5%	16.9%	3.4%
	no graph features	large RF	46.1%	1.2%	0.3%	21.6%	2.0%
		pruned	5.5%	2.2%	0.5%	21.5%	2.4%
without library data	graph features	large RF	9.6%	0.4%	0.1%	14.3%	0.4%
		pruned	4.1%	0.4%	0.1%	15.6%	0.4%
	no graph features	large RF	8.1%	0.3%	0.1%	18.9%	0.3%
		pruned	2.9%	0.3%	0.1%	21.3%	0.3%

data. This is especially expected for the the Scala-DaCapo suite, as Scala code produces significantly different features when being compiled in contrast to Java code which is used in the (micro-)benchmarks.

7 Related Work

Loop vectorization was introduced by Allen and Kennedy [1]. More versatile vectorization strategies such as superword-level parallelism (SLP) [17] and auto-vectorization [25] could not replace traditional loop vectorization [7]. This fostered recent research analyzing the impact of loop unrolling [4] on vectorization [28] or trying to integrate control-flow-based (i.e. loop-based) vectorization within SLP vectorization [7]. Rocha et al. [28] present *Vectorization-Aware Loop Unrolling* where loop unrolling is performed in such a way that it benefits vectorization, which is traditionally performed later during compilation. This is done by examining the vectorization potential at the time of unrolling and selecting the unroll factor accordingly. Furthermore, they forward hints to the vectorizer which override the default behavior and unveil more vectorization opportunities. Our approach addresses the interplay between unrolling and vectorization in reverse order. Based on how a loop has been vectorized, a suitable unroll factor is chosen.

There is extensive research in the domain of machine learning for compilers [2, 32]. A number of works also addresses using machine learning for either vectorization [14, 20, 31] or unrolling [22, 30]. For example, Haj-Ali et al. [14] recently presented *NeuroVectorizer*, which is an end-to-end tool for LLVM. It is able to learn the number of loop iterations which need to be packed into vector operations, i.e. the vectorization factor (VF). *NeuroVectorizer* automatically extracts loop features from source code, selects a VF and compiles the functions using clang. The speedup compared to the last measured configuration is used as a reward for training a model with deep reinforcement learning. They are able to achieve speedups between factors of 1.29 and 4.73 and get close to the performance of a brute force search over all parameter combinations. This is interesting, as their approach (and the baseline in LLVM) lacks profiling information of how often loops are assumed to be iterated. In contrast to their work for a static compiler, our approach targets dynamic compilation where data generation and performance measurements are more noisy and less reproducible, which requires an adaptive approach.

Papadimitriou et al. [26] used machine learning in a concurrent execution environment based on the GraalVM to predict where certain tasks should be executed, i.e. CPU, GPU or IGPU. They use a combination of static features and profiling information to train a tree-based model, which, when deployed, yields performance results close to the optimum, which is obtained by exhaustive state space exploration.

During data generation, we employed compilation forking [23] (see Section 2.1, which is related to iterative compilation [5, 12, 13, 19] and multi-versioning [11, 18, 34], but takes the previous compilation history into account. Iterative compilation and multi-versioning typically aim to create globally optimal compilations while we create versions to derive knowledge about local optimization decisions.

While we used traditional histogram-based features as our models, graph-based feature representations have also been used in related work [6, 27]. However, Brauckmann et al. [6] concluded that no single best feature representation exists.

8 Conclusion

We have presented an approach to determine a vectorization parameter in a dynamic compiler with machine learning models. These models were able to produce speedups of 8-11% on a benchmark suite made for evaluating the vectorization in the GraalVM compiler. Significant speedups of factors between 4 and 6 unveiled a compiler performance bug which our approach helped to fix. Building on that, we want to further analyze the generated decision trees to infer improvements for existing heuristics. In addition, in future work we want to analyze how well our models generalize on vastly different data and whether algorithms such as *Extra-Trees*, as used in [26], affect overfitting. Furthermore, as our approach is applicable to other vectorization related optimizations, we plan to predict other parameter, such as vector size or *AVX* versions dynamically for vectorized loops.

References

[1] Randy Allen and Ken Kennedy. 1987. Automatic Translation of FORTRAN Programs to Vector Form. *ACM Trans. Program. Lang. Syst.* 9, 4 (oct 1987), 491–542. https://doi.org/10.1145/29873.29875

[2] Amir H. Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. 2018. A Survey on Compiler Autotuning Using Machine Learning. *ACM Comput. Surv.* 51, 5, Article 96 (Sept. 2018), 42 pages. https://doi.org/10.1145/3197978

[3] Joel Auslander, Matthai Philipose, Craig Chambers, Susan J. Eggers, and Brian N. Bershad. 1996. Fast, Effective Dynamic Compilation. *SIGPLAN Not.* 31, 5 (may 1996), 149–159. https://doi.org/10.1145/249069.231409

[4] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. 1994. Compiler Transformations for High-Performance Computing. *ACM Comput. Surv.* 26, 4 (Dec. 1994), 345–420. https://doi.org/10.1145/197405.197406

[5] François Bodin, Toru Kisuki, Peter Knijnenburg, Mike OBoyle, and Erven Rohou. 1998. Iterative compilation in a non-linear optimisation space. *Workshop on Profile and Feedback-Directed Compilation* (03 1998). https://hal.inria.fr/inria-00475919/document

[6] Alexander Brauckmann, Andrés Goens, Sebastian Ertel, and Jeronimo Castrillon. 2020. Compiler-Based Graph Representations for Deep Learning Models of Code. In *Proceedings of the 29th International Conference on Compiler Construction* (San Diego, CA, USA) *(CC 2020)*. Association for Computing Machinery, New York, NY, USA, 201–211. https://doi.org/10.1145/3377555.3377894

[7] Yishen Chen, Charith Mendis, and Saman Amarasinghe. 2022. All You Need is Superword-Level Parallelism: Systematic Control-Flow Vectorization with SLP. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 301–315. https://doi.org/10.1145/3519939.3523701

[8] Gilles Duboscq, Lukas Stadler, Thomas Würthinger, Doug Simon, Christian Wimmer, and Hanspeter Mössenböck. 2013. Graal IR: An Extensible Declarative Intermediate Representation. In *Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop*. 1–9. https://ssw.jku.at/General/Staff/GD/APPLC-2013-paper_12.pdf

[9] Gilles Duboscq, Thomas Würthinger, and Hanspeter Mössenböck. 2014. Speculation without Regret: Reducing Deoptimization Meta-Data in the Graal Compiler. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools* (Cracow, Poland) *(PPPJ '14)*. Association for Computing Machinery, New York, NY, USA, 187–193. https://doi.org/10.1145/2647508.2647521

[10] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. 2013. An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler. In *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages* (Indianapolis, Indiana, USA) *(VMIL '13)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/2542142.2542143

[11] Peng fei Chuang, Howard Chen, Gerolf F. Hoflehner, Daniel M. Lavery, and Wei chung Hsu. 2007. Dynamic profile driven code version selection. In *the 11th Annual Workshop on the Interaction between Compilers and Computer Architecture*. https://www.researchgate.net/publication/228952289_Dynamic_Profile_Driven_Code_Version_Selection

[12] Grigori Fursin, Albert Cohen, Michael O'Boyle, and Olivier Temam. 2005. A Practical Method for Quickly Evaluating Program Optimizations. In *High Performance Embedded Architectures and Compilers*, Nacho Conte, Tomband Navarro, Wen-mei W. Hwu, Mateo Valero, and Theo Ungerer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 29–46. https://doi.org/10.1007/11587514_4

[13] Grigori Fursin, Cupertino Miranda, Olivier Temam, Mircea Namolaru, Ayal Zaks, Bilha Mendelson, Edwin Bonilla, John Thomson, Hugh Leather, Chris Williams, Michael O'Boyle, Phil Barnard, Elton Ashton, Eric Courtois, and François Bodin. 2008. MILEPOST GCC: machine learning based research compiler. In *Proceedings of the GCC Developers' Summit 2008*. https://hal.inria.fr/inria-00294704

[14] Ameer Haj-Ali, Nesreen K. Ahmed, Ted Willke, Yakun Sophia Shao, Krste Asanovic, and Ion Stoica. 2020. NeuroVectorizer: End-to-End Vectorization with Deep Reinforcement Learning. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization* (San Diego, CA, USA) *(CGO 2020)*. Association for Computing Machinery, New York, NY, USA, 242–255. https://doi.org/10.1145/3368826.3377928

[15] Tin Kam Ho. 1995. Random decision forests. In *Proceedings of 3rd international conference on document analysis and recognition*, Vol. 1. IEEE, 278–282.

[16] Sameer Kulkarni, John Cavazos, Christian Wimmer, and Douglas Simon. 2013. Automatic Construction of Inlining Heuristics Using Machine Learning. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO '13)*. IEEE Computer Society, Washington, DC, USA, 1–12. https://doi.org/10.1109/CGO.2013.6495004

[17] Samuel Larsen and Saman Amarasinghe. 2000. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. *SIGPLAN Not.* 35, 5 (may 2000), 145–156. https://doi.org/10.1145/358438.349320

[18] Jeremy Lau, Matthew Arnold, Michael Hind, and Brad Calder. 2006. Online Performance Auditing: Using Hot Optimizations without Getting Burned. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Ottawa, Ontario, Canada) *(PLDI '06)*. Association for Computing Machinery, New York, NY, USA, 239–251. https://doi.org/10.1145/1133981.1134010

[19] Hugh Leather and Chris Cummins. 2020. Machine Learning in Compilers: Past, Present and Future. In *2020 Forum for Specification and Design Languages (FDL)*. IEEE Computer Society, 1–8. https://doi.org/10.1109/FDL50818.2020.9232934

[20] Charith Mendis, Cambridge Yang, Yewen Pu, Saman Amarasinghe, and Michael Carbin. 2019. *Compiler Auto-Vectorization with Imitation Learning*. Curran Associates Inc., Red Hook, NY, USA.

[21] Microsoft. 2022. ONNX Runtime. https://www.onnxruntime.ai retrieved September 1, 2022.

[22] Antoine Monsifrot, François Bodin, and Rene Quiniou. 2002. A Machine Learning Approach to Automatic Production of Compiler Heuristics. In *Proceedings of the 10th International Conference on Artificial Intelligence: Methodology, Systems, and Applications (AIMSA '02)*. Springer-Verlag, London, UK, UK, 41–50. http://dl.acm.org/citation.cfm?id=646053.677574

[23] Raphael Mosaner, David Leopoldseder, Wolfgang Kisling, Lukas Stadler, and Hanspeter Mössenböck. 2022. Compilation Forking: A Fast and Flexible Way of Generating Data for Compiler-Internal Machine Learning Tasks. *The Art, Science, and Engineering of Programming* 7 (06 2022). https://doi.org/10.22152/programming-journal.org/2023/7/3

[24] Raphael Mosaner, David Leopoldseder, Lukas Stadler, and Hanspeter Mössenböck. 2021. Using Machine Learning to Predict the Code Size Impact of Duplication Heuristics in a Dynamic Compiler. In *Proceedings of the 18th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR '21)*. Association for Computing Machinery, 127–135. https://doi.org/10.1145/3475738.3480943

[25] Dorit Nuzman, Ira Rosen, and Ayal Zaks. 2006. Auto-Vectorization of Interleaved Data for SIMD. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Ottawa, Ontario, Canada) *(PLDI '06)*. Association for Computing Machinery, New York, NY, USA, 132–143. https://doi.org/10.1145/1133981.1133997

[26] Michail Papadimitriou, Eleni Markou, Juan Fumero, Athanasios Stratikopoulos, Florin Blanaru, and Christos Kotselidis. 2021. Multiple-Tasks on Multiple-Devices (MTMD): Exploiting Concurrency in Heterogeneous Managed Runtimes. In *Proceedings of the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Virtual, USA) *(VEE 2021)*. Association for Computing Machinery, New York, NY, USA, 125–138. https://doi.org/10.1145/3453933.3454019

[27] Eunjung Park, John Cavazos, and Marco A. Alvarez. 2012. Using Graph-Based Program Characterization for Predictive Modeling. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization* (San Jose, California) *(CGO '12)*. Association for Computing Machinery, New York, NY, USA, 196–206. https://doi.org/10.1145/2259016.2259042

[28] Rodrigo C. O. Rocha, Vasileios Porpodas, Pavlos Petoumenos, Luís F. W. Góes, Zheng Wang, Murray Cole, and Hugh Leather. 2020. Vectorization-Aware Loop Unrolling with Seed Forwarding. In *Proceedings of the 29th International Conference on Compiler Construction* (San Diego, CA, USA) *(CC 2020)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3377555.3377890

[29] Ricardo Nabinger Sanchez, Jose Nelson Amaral, Duane Szafron, Marius Pirvu, and Mark Stoodley. 2011. Using Machines to Learn Method-Specific Compilation Strategies. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '11)*. IEEE Computer Society, USA, 257–266. https://doi.org/10.1109/CGO.2011.5764693

[30] Mark Stephenson and Saman Amarasinghe. 2005. Predicting unroll factors using supervised classification. In *International Symposium on Code Generation and Optimization*. IEEE Computer Society, 123–134. https://doi.org/10.1109/CGO.2005.29

[31] Kevin Stock, Louis-Noël Pouchet, and P. Sadayappan. 2012. Using Machine Learning to Improve Automatic Vectorization. *ACM Trans. Archit. Code Optim.* 8, 4, Article 50 (jan 2012), 23 pages. https://doi.org/10.1145/2086696.2086729

[32] Zheng Wang and Michael O'Boyle. 2018. Machine Learning in Compiler Optimization. *Proc. IEEE* 106, 11 (Nov 2018), 1879–1901. https://doi.org/10.1109/JPROC.2018.2817118

[33] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to Rule Them All. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (Indianapolis, Indiana, USA) *(Onward! 2013)*. Association for Computing Machinery, New York, NY, USA, 187–204. https://doi.org/10.1145/2509578.2509581

[34] Mingzhou Zhou, Xipeng Shen, Yaoqing Gao, and Graham Yiu. 2014. Space-Efficient Multi-Versioning for Input-Adaptive Feedback-Driven Program Optimizations. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (Portland, Oregon, USA) *(OOPSLA '14)*. Association for Computing Machinery, New York, NY, USA, 763–776. https://doi.org/10.1145/2660193.2660229

Part III

Related Work and Conclusions

Chapter 9

Related Work

The use of machine learning in compilers is a well-researched concept; first approaches [22] date back to the end of the last millennium. Wang and O'Boyle [157], Ashouri et al. [10] and Leather and Cummins [96] provide extensive surveys of the developments in the field over the last 20 years. The research mostly focused on static compilers whereas the interplay of machine learning and dynamic compilers is often neglected. We identified four major dimensions to differentiate research in this broad field and highlight (in bold-face) how our work is positioned along these axes:

- System: static or **dynamic** compilation
- Granularity: global compiler flag, method compilation strategy or **method-local optimization decision**
- Kind of optimization: flag tuning, phase ordering, phase selection, inlining, **loop unrolling, loop peeling, vectorization**, ...
- Technique: genetic algorithms, **decision trees**, support vector machines, **deep learning**, reinforcement learning, ...

In this chapter, we compare our work to influential work from the past and the present, and identify similarities and differences along these dimensions. We also discuss concepts such as iterative compilation or multi-versioning, which are inextricably linked to the use of machine learning in compilers.

9.1 Previous PhD Theses

Over the past decades, the research on machine learning for compilers has culminated in several PhD theses.

Thesis	Year	Major Topics
Fursin [50]	2004	Iterative compilation and search space reduction Performance prediction
Cavazos [23]	2005	Optimization and algorithm selection Supervised learning of heuristics in the JikesRVM
Dubach [44]	2009	Architecture-specific compilation strategies
Leather [95]	2010	Feature engineering Iterative compilation
Kulkarni [90]	2014	Phase ordering and optimization selection Inlining heuristics
Mendis [106]	2020	Auto-vectorization Throughput prediction
Cummins [33]	2020	Data generation Learning on raw code

Table 9.1: Overview of related PhD theses

The research described in all the above theses inspired our own work, and we will discuss corresponding publications in the sections below. We want to especially highlight the theses of Cavazos [23] and Kulkarni [90], in which they also deal with dynamic compilers, but in contrast to our work, do not address their peculiarities regarding compilation noise and consistency nor make use of concepts such as deoptimization.

9.2 Iterative Compilation and Multi-versioning

Iterative compilation [17; 51; 80] is a fundamental concept for finding compilation parameters, which optimize the program to be compiled. In its simplest form, this is done by exhaustively enumerating the state space of all compiler parameter configurations,

compiling the target program with each configuration, and selecting the best-performing version for further use. In combination with search techniques or genetic algorithms [4; 10], iterative compilation is also the core of autotuning frameworks, such as OpenTuner [8] and SRTuner [125], which are discussed in Section 9.3. With the rise of using machine learning in compilers, iterative compilation has been used to create labels for enabling supervised learning [96].

Compilation forking has similarities to iterative compilation. However, traditional approaches use iterative compilation to optimize a method as a whole. We optimize different decisions within a method separately but with an identical compilation history, which is paramount in dynamic compilers. In addition, we deploy and execute the different versions alternatingly in single program runs. This positions our approach more closely to multi-versioning [49; 93; 94; 172]. Multi-versioning compiles and deploys multiple versions of a program region—for example, methods or loops—and selects, at run time, what version to use. Its accustomed goal is to have optimized code versions executed for a particular input category [49; 172]. For example, Lau et al. [93] create multiple versions of a method in the V9 JVM [57] and modify the method dispatch to execute the versions alternatingly. After enough measurements have been collected, they install the fastest version and discard the others.

Fursin et al. [51] proposed the use multi-versioning to speed up iterative compilation in the static EKOPath compiler. Instead of evaluating one compiler parameter configuration per program execution, they identified the most important functions of a program, compiled them with different configurations, and measured all versions in a single execution. Compilation forking adopts this idea but leverages it into dynamic compilation by ensuring an identical compilation history. In addition, its versions do not stem from changing global flags prior to compilation as in [51], but from method-local changes to single optimization decisions. This makes our approach more fine-grained.

9.3 Autotuning

Autotuning is the process of automatically optimizing a program by exploring different compilation scenarios [10]. Typically, autotuning denotes a holistic optimization process, where the optimization space is too large to be fully explored by exhaustive iterative compilation. Frameworks, such as OpenTuner [8] and SRTuner [125], find (near-)optimal

compiler flag configurations for single programs but not necessarily implement means for generalizing the configurations to new data.

OpenTuner OpenTuner [8] takes a definition of the optimization space (parameters and value ranges) and a success function for evaluating an optimized program. Then it selects promising optimization parameter configurations, evaluates their quality via iterative compilation and uses the results to guide the search for (near-)optimal configurations. OpenTuner uses multiple search algorithms at the same time which share their results with each other. These algorithms include evolutionary mutation, hill climbing, pattern search or random search. Trimming down large search spaces efficiently is an important task. In our work, we addressed only single optimization heuristics per model, keeping the search space small and, thus, not requiring complex autotuning techniques to acquire the training labels.

SRTuner SRTuner [125] is another purely search-based autotuning framework, which also considers inter-optimization relationships. It identifies more impactful optimizations and tunes their parameters first. To not get stuck in local optima, they trade-off exploration and exploitation with a custom algorithm, which uses feedback from previous trials. It either evaluates flag combinations in close proximity to the past (exploitation) or, if rewards have become smaller, it falls back to exploring other regions of the search space.

Interactive Applications Mpeis et al. [119] devised an interesting approach for replay-based offline optimization of Android programs. At run time, they capture hot regions, inputs and memory snapshots when entering the regions as well as the architectural state of the processor. When the phone is idle, they perform iterative compilation with genetic algorithms to replay the hot code with different optimization configurations, i.e., LLVM passes. Upon the next start of the Android application, the best optimization settings for each region are selected. While the costly optimization process is performed "offline", i.e., when the user is not running the application, they have to exclude code with I/O or other side-effecting behavior, which cannot be captured and replayed.

ML for Autotuning Park et al. [125] acknowledge that autotuning is expensive and see a development towards generalization techniques (i.e., machine learning) and the ability of autotuners to provide them with training data. COBAYN [11] follows such an approach by

applying iterative compilation excessively offline to obtain the best compiler flag settings for a program and derive a Bayesian network, which stores the obtained knowledge. After deployment, this network can be used to create one or multiple flag configurations. This approach converges faster towards a good compilation of the program than with iterative compilation. Park et al. [124] used support vector machines and linear regression models in a similar way to guide iterative compilation towards few promising sets of optimizations to enable. They compared three types of predictors: predicting optimization sequences to apply, predicting the speedup of a particular configuration compared to the baseline, and predicting whether one configuration works better on a specific program than another configuration. In our work, as we addressed single optimizations, only the speedup compared to the baseline was a suitable metric for model training.

Granularity Autotuning frameworks generally operate on (global) flags for static compilers, e.g. GCC flags [11; 89; 125]; tuning optimizations for performing method-local decisions differently is generally not supported. We proposed compilation forking to support this important use case, especially during dynamic compilation. If a method contains multiple optimization targets (e.g. loops), each of them is analyzed independently by creating a new fork. In that case, no holistically optimized version of a method is created, which is a fundamental and conceptional difference to autotuning.

9.4 Machine Learning in Static Compilers

This thesis contributes to facilitating the use of machine learning particularly in dynamic compilers. However, the majority of research on data-driven compiler optimizations addresses static compilers, often for LLVM. Despite being different systems, we want to highlight related work in static compilers, as there are still concepts which carry over to our work.

MilepostGCC The Milepost project [52; 53] was one of first larger projects for applying machine learning in a compiler, and we consider it among the most influential work in the field. The MilepostGCC framework builds on top of GCC and provides means for interacting with GCC's optimization phases and parameters. Fursin et al. [53] used this framework to create training data by investigating the impact of 500 different compiler flag combinations for each benchmark program. They created a probabilistic model which

maps the program features of the training data to the distribution of good solutions, which are all optimization sequences that achieve at least 98% of the optimal performance. For selecting an optimization sequence for a new feature set, they used 1-nearest neighbor to fetch the distribution for the most similar feature vector in the training data. Milepost-GCC [53] was able to outperform the default GCC by around 11% on the MiBench [62] benchmark suite. Fursin et al. [53] presented the first holistic framework for supporting data generation, model training and model deployment. It had a great impact on subsequent research as well as on this thesis, although MilepostGCC focuses on static compilation and compiler flag optimization.

DeepTune Cummins et al. [34] presented *DeepTune*, which is a machine learning pipeline capable of creating models based on source code without the need for manually defining features. They map source code to tokens and use embeddings to create more dense representations of the code, which is then collapsed into one discriminating "feature" vector obtained via an LSTM [69] network. A dense neural network is then responsible for learning the relationship between their created feature vectors and the heuristic value. They evaluated DeepTune on two OpenCL-related classification problems: finding the optimal device (CPU or GPU) on which to execute a kernel and finding an optimal thread-coarsening factor $\in 1, 2, 4, 8, 16, 32$. Both case studies outperformed the hand-crafted heuristics in most cases. In this thesis, we dealt with a dynamic compiler and its internal IR representation of code, which are two fundamental differences to the work of [34]. Creating feature vectors implicitly is a very interesting approach, although it increases the black-box character of the whole system.

Vemal Mendis et al. [108; 109] used machine learning for improving Superword-Level Parallelism (SLP) vectorization and for predicting the execution cycles of basic blocks on LLVM. Their initial, non-machine-learning-related work, goSLP [107], solved the statement packing problem and the vector permutation selection problem for SLP auto-vectorization. In their follow-up work [109], they used gated graph neural networks (GGNN) and imitation learning to mimic the vectorization decisions of goSLP, which are optimal but compile-time-intensive. Their learned policy, called *Vemal* [106; 109], outperformed the existing LLVM SLP heuristics on most benchmarks, with a geometric mean run-time speedup of 1.5%. However, goSLP still outperforms the imitating model for six out of seven benchmarks. Their work shows that it is possible to derive a well-performing machine learning model from a (near)-optimal but expensive solution.

Ithemal In order to support performance optimizations without having a well-performing solution to imitate, Mendis et al. [108] created *Ithemal*, which predicts the execution cycles of basic blocks with machine learning. They decomposed assembly instructions into tokens and hierarchically created an embedding for each instruction based on the token's embeddings. To account for the variable length of instructions when creating the embeddings and the variable number of instructions when predicting the execution cycle for a basic blocks, they used recurrent neural networks (RNN) with long short term memory (LSTM) [69]. They fetched more than four million basic blocks from benchmark and real world applications and timed each basic block by moving it to a loop for achieving steady performance. Ithemal outperforms state-of-the-art, hand-written analytical models by far in terms of prediction accuracy while being executed equally fast. Their work shows that in an ideal world, where problems are isolated from noise and the environment, hand-crafted models stand no chance against data-driven approaches.

MLGO Recently, Trofin et al. [154] introduced MLGO, which is a machine learning framework available in the LLVM repository. For their initial case study, they learned a model to perform inlining-for-size. Similar to our early work [116], they found working with code size metrics less prone to measurements noise. Conceptually, their work shows substantial differences to ours: The immediate effect of inlining a particular callee is never measured; instead, they measure only the cumulative effect (code size impact) after LLVM's inlining phase has processed all callees of a method. This design decision requires more training data and produces less accurate models [154]. In contrast, compilation forking enables us to isolate the effect of method-local decisions. Trofin et al. [154] trained their model with reinforcement learning [77], which does not require labelled training data. Only eleven features, such as `callee_basic_block_count` or `callsite_height`, were used for training. To not start from a blank model, they initially trained a default model which imitates the decisions of the LLVM inlining heuristic. Then, their approach uses the cumulative reward, which is the code size increase or decrease measured from the compiled program after model usage, as feedback for refining the model. Interestingly, they have an oppositional take on machine learning in compilers, where they define understandability of models for compiler engineers secondary to model performance. To the best of our knowledge, this is the only related work, where a model is actually deployed in in a state-of-the-art industry compiler.

MLGOPerf Ashouri et al. [9] picked up the MLGO approach and extended it to incorporate performance metrics. They created a two-step approach: At first, they trained a

supervised model to predict the speedup of a method compared to O3, based on features extracted *after* inlining. The training data was obtained via autotuning, their approach being based on OpenTuner [8]. During reinforcement learning, the supervised model is used to predict the cumulative reward for the method after inlining. This two-step approach bypasses the problem of identifying the speedup of individual inlining decisions. We used a similar approach in the past [116], where we predicted the code size impact of individual code duplications by invoking a model, which was trained to predict the total code size after the duplication phase, twice. While Ashouri et al. [9] identified multiple challenges to be solved in future work, the original work of Trofin et al. [154] can be used in LLVM.

9.5 Machine Learning in Dynamic Compilers

The majority of research in the domain of machine learning in compilers has been conducted for static compilation. However, there have also been a number of approaches which explore the use of ML in dynamic or just-in-time (JIT) compilers. One of the more frequently targeted compilers is the one used in the Jikes RVM [5], which is a Java virtual machine designed during and for research purposes.

The inlining heuristics in the Jikes RVM depended on multiple manually-tuned static thresholds. Cavazos and O'Boyle [25] used genetic algorithms to search the value space for this tuple of thresholds. In the end, they replaced the initial inlining thresholds with the values found to be optimal for the training set, which outperformed the previous heuristic on the test set as well. While no machine learning model is used during compilation, this approach shows that data-driven approaches can outperform hand-tuned flags.

Simon et al. [143] developed a more machine-learning-centered approach for replacing the inlining heuristics in Maxine [159] and in the Java HotSpot VM [71]. Their neural networks were created by the NEAT [148] framework. NEAT uses a genetic process to create a neural network from unlabelled data, by evaluating randomly generated neural networks on benchmarks and using performance measurement for genetic refinements—in our thesis, we showed how to efficiently extract labelled optimization data. To produce a human-understandable heuristic, Simon et al. [143] then used the resulting neural network for producing a data set with artificially created labels. This data set is then used for training a decision tree. Both, the neural network and the decision tree outperform the heuristic created in [25], which was implemented as a reference.

Kulkarni and Cavazos [91] addressed the phase-ordering problem in the Jikes RVM with neural networks. Instead of predicting the order of a fixed set of N optimizations, they invoke an ANN after each optimization to select the next most promising optimization to apply. This process is continued until the network chooses to finish optimizing the program further. The network itself is created by the NEAT [148] framework, which circumvents the task of creating labelled data sets, similar to [143]. Kulkarni and Cavazos [91] report speedups of more than 5% for several benchmarks in terms of peak performance. The speedups for the total time (including compile time) are significantly smaller, which we assume is due to the frequent invocation of the neural network between optimizations.

Cavazos and O'Boyle [26] found out that the overall run time of SPECjvm98 benchmarks is significantly reduced by using compilation strategies specific to each method, rather than one-size-fits-all global compilation settings. They investigated 20 different optimizations (e.g., loop unrolling or constant propagation) in the Jikes RVM, which can be either enabled or disabled. To not exhaustively search the large state space, they generated training data by executing random sets of optimization combinations and used the best performing configurations as training labels. The features consist of static information about the method, such as the portion of branch operations in the method body or whether there is exception handling code. Using logistic regression, they then devised a model to predict for new feature sets which optimizations to enable or disable. They also pointed out the importance of keeping the compile time low for dynamically compiled programs.

Sanchez et al. [140] used support vector machines to learn which optimization parameters to choose when compiling a method in V9's [57] Testarossa compiler. While optimization type, granularity and ML technique differ from our work, their approach for data generation shows similarities. They compile and execute multiple versions of a method, with different per-method optimization parameters, in single program executions. However, their approach works iteratively. After they have gathered enough measurements for a version, they perform deoptimization and compile another version with different parameters. In contrast to our work, multiple versions are never deployed at the same time, which could be a problem regarding usage noise. Sanchez et al. [140] also use the processor time stamp counter for their measurements but only collect the total time of a method instead of its self time.

9.6 Data Generation

Our research on compilation forking not only addresses *how* to obtain high-quality training data from dynamically compiled programs but also facilitates creating large amounts of training data by being universally applicable to arbitrary user programs. The importance of versatile training data in large quantities is also outlined in related work [35; 56; 155], where approaches have emerged to synthesize benchmark programs to obtain larger data sets for model training.

CLGen Cummins et al. [35] proposed *CLgen*, a tool for synthesizing OpenCL programs with pre-defined signatures, based on an LSTM [69] language model. The model was trained on a large corpus of OpenCL programs, which was collected from GitHub and sanitized in terms of non-functional variance, such as comments or variable names. In a controlled experiment, human experts were incapable of detecting whether the programs were synthesized or created by humans. They also compared CLgen to *CLsmith*[101], which was developed for differential testing and produces random OpenCL kernels with deterministic outputs. Both, their qualitative experiment and a quantitative measure of similarity to existing benchmark programs, indicate that CLgen produces programs that looked more as if they were created by humans than CLsmith [35]. However, Tsimpourlas et al. [155] criticized that only 2.33% of programs, synthesized by CLgen, actually compiled.

In a subsequent study, Goens et al. [56] found out that using only data from GitHub when training a model works significantly better compared to only using the programs synthesized by CLGen. They identified the lack in feature diversity and the repetitive nature of the synthesized programs as major causes for performing so poor on their own. However, Cummins et al. [35] proposed to use synthesized benchmarks only to enhance genuine data. This indicates that our approach, which can generate data from real user programs, produces data that is better suited for predictive tasks.

BenchPress In a very recent study, Tsimpourlas et al. [155] addressed the shortcomings of CLgen and proposed *BenchPress* as an alternative. BenchPress is based on the deep learning natural language processing model BERT [43]. However, it is extended by the concept of *holes*, which allows the model to learn and to predict lines of code at arbitrary positions. With this approach, they can control the feature sets in the generated code, which

is a big advantage over tools such as CLgen or CLsmith. 86% of synthesized programs did compile (in contrast to 2.33% for CLgen). However, the synthesized programs are typically very small (3-16 LLVM IR instructions). Applying compilation forking to a program cannot break anything, therefore having a compilation rate of 100%. However, the capability of creating benchmarks with pre-defined features is very interesting, as it enables model training with evenly distributed features from executable programs, rather than achieving that by data augmentation during pre-processing.

9.7 Entry Barrier

The ability to benefit from machine learning in a compiler is aggravated by a steep learning curve for compiler engineers when entering the field of machine learning. Frameworks such as OpenTuner [8], ComPy-Learn [18] or CompilerGym [36] lower the entry barrier by providing pre-built pipelines, feature sets and algorithms for experimenting with common compiler optimizations.

ComPy-Learn Brauckmann et al. [18] developed the ComPy-Learn toolbox to provide compiler engineers with APIs to setup up models with different program representations and architectures for LLVM. They support program representations such as syntax token sequences, ASTs or LLVM IR graphs as well as model types, such as recurrent neural networks (RNN) or (gated) graph neural networks ((G)GNN). However, creating labelled training data is not part of their pipeline.

CompilerGym Cummins et al. [36] proposed CompilerGym to bring the expertise of compiler engineers and ML researchers closer together. They formulate compiler optimization tasks as machine learning problems and expose these tasks as environments via a Python frontend based on the OpenAI Gym [21] interfaces, whereas the backend handles the interaction with the actual problem domain, e.g., GCC or LLVM. In their initial implementation, they provided environments for experimenting with LLVM phase ordering, GCC flag selection and CUDA loop nest generation. For each of these environments the user can select different feature representations, optimization goals, data sets and configurations for the optimization itself.

In our work, we also created a small framework [112; 113] for replacing compiler tasks with learned models and for configuring the training process, which is described in Chapter 6 and Chapter 7. We would be curious about extending our framework with functionality of above works, as they do not support dynamic compilers at the moment.

9.8 Self-optimizing Models

Updating models after deployment or tailoring them to specific environments, as we did in [112], is rarely addressed in related work.

Tartara and Crespi Reghizzi [151] proposed *continuous learning of compiler heuristics*, which is a holistic approach for finding a set of optimization heuristics in a static compiler. They defined a grammar from which new heuristics can be inferred based on a pre-defined set of program features and mathematical or logical operations on them. For the composition of heuristics and a particular compilation plan they used genetic algorithms [25; 30; 151]. Their approach outperformed GCC O3 on the selected benchmarks. However, this approach needs a controlled environment and multiple program runs to compare the performance and to update heuristics in the static compiler.

MLGO [154], which we discussed in Section 9.4 is based on reinforcement learning and therefore has the capabilities of refining the model at run time. However, Trofin et al. [154] state that they focus on deploying models with good generalization and refrain from frequent refinements after deployment.

Chapter 10

Future Work

The goal of this thesis was to improve particular optimization heuristics in a dynamic compiler by either directly employing machine learning or assisting compiler engineers during the heuristics designs process, as discussed in Chapter 4 and Chapter 5. Traditional approaches, based on iterative compilation, do not translate well to dynamic compilers in terms of consistency and comparable measurements. We contributed to the solution of these challenges by proposing compilation forking in Chapter 6 which we successfully used in Chapter 7 and Chapter 8. However, our vision of using machine learning in dynamic compilers allows for future work, which we briefly outline in this chapter.

Nested Forking In this thesis, compilation forking has only been used for consistently measuring the impact of single optimization decisions. However, we prototyped an extension to our approach, which enables *nested forking* at multiple decision points within the compilation process. For example, compilation forking could be applied to a method with a vectorizable loop to create versions with different vector lengths during loop vectorization. Later, the resulting versions could be forked again with different unroll factors. This could be used for analyzing the interplay between vectorization and unrolling parameters. In the context of machine learning, it would allow us to create more holistic models, predicting a vector of optimization parameters for related optimizations. Nested forking would impose challenges regarding scalability, as the optimization space would grow exponentially.

Large Optimization Spaces Iterative compilation for optimizing statically compiled programs can explore very large optimization spaces, given enough time. Our approach for dynamic compilation focuses on local optimizations, but ensures comparable measure-

ments by multi-versioning. The resulting code size growth limits the number of versions that can be explored in single executions. To explore larger optimization spaces, we would require a somehow consistent compilation history between different executions to identify the best performing parameters across multiple executions. As an alternative, future work could also formulate the machine learning task in a different way. For example, Lau et al. [93] proposed learning which of two parameter configurations would result in higher performance for a given function. The training data for such models would not require to identify global maxima in the parameter space. This would allow to train machine learning models on data from single executions, even for large optimization spaces.

Capture-and-replay Mpeis et al. [119] used a capture-and-replay approach for exploring the impact of optimization decisions on a user-specific environment offline. While their approach has limitations in terms of I/O and side-effecting code regions, it would be interesting if a capture mechanism could enable iterative replay of dynamically compiled methods in a consistent way. In addition to their work, we think about capturing the profiling information gathered during interpretation as well as the order in which methods were compiled. Injecting this information during the replay process could result in comparable program versions and enable traditional iterative compilation. However, exploring n versions of a method would require n replays of the program, which would increase the data generation time linearly with the number of versions. We still think that it would be an interesting trade-off to gain scalability and still keeping the requirements regarding measurement consistency of different versions.

Reinforcement Learning In this work, we focused on random forests to produce human-readable models and deep neural networks, which are considered state-of-the-art. However, more recently, reinforcement learning (RL) [77] has received increasing attention, also in the domain of compiler optimizations [63; 104; 156]. It would be interesting to investigate RL in the context of our self-optimizing models, which we proposed in Chapter 7. In combination with compilation forking, RL could, at run time, improve the model after each compilation. Instead of creating multiple versions at once, the state of the intermediate compilation before the first forking point could be persisted to ensure a common past for forked compilations. Based on the decisions of the RL model, a specific version could be compiled from the persisted state. This version could then be evaluated and could again provide feedback to the model. Coming up of with a feasible architecture for such an approach and investigating its integration with existing frameworks, such as SuperSonic [156], could be a very interesting area of future work.

Chapter 11

Conclusions

Machine learning has found its way into compiler research many years ago. However, to the best of our knowledge, only a single static industry compiler [154] uses machine learning in production. This thesis makes multiple contributions to the field of machine learning in dynamic compilers:

Machine Learning to Assist Compiler Engineers We showed how machine learning can be used during the design process of compiler heuristics. The models either point compiler engineers to flaws in existing hand-crafted heuristics or provide them with insights about important program features and thresholds. This way, the improved compiler retains maintainability and understandability, because no machine learning black boxes are deployed in the production system.

Compilation Forking Compilation forking is an approach for consistently measuring the impact of method-local optimization decisions in a dynamic compilation environment. It ensures that multiple versions of a method share the same compilation history and are executed in a single program execution to expose them to the same environment state and the same user behavior. Our approach not only enables generating data for training machine learning models in a dynamic compiler, but also facilitates large-scale data generation in single executions instead of multiple runs.

Self-optimizing Heuristics We devised an approach where compilation forking together with deoptimization can be used to automatically update deployed models at run time and at the user site. This allows us to create self-optimizing compiler heuristics in contrast to

general models which are trained only prior to deployment. Our evaluation showed that highly specialized models can significantly outperform existing hand-crafted heuristics in a highly tuned industry compiler.

Case Studies on Learned Optimizations Throughout this thesis we conducted multiple case studies to evaluate our approaches. In the course of this, we created machine learning models for the optimizations: code duplication, loop peeling, partial loop unrolling and unrolling of vectorized loops. These models were centered around the major success metrics—code size and execution time—for compiled programs. Apart from state-of-the-art deep learning models, we also showed that simpler decision-tree-based models can provide good results with the benefit of being human-readable.

The research for this thesis has been conducted in cooperation with Oracle Labs, which enabled us to implement and benchmark our approaches in one of the highest-optimizing dynamic Java compilers on the market, the GraalVM compiler.

List of Figures

List of Tables

Bibliography

[1] AARCH64 2023. AARCH64 architecture manual. https://developer.arm.com/documentation/ddi0600/latest/

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

[3] Laksono Adhianto, S Banerjee, M. Fagan, M Krentel, Gabriel Marin, John Mellor-Crummey, and Nathan Tallent. 2010. HPCTOOLKIT: Tools for Performance Analysis of Optimized Parallel Programs. *Concurrency and Computation: Practice and Experience* 22, 6 (apr 2010), 685–701. https://doi.org/10.1002/cpe.1553

[4] L. Almagor, Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven W. Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. 2004. Finding Effective Compilation Sequences. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems* (Washington, DC, USA) *(LCTES '04)*. Association for Computing Machinery, New York, NY, USA, 231–239. https://doi.org/10.1145/997163.997196

[5] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. 2000. The Jalapeño virtual machine. *IBM Systems Journal* 39, 1 (2000), 211–238. https://doi.org/10.1147/sj.391.0211

[6] Laith Alzubaidi, Jinglan Zhang, Amjad J. Humaidi, Ayad Q. Al-Dujaili, Ye Duan, Omran Al-Shamma, Jesus Santamaría, Mohammed Abdulraheem Fadhel, Muthana Al-Amidie, and Laith Farhan. 2021. Review of deep learning: concepts, CNN architectures, challenges, applications, future directions. *Journal of Big Data* 8 (2021).

[7] AMD64 2023. AMD64 architecture manual. `https://www.amd.com/system/files/TechDocs/24594.pdf`

[8] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. OpenTuner: An extensible framework for program autotuning. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*. IEEE Computer Society, 303–315. `https://doi.org/10.1145/2628071.2628092`

[9] Amir H. Ashouri, Mostafa Elhoushi, Yuzhe Hua, Xiang Wang, Muhammad Asif Manzoor, Bryan Chan, and Yaoqing Gao. 2022. Work-in-Progress: MLGOPerf: An ML Guided Inliner to Optimize Performance. In *2022 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*. 3–4. `https://doi.org/10.1109/CASES55004.2022.00008` long version: `https://arxiv.org/pdf/2207.08389.pdf`.

[10] Amir H. Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. 2018. A Survey on Compiler Autotuning Using Machine Learning. *ACM Comput. Surv.* 51, 5, Article 96 (Sept. 2018), 42 pages. `https://doi.org/10.1145/3197978`

[11] Amir Hossein Ashouri, Giovanni Mariani, Gianluca Palermo, Eunjung Park, John Cavazos, and Cristina Silvano. 2016. COBAYN: Compiler Autotuning Framework Using Bayesian Networks. *ACM Trans. Archit. Code Optim.* 13, 2, Article 21 (jun 2016), 25 pages. `https://doi.org/10.1145/2928270`

[12] Joel Auslander, Matthai Philipose, Craig Chambers, Susan J. Eggers, and Brian N. Bershad. 1996. Fast, Effective Dynamic Compilation. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation* (Philadelphia, Pennsylvania, USA) *(PLDI '96)*. Association for Computing Machinery, New York, NY, USA, 149–159. `https://doi.org/10.1145/231379.231409`

[13] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. 1994. Compiler Transformations for High-Performance Computing. *ACM Comput. Surv.* 26, 4 (Dec. 1994), 345–420. `https://doi.org/10.1145/197405.197406`

[14] Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. 2017. Virtual Machine Warmup Blows Hot and Cold. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 52 (oct 2017), 27 pages. https://doi.org/10.1145/3133876

[15] Christopher M. Bishop. 2006. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg.

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

[17] François Bodin, Toru Kisuki, Peter Knijnenburg, Mike OBoyle, and Erven Rohou. 1998. Iterative compilation in a non-linear optimisation space. *Workshop on Profile and Feedback-Directed Compilation* (03 1998). https://hal.inria.fr/inria-00475919/document

[18] Alexander Brauckmann, Andrés Goens, and Jeronimo Castrillon. 2020. ComPy-Learn: A toolbox for exploring machine learning representations for compilers. In *2020 Forum for Specification and Design Languages (FDL)*. 1–4. https://doi.org/10.1109/FDL50818.2020.9232946

[19] Alexander Brauckmann, Andrés Goens, Sebastian Ertel, and Jeronimo Castrillon. 2020. Compiler-Based Graph Representations for Deep Learning Models of Code. In *Proceedings of the 29th International Conference on Compiler Construction* (San Diego, CA, USA) *(CC 2020)*. Association for Computing Machinery, New York, NY, USA, 201–211. https://doi.org/10.1145/3377555.3377894

[20] L. Breiman, J. Friedman, C.J. Stone, and R.A. Olshen. 1984. *Classification and Regression Trees*. Taylor & Francis. https://books.google.at/books?id=JwQx-WOmSyQC

[21] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. 2016. OpenAI Gym. (2016). https://doi.org/10.48550/ARXIV.1606.01540

[22] Brad Calder, Dirk Grunwald, Michael Jones, Donald Lindsay, James Martin, Michael Mozer, and Benjamin Zorn. 1997. Evidence-Based Static Branch Prediction Using

Machine Learning. *ACM Trans. Program. Lang. Syst.* 19, 1 (jan 1997), 188–222. https://doi.org/10.1145/239912.239923

[23] John Cavazos. 2005. *Automatically Constructing Compiler Optimization Heuristics using Supervised Learning*. Ph. D. Dissertation. Amherst, MA, USA. https://www.eecis.udel.edu/~cavazos/thesis.pdf

[24] John Cavazos, Grigori Fursin, Felix Agakov, Edwin Bonilla, Michael F. P. O'Boyle, and Olivier Temam. 2007. Rapidly Selecting Good Compiler Optimizations Using Performance Counters. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '07)*. IEEE Computer Society, USA, 185–197. https://doi.org/10.1109/CGO.2007.32

[25] John Cavazos and Michael F. P. O'Boyle. 2005. Automatic Tuning of Inlining Heuristics. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing (SC '05)*. IEEE Computer Society, USA, 14. https://doi.org/10.1109/SC.2005.14

[26] John Cavazos and Michael F. P. O'Boyle. 2006. Method-Specific Dynamic Compilation Using Logistic Regression. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications* (Portland, Oregon, USA) *(OOPSLA '06)*. Association for Computing Machinery, New York, NY, USA, 229–240. https://doi.org/10.1145/1167473.1167492

[27] Stefano Cazzulani. 2012. Octane: The JavaScript benchmark suite for the modern web. https://blog.chromium.org/2012/08/octane-javascript-benchmark-suite-for.html

[28] François Chollet et al. 2015. Keras. https://keras.io.

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

[30] Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. 1999. Optimizing for Reduced Code Space Using Genetic Algorithms. *SIGPLAN Not.* 34, 7 (May 1999), 1–9. https://doi.org/10.1145/315253.314414

[31] Keith D. Cooper, Devika Subramanian, and Linda Torczon. 2002. Adaptive Optimizing Compilers for the 21st Century. *J. Supercomput.* 23, 1 (Aug. 2002), 7–22. https://doi.org/10.1023/A:1015729001611

[32] Corinna Cortes and Vladimir Vapnik. 1995. Support-Vector Networks. *Mach. Learn.* 20, 3 (sep 1995), 273–297. https://doi.org/10.1023/A:1022627411411

[33] Chris Cummins. 2020. *Deep Learning for Compilers*. Ph. D. Dissertation. Edinburgh, Scotland, UK. https://era.ed.ac.uk/handle/1842/36866

[34] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. 2017. End-to-End Deep Learning of Optimization Heuristics. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 219–232. https://doi.org/10.1109/PACT.2017.24

[35] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. 2017. Synthesizing Benchmarks for Predictive Modeling. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization* (Austin, USA) *(CGO '17)*. IEEE Press, 86–99.

[36] Chris Cummins, Bram Wasti, Jiadong Guo, Brandon Cui, Jason Ansel, Sahir Gomez, Somya Jain, Jia Liu, Olivier Teytaud, Benoit Steiner, Yuandong Tian, and Hugh Leather. 2022. CompilerGym: Robust, Performant Compiler Optimization Environments for AI Research. In *Proceedings of the 20th IEEE/ACM International Symposium on Code Generation and Optimization* (Virtual Event, Republic of Korea) *(CGO '22)*. IEEE Press, 92–105. https://doi.org/10.1109/CGO53902.2022.9741258

[37] Pádraig Cunningham, Matthieu Cord, and Sarah Jane Delany. 2008. *Supervised Learning*. Springer Berlin Heidelberg, Berlin, Heidelberg, 21–49. https://doi.org/10.1007/978-3-540-75171-7_2

[38] Benoit Daloze, Stefan Marr, Daniele Bonetta, and Hanspeter Mössenböck. 2016. Efficient and Thread-Safe Objects for Dynamically-Typed Languages. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Amsterdam, Netherlands) *(OOPSLA 2016)*. Association for Computing Machinery, New York, NY, USA, 642–659. https://doi.org/10.1145/2983990.2984001

[39] Benoit Daloze, Chris Seaton, Daniele Bonetta, and Hanspeter Mössenböck. 2015. Techniques and Applications for Guest-Language Safepoints. In *Proceedings of the 10th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems* (Prague, Czech Republic) *(ICOOOLPS '15)*. Association for Computing Machinery, New York, NY, USA, Article 8, 10 pages. https://doi.org/10.1145/2843915.2843921

[40] Benoit Daloze, Arie Tal, Stefan Marr, Hanspeter Mössenböck, and Erez Petrank. 2018. Parallelization of Dynamic Languages: Synchronizing Built-in Collections. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 108 (oct 2018), 30 pages. https://doi.org/10.1145/3276478

[41] Daniele Cono D'Elia and Camil Demetrescu. 2018. On-Stack Replacement, Distilled. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 166–180. https://doi.org/10.1145/3192366.3192396

[42] L. Peter Deutsch and Allan M. Schiffman. 1984. Efficient Implementation of the Smalltalk-80 System. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Salt Lake City, Utah, USA) *(POPL '84)*. Association for Computing Machinery, New York, NY, USA, 297–302. https://doi.org/10.1145/800017.800542

[43] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/N19-1423

[44] Christophe Dubach. 2009. *Using Machine-Learning to Efficiently Explore the Architecture/Compiler Co-Design Space*. Ph. D. Dissertation. Edinburgh, Scotland, UK. https://era.ed.ac.uk/handle/1842/3867

[45] Gilles Duboscq, Lukas Stadler, Thomas Würthinger, Doug Simon, Christian Wimmer, and Hanspeter Mössenböck. 2013. Graal IR: An Extensible Declarative Intermediate Representation. In *Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop*. 1–9. https://ssw.jku.at/General/Staff/GD/APPLC-2013-paper_12.pdf

[46] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. 2013. An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler. In *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages* (Indianapolis, Indiana, USA) *(VMIL '13)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/2542142.2542143

[47] Josef Eisl, Matthias Grimmer, Doug Simon, Thomas Würthinger, and Hanspeter Mössenböck. 2016. Trace-Based Register Allocation in a JIT Compiler. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools* (Lugano, Switzerland) *(PPPJ '16)*. Association for Computing Machinery, New York, NY, USA, Article 14, 11 pages. https://doi.org/10.1145/2972206.2972211

[48] Josef Eisl, Stefan Marr, Thomas Würthinger, and Hanspeter Mössenböck. 2017. Trace Register Allocation Policies: Compile-Time vs. Performance Trade-Offs. In *Proceedings of the 14th International Conference on Managed Languages and Runtimes* (Prague, Czech Republic) *(ManLang 2017)*. Association for Computing Machinery, New York, NY, USA, 92–104. https://doi.org/10.1145/3132190.3132209

[49] Peng fei Chuang, Howard Chen, Gerolf F. Hoflehner, Daniel M. Lavery, and Wei chung Hsu. 2007. Dynamic profile driven code version selection. In *the 11th Annual Workshop on the Interaction between Compilers and Computer Architecture*. https://www.researchgate.net/publication/228952289_Dynamic_Profile_Driven_Code_Version_Selection

[50] Grigory Fursin. 2004. *Iterative Compilation and Performance Prediction for Numerical Applications*. Ph. D. Dissertation. Edinburgh, Scotland, UK. https://era.ed.ac.uk/handle/1842/565

[51] Grigori Fursin, Albert Cohen, Michael O'Boyle, and Olivier Temam. 2005. A Practical Method for Quickly Evaluating Program Optimizations. In *High Performance Embedded Architectures and Compilers*, Nacho Conte, Tomband Navarro, Wen-mei W. Hwu, Mateo Valero, and Theo Ungerer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 29–46.

[52] Grigori Fursin, Yuriy Kashnikov, Abdul Wahid Memon, Zbigniew Chamski, Olivier Temam, Mircea Namolaru, Bilha Mendelson, Ayal Zaks, Eric Courtois, François Bodin, Phil Barnard, Elton Ashton, Edwin Bonilla, John Thomson, Christopher Williams, and Michael O'Boyle. 2011. Milepost GCC: Machine Learning Enabled Self-tuning Compiler. *International Journal of Parallel Programming* 39 (06 2011), 296–327. https://doi.org/10.1007/s10766-010-0161-2

[53] Grigori Fursin, Cupertino Miranda, Olivier Temam, Mircea Namolaru, Ayal Zaks, Bilha Mendelson, Edwin Bonilla, John Thomson, Hugh Leather, Chris Williams, Michael O'Boyle, Phil Barnard, Elton Ashton, Eric Courtois, and François Bodin. 2008. MILEPOST GCC: machine learning based research compiler. *Proceedings of the GCC Developers' Summit 2008* (06 2008).

[54] Yoshihiko Futamura. 1999. Partial Evaluation of Computation Process—AnApproach to a Compiler-Compiler. *Higher Order Symbol. Comput.* 12, 4 (dec 1999), 381–391. https://doi.org/10.1023/A:1010095604496

[55] GCC 2023. GNU Compiler Collection. https://gcc.gnu.org/

[56] Andrés Goens, Alexander Brauckmann, Sebastian Ertel, Chris Cummins, Hugh Leather, and Jeronimo Castrillon. 2019. A Case Study on Machine Learning for Synthesizing Benchmarks. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (Phoenix, AZ, USA) *(MAPL 2019)*. Association for Computing Machinery, New York, NY, USA, 38–46. https://doi.org/10.1145/3315508.3329976

[57] Nikola Grcevski, Allan Kielstra, Kevin Stoodley, Mark Stoodley, and Vijay Sundaresan. 2004. JavaTM Just-in-Time Compiler and Virtual Machine Improvements for Server and Middleware Applications. In *Proceedings of the 3rd Conference on Virtual Machine Research And Technology Symposium - Volume 3* (San Jose, California) *(VM'04)*. USENIX Association, USA, 12.

[58] Matthias Grimmer, Manuel Rigger, Roland Schatz, Lukas Stadler, and Hanspeter Mössenböck. 2014. TruffleC: Dynamic Execution of C on a Java Virtual Machine. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools* (Cracow, Poland) *(PPPJ '14)*. Association for Computing Machinery, New York, NY, USA, 17–26. https://doi.org/10.1145/2647508.2647528

[59] Matthias Grimmer, Roland Schatz, Chris Seaton, Thomas Würthinger, and Hanspeter Mössenböck. 2015. Memory-Safe Execution of C on a Java VM. In *Proceedings of the 10th ACM Workshop on Programming Languages and Analysis for Security* (Prague, Czech Republic) *(PLAS'15)*. Association for Computing Machinery, New York, NY, USA, 16–27. https://doi.org/10.1145/2786558.2786565

[60] Matthias Grimmer, Chris Seaton, Roland Schatz, Thomas Würthinger, and Hanspeter Mössenböck. 2015. High-Performance Cross-Language Interoperability in a Multi-Language Runtime. In *Proceedings of the 11th Symposium on Dynamic Languages* (Pittsburgh, PA, USA) *(DLS 2015)*. Association for Computing Machinery, New York, NY, USA, 78–90. https://doi.org/10.1145/2816707.2816714

[61] Matthias Grimmer, Chris Seaton, Thomas Würthinger, and Hanspeter Mössenböck. 2015. Dynamically Composing Languages in a Modular Way: Supporting C Extensions for Dynamic Languages. In *Proceedings of the 14th International Conference on*

Modularity (Fort Collins, CO, USA) *(MODULARITY 2015)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/2724525.2728790

[62] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*. 3–14. https://doi.org/10.1109/WWC.2001.990739

[63] Ameer Haj-Ali, Nesreen K. Ahmed, Ted Willke, Yakun Sophia Shao, Krste Asanovic, and Ion Stoica. 2020. NeuroVectorizer: End-to-End Vectorization with Deep Reinforcement Learning. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization* (San Diego, CA, USA) *(CGO 2020)*. Association for Computing Machinery, New York, NY, USA, 242–255. https://doi.org/10.1145/3368826.3377928

[64] Christian Häubl and Hanspeter Mössenböck. 2011. Trace-Based Compilation for the Java HotSpot Virtual Machine. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java* (Kongens Lyngby, Denmark) *(PPPJ '11)*. Association for Computing Machinery, New York, NY, USA, 129–138. https://doi.org/10.1145/2093157.2093176

[65] Christian Häubl, Christian Wimmer, and Hanspeter Mössenböck. 2012. Evaluation of Trace Inlining Heuristics for Java. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing* (Trento, Italy) *(SAC '12)*. Association for Computing Machinery, New York, NY, USA, 1871–1876. https://doi.org/10.1145/2245276.2232084

[66] Christian Häubl, Christian Wimmer, and Hanspeter Mössenböck. 2014. Trace Transitioning and Exception Handling in a Trace-Based JIT Compiler for Java. *ACM Trans. Archit. Code Optim.* 11, 1, Article 6 (feb 2014), 26 pages. https://doi.org/10.1145/2579673

[67] Kaiming He, Xiangyu Zhang, and Shaoqing Ren andJian Sun. 2015. Deep Residual Learning for Image Recognition. *CoRR* abs/1512.03385 (2015). arXiv:1512.03385 http://arxiv.org/abs/1512.03385

[68] Tin Kam Ho. 1995. Random decision forests. In *Proceedings of 3rd International Conference on Document Analysis and Recognition*, Vol. 1. 278–282 vol.1. https://doi.org/10.1109/ICDAR.1995.598994

[69] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[70] Urs Hölzle, Craig Chambers, and David Ungar. 1992. Debugging Optimized Code with Dynamic Deoptimization. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation* (San Francisco, California, USA) *(PLDI '92)*. Association for Computing Machinery, New York, NY, USA, 32–43. https://doi.org/10.1145/143095.143114

[71] HotSpot JVM. 2023. HotSpot Runtime Overview. https://openjdk.org/groups/hotspot/docs/RuntimeOverview.html

[72] Sergey Ioffe and Christian Szegedy. 2015. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37* (Lille, France) *(ICML'15)*. JMLR.org, 448–456.

[73] Java 20. 2023. Java Language Specification. https://docs.oracle.com/javase/specs/jls/se20/html/index.html

[74] Tarindu Jayatilaka, Hideto Ueno, Giorgis Georgakoudis, EunJung Park, and Johannes Doerfert. 2021. Towards Compile-Time-Reducing Compiler Optimization Selection via Machine Learning. In *50th International Conference on Parallel Processing Workshop* (Lemont, IL, USA) *(ICPP Workshops '21)*. Association for Computing Machinery, New York, NY, USA, Article 23, 6 pages. https://doi.org/10.1145/3458744.3473355

[75] JVM 20. 2023. Java Virtual Machine Specification. https://docs.oracle.com/javase/specs/jvms/se20/html/

[76] JVMCI 2023. JCM Compiler Interface. https://openjdk.org/jeps/243

[77] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. 1996. Reinforcement Learning: A Survey. *J. Artif. Int. Res.* 4, 1 (May 1996), 237–285. https://doi.org/10.1613/jair.301

[78] Sebastian Kloibhofer. 2021. Run-Time Data Analysis to Drive Compiler Optimizations. In *Companion Proceedings of the 2021 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity* (Chicago, IL, USA) *(SPLASH Companion 2021)*. Association for Computing Machinery, New York, NY, USA, 9–12. https://doi.org/10.1145/3484271.3484974

[79] Sebastian Kloibhofer, Lukas Makor, David Leopoldseder, Daniele Bonetta, Lukas Stadler, and Hanspeter Mössenböck. 2023. Control Flow Duplication for Columnar Arrays in a Dynamic Compiler. *The Art, Science, and Engineering of Programming* 7 (2023). accepted for publication.

[80] P. Knijnenburg, T. Kisuki, and M. O'Boyle. 2002. *Iterative Compilation*. 171–187. https://doi.org/10.1007/3-540-45874-3_10

[81] Ron Kohavi. 1995. A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2* (Montreal, Quebec, Canada) *(IJCAI'95)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1137–1143.

[82] Thomas Kotzmann and Hanspeter Mössenböck. 2005. Escape Analysis in the Context of Dynamic Compilation and Deoptimization. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments* (Chicago, IL, USA) *(VEE '05)*. ACM, New York, NY, USA, 111–120. https://doi.org/10.1145/1064979.1064996

[83] Thomas Kotzmann and Hanspeter Mossenbock. 2007. Run-Time Support for Optimizations Based on Escape Analysis. In *International Symposium on Code Generation and Optimization (CGO'07)*. 49–60. https://doi.org/10.1109/CGO.2007.34

[84] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. 2008. Design of the Java HotSpot&Trade; Client Compiler for Java 6. *ACM Transactions on Architecture and Code Optimization* 5, 1, Article 7 (May 2008), 32 pages. https://doi.org/10.1145/1369396.1370017

[85] Jacob Kreindl, Daniele Bonetta, and Hanspeter Mössenböck. 2019. Towards Efficient, Multi-Language Dynamic Taint Analysis. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes* (Athens, Greece) *(MPLR 2019)*. Association for Computing Machinery, New York, NY, USA, 85–94. https://doi.org/10.1145/3357390.3361028

[86] Jacob Kreindl, Daniele Bonetta, Lukas Stadler, David Leopoldseder, and Hanspeter Mössenböck. 2020. Multi-Language Dynamic Taint Analysis in a Polyglot Virtual Machine. In *Proceedings of the 17th International Conference on Managed Programming Languages and Runtimes* (Virtual, UK) *(MPLR '20)*. Association for Computing Machinery, New York, NY, USA, 15–29. https://doi.org/10.1145/3426182.3426184

[87] Jacob Kreindl, Daniele Bonetta, Lukas Stadler, David Leopoldseder, and Hanspeter Mössenböck. 2021. Low-Overhead Multi-Language Dynamic Taint Analysis on

Managed Runtimes through Speculative Optimization. In *Proceedings of the 18th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes* (Münster, Germany) *(MPLR 2021)*. Association for Computing Machinery, New York, NY, USA, 70–87. https://doi.org/10.1145/3475738.3480939

[88] Jacob Kreindl, Daniele Bonetta, Lukas Stadler, David Leopoldseder, and Hanspeter Mössenböck. 2022. Dynamic Taint Analysis with Label-Defined Semantics. In *Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes* (Brussels, Belgium) *(MPLR '22)*. Association for Computing Machinery, New York, NY, USA, 64–84. https://doi.org/10.1145/3546918.3546927

[89] Michael Kruse, Hal Finkel, and Xingfu Wu. 2020. Autotuning Search Space for Loop Transformations. *2020 IEEE/ACM 6th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC) and Workshop on Hierarchical Parallelism for Exascale Computing (HiPar)* (2020), 12–22.

[90] Sameer Kulkarni. 2014. *Improving Compiler Optimizations using Machine Learning*. Ph. D. Dissertation. Newark, DE, USA. https://udspace.udel.edu/items/3e5914d3-1579-46fc-b8aa-7fe406088741

[91] Sameer Kulkarni and John Cavazos. 2012. Mitigating the Compiler Optimization Phase-Ordering Problem Using Machine Learning. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Tucson, Arizona, USA) *(OOPSLA '12)*. Association for Computing Machinery, New York, NY, USA, 147–162. https://doi.org/10.1145/2384616.2384628

[92] Chris Lattner and Vikram Adve. 2004. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 75–86. https://doi.org/10.1109/CGO.2004.1281665

[93] Jeremy Lau, Matthew Arnold, Michael Hind, and Brad Calder. 2006. Online Performance Auditing: Using Hot Optimizations without Getting Burned. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Ottawa, Ontario, Canada) *(PLDI '06)*. Association for Computing Machinery, New York, NY, USA, 239–251. https://doi.org/10.1145/1133981.1134010

[94] Raquel Lazcano, Daniel Madroñal, Eduardo Juarez, and Philippe Clauss. 2020. Runtime Multi-Versioning and Specialization inside a Memoized Speculative Loop Optimizer. In *Proceedings of the 29th International Conference on Compiler Construction*

(San Diego, CA, USA) *(CC 2020)*. Association for Computing Machinery, New York, NY, USA, 96–107. https://doi.org/10.1145/3377555.3377886

[95] Hugh Leather. 2010. *Machine Learning in Compilers*. Ph. D. Dissertation. Edinburgh, Scotland, UK. https://era.ed.ac.uk/handle/1842/9810

[96] Hugh Leather and Chris Cummins. 2020. Machine Learning in Compilers: Past, Present and Future. In *2020 Forum for Specification and Design Languages (FDL)*. IEEE Computer Society, 1–8. https://doi.org/10.1109/FDL50818.2020.9232934

[97] David Leopoldseder. 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* (Vancouver, BC, Canada) *(SPLASH Companion 2017)*. Association for Computing Machinery, New York, NY, USA, 10–12. https://doi.org/10.1145/3135932.3135935

[98] David Leopoldseder, Roland Schatz, Lukas Stadler, Manuel Rigger, Thomas Würthinger, and Hanspeter Mössenböck. 2018. Fast-Path Loop Unrolling of Non-Counted Loops to Enable Subsequent Compiler Optimizations. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes* (Linz, Austria) *(ManLang '18)*. Association for Computing Machinery, New York, NY, USA, Article 2, 13 pages. https://doi.org/10.1145/3237009.3237013

[99] David Leopoldseder, Lukas Stadler, Manuel Rigger, Thomas Würthinger, and Hanspeter Mössenböck. 2018. A Cost Model for a Graph-Based Intermediate-Representation in a Dynamic Compiler. In *Proceedings of the 10th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages* (Boston, MA, USA) *(VMIL 2018)*. Association for Computing Machinery, New York, NY, USA, 26–35. https://doi.org/10.1145/3281287.3281290

[100] David Leopoldseder, Lukas Stadler, Thomas Würthinger, Josef Eisl, Doug Simon, and Hanspeter Mössenböck. 2018. Dominance-Based Duplication Simulation (DBDS): Code Duplication to Enable Compiler Optimizations. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization* (Vienna, Austria) *(CGO 2018)*. Association for Computing Machinery, New York, NY, USA, 126–137. https://doi.org/10.1145/3168811

[101] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. 2015. Many-Core Compiler Fuzzing. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA)

(PLDI '15). Association for Computing Machinery, New York, NY, USA, 65–76. https://doi.org/10.1145/2737924.2737986

[102] Lukas Makor. 2021. Run-Time Data Analysis in Dynamic Runtimes. In *Companion Proceedings of the 2021 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity* (Chicago, IL, USA) *(SPLASH Companion 2021)*. Association for Computing Machinery, New York, NY, USA, 6–8. https://doi.org/10.1145/3484271.3484973

[103] Lukas Makor, Sebastian Kloibhofer, David Leopoldseder, Daniele Bonetta, Lukas Stadler, and Hanspeter Mössenböck. 2022. Automatic Array Transformation to Columnar Storage At Run Time. In *Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes* (Brussels, Belgium) *(MPLR '22)*. Association for Computing Machinery, New York, NY, USA, 16–28. https://doi.org/10.1145/3546918.3546919

[104] Rahim Mammadli, Ali Jannesari, and Felix A. Wolf. 2020. Static Neural Compiler Optimization via Deep Reinforcement Learning. *2020 IEEE/ACM 6th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC) and Workshop on Hierarchical Parallelism for Exascale Computing (HiPar)* (2020), 1–11.

[105] Rahim Mammadli, Marija Selakovic, Felix Wolf, and Michael Pradel. 2021. Learning to Make Compiler Optimizations More Effective. In *Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming* (Virtual, Canada) *(MAPS 2021)*. Association for Computing Machinery, New York, NY, USA, 9–20. https://doi.org/10.1145/3460945.3464952

[106] Charith Mendis. 2020. *Towards Automated Construction of Compiler Optimizations*. Ph. D. Dissertation. Cambridge, MA, USA. https://groups.csail.mit.edu/commit/papers/2020/mendis-thesis.pdf

[107] Charith Mendis and Saman Amarasinghe. 2018. GoSLP: Globally Optimized Superword Level Parallelism Framework. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 110 (oct 2018), 28 pages. https://doi.org/10.1145/3276480

[108] Charith Mendis, Alex Renda, Dr.Saman Amarasinghe, and Michael Carbin. 2019. Ithemal: Accurate, Portable and Fast Basic Block Throughput Estimation using Deep Neural Networks. In *Proceedings of the 36th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, 4505–4515. http://proceedings.mlr.press/v97/mendis19a.html

[109] Charith Mendis, Cambridge Yang, Yewen Pu, Saman Amarasinghe, and Michael Carbin. 2019. *Compiler Auto-Vectorization with Imitation Learning*. Curran Associates Inc., Red Hook, NY, USA. https://proceedings.neurips.cc/paper/2019/file/d1d5923fc822531bbfd9d87d4760914b-Paper.pdf

[110] Raphael Mosaner. 2020. Machine Learning to Ease Understanding of Data Driven Compiler Optimizations. In *Companion Proceedings of the 2020 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity* (Virtual, USA) *(SPLASH Companion 2020)*. Association for Computing Machinery, New York, NY, USA, 4–6. https://doi.org/10.1145/3426430.3429451

[111] Raphael Mosaner, Gergö Barany, David Leopoldseder, and Hanspeter Mössenböck. 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* (Auckland, New Zealand) *(VMIL 2022)*. Association for Computing Machinery, New York, NY, USA, 36–47. https://doi.org/10.1145/3563838.3567679

[112] Raphael Mosaner, David Leopoldseder, Wolfgang Kisling, Lukas Stadler, and Hanspeter Mössenböck. 2022. Machine-Learning-Based Self-Optimizing Compiler Heuristics. In *Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes* (Brussels, Belgium) *(MPLR '22)*. Association for Computing Machinery, New York, NY, USA, 98–111. https://doi.org/10.1145/3546918.3546921

[113] Raphael Mosaner, David Leopoldseder, Wolfgang Kisling, Lukas Stadler, and Hanspeter Mössenböck. 2022. Compilation Forking: A Fast and Flexible Way of Generating Data for Compiler-Internal Machine Learning Tasks. *The Art, Science, and Engineering of Programming* 7 (2022). https://doi.org/10.22152/programming-journal.org/2023/7/3

[114] Raphael Mosaner, David Leopoldseder, Manuel Rigger, Roland Schatz, and Hanspeter Mössenböck. 2019. Supporting On-Stack Replacement in Unstructured Languages by Loop Reconstruction and Extraction. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes* (Athens, Greece) *(MPLR 2019)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3357390.3361030

[115] Raphael Mosaner, David Leopoldseder, and Lukas Stadler. 2022. Online Machine Learning Based Compilation. U.S. Patent Number 11.392.356, filed February 26th, 2021, Issued July 19th, 2022.

[116] Raphael Mosaner, David Leopoldseder, Lukas Stadler, and Hanspeter Mössenböck. 2021. Using Machine Learning to Predict the Code Size Impact of Duplication Heuristics in a Dynamic Compiler. In *Proceedings of the 18th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR 2021)*. Association for Computing Machinery, 127–135. https://doi.org/10.1145/3475738.3480943

[117] Hanspeter Mössenböck. 2000. *Adding static single assignment form and a graph coloring register allocator to the Java HotSpotTM client compiler*. Technical Report. Citeseer. https://ssw.jku.at/Research/Reports/Report15.PDF

[118] Hanspeter Mössenböck and Michael Pfeiffer. 2002. Linear Scan Register Allocation in the Context of SSA Form and Register Constraints. In *Compiler Construction*, R. Nigel Horspool (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 229–246.

[119] Paschalis Mpeis, Pavlos Petoumenos, Kim Hazelwood, and Hugh Leather. 2021. Developer and User-Transparent Compiler Optimization for Interactive Applications. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 268–281. https://doi.org/10.1145/3453483.3454043

[120] ONNX 2021. ONNX Runtime. https://onnxruntime.ai/.

[121] OpenJDK. 2023. OpenJDK. https://openjdk.org/

[122] Michael Paleczny, Christopher Vick, and Cliff Click. 2001. The Java HotspotTM Server Compiler. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1* (Monterey, California) *(JVM'01)*. USENIX Association, USA, 1.

[123] Eunjung Park, John Cavazos, and Marco A. Alvarez. 2012. Using Graph-Based Program Characterization for Predictive Modeling. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization* (San Jose, California) *(CGO '12)*. Association for Computing Machinery, New York, NY, USA, 196–206. https://doi.org/10.1145/2259016.2259042

[124] Eunjung Park, Sameer Kulkarni, and John Cavazos. 2011. An Evaluation of Different Modeling Techniques for Iterative Compilation. In *Proceedings of the 14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems* (Taipei, Taiwan) *(CASES '11)*. Association for Computing Machinery, New York, NY, USA, 65–74. https://doi.org/10.1145/2038698.2038711

[125] Sunghyun Park, Salar Latifi, Yongjun Park, Armand Behroozi, Byungsoo Jeon, and Scott Mahlke. 2022. SRTuner: Effective Compiler Optimization Customization by Exposing Synergistic Relations. In *Proceedings of the 20th IEEE/ACM International Symposium on Code Generation and Optimization* (Virtual Event, Republic of Korea) *(CGO '22)*. IEEE Press, 118–130. https://doi.org/10.1109/CGO53902.2022.9741263

[126] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019), 8026–8037. https://dl.acm.org/doi/10.5555/3454287.3455008

[127] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Peter Prettenhofer, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *the Journal of machine Learning research* 12 (2011), 2825–2830. https://www.jmlr.org/papers/volume12/pedregosa11a/pedregosa11a.pdf

[128] Filip Pizlo. 2014. JetStream Benchmark Suite. http://browserbench.org/JetStream/

[129] Adam Pocock. 2021. Tribuo: Machine Learning with Provenance in Java. (2021). https://doi.org/10.48550/ARXIV.2110.03022

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

[131] Manuel Rigger. 2016. Sulong: Memory Safe and Efficient Execution of LLVM-Based Languages. In *ECOOP 2016 Doctoral Symposium* (Rome, Italy) *(ECOOP DS 2016)*. http://ssw.jku.at/General/Staff/ManuelRigger/ECOOP16-DS.pdf

[132] Manuel Rigger. 2018. Sandboxed Execution of C and Other Unsafe Languages on the Java Virtual Machine. In *Companion Proceedings of the 2nd International Conference on the Art, Science, and Engineering of Programming* (Nice, France) *(Programming '18)*. Association for Computing Machinery, New York, NY, USA, 227–229. https://doi.org/10.1145/3191697.3213795

[133] Manuel Rigger, Matthias Grimmer, and Hanspeter Mössenböck. 2016. Sulong - Execution of LLVM-Based Languages on the JVM: Position Paper. In *Proceedings of the 11th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems* (Rome, Italy) *(ICOOOLPS '16)*. Association for Computing Machinery, New York, NY, USA, Article 7, 4 pages. https://doi.org/10.1145/3012408.3012416

[134] Manuel Rigger, Matthias Grimmer, Christian Wimmer, Thomas Würthinger, and Hanspeter Mössenböck. 2016. Bringing Low-Level Languages to the JVM: Efficient Execution of LLVM IR on Truffle. In *Proceedings of the 8th International Workshop on Virtual Machines and Intermediate Languages* (Amsterdam, Netherlands) *(VMIL 2016)*. Association for Computing Machinery, New York, NY, USA, 6–15. https://doi.org/10.1145/2998415.2998416

[135] Manuel Rigger, Roland Schatz, Matthias Grimmer, and Hanspeter Mössenböck. 2017. Lenient Execution of C on a Java Virtual Machine: Or: How I Learned to Stop Worrying and Run the Code. In *Proceedings of the 14th International Conference on Managed Languages and Runtimes* (Prague, Czech Republic) *(ManLang 2017)*. Association for Computing Machinery, New York, NY, USA, 35–47. https://doi.org/10.1145/3132190.3132204

[136] Manuel Rigger, Roland Schatz, Jacob Kreindl, Christian Häubl, and Hanspeter Mössenböck. 2018. Sulong, and Thanks for All the Fish. In *Companion Proceedings of the 2nd International Conference on the Art, Science, and Engineering of Programming* (Nice, France) *(Programming '18)*. Association for Computing Machinery, New York, NY, USA, 58–60. https://doi.org/10.1145/3191697.3191726

[137] Manuel Rigger, Roland Schatz, René Mayrhofer, Matthias Grimmer, and Hanspeter Mössenböck. 2018. Sulong, and Thanks for All the Bugs: Finding Errors in C Programs by Abstracting from the Native Execution Model. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Williamsburg, VA, USA) *(ASPLOS '18)*. Association for Computing Machinery, New York, NY, USA, 377–391. https://doi.org/10.1145/3173162.3173174

[138] Barry Rosen, Mark Wegman, and Kenneth Zadeck. 1988. Global Value Numbers and Redundant Computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '88)*. Association for Computing Machinery, New York, NY, USA, 12–27. https://doi.org/10.1145/73560.73562

[139] Safepoint 2023. Safepoints in Java. https://medium.com/software-under-the-hood/under-the-hood-java-peak-safepoints-dd45af07d766

[140] Ricardo Nabinger Sanchez, Jose Nelson Amaral, Duane Szafron, Marius Pirvu, and Mark Stoodley. 2011. Using Machines to Learn Method-Specific Compilation Strategies. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '11)*. IEEE Computer Society, USA, 257–266. https://doi.org/10.1109/CGO.2011.5764693

[141] Jürgen Schmidhuber. 2015. Deep learning in neural networks: An overview. *Neural Networks* 61 (2015), 85–117. https://doi.org/10.1016/j.neunet.2014.09.003

[142] Andreas Sewe, Mira Mezini, Aibek Sarimbekov, and Walter Binder. 2011. Da Capo Con Scala: Design and Analysis of a Scala Benchmark Suite for the Java Virtual Machine. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Portland, Oregon, USA) *(OOPSLA '11)*. Association for Computing Machinery, New York, NY, USA, 657–676. https://doi.org/10.1145/2048066.2048118

[143] Douglas Simon, John Cavazos, Christian Wimmer, and Sameer Kulkarni. 2013. Automatic Construction of Inlining Heuristics Using Machine Learning. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO '13)*. IEEE Computer Society, Washington, DC, USA, 1–12. https://doi.org/10.1109/CGO.2013.6495004

[144] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *J. Mach. Learn. Res.* 15, 1 (jan 2014), 1929–1958.

[145] Lukas Stadler, Gilles Duboscq, Hanspeter Mössenböck, and Thomas Würthinger. 2012. Compilation Queuing and Graph Caching for Dynamic Compilers. In *Proceedings of the Sixth ACM Workshop on Virtual Machines and Intermediate Languages* (Tucson, Arizona, USA) *(VMIL '12)*. Association for Computing Machinery, New York, NY, USA, 49–58. https://doi.org/10.1145/2414740.2414750

[146] Lukas Stadler, Gilles Duboscq, Hanspeter Mössenböck, Thomas Würthinger, and Doug Simon. 2013. An Experimental Study of the Influence of Dynamic Compiler Optimizations on Scala Performance. In *Proceedings of the 4th Workshop on Scala* (Montpellier, France) *(SCALA '13)*. Association for Computing Machinery, New York, NY, USA, Article 9, 8 pages. https://doi.org/10.1145/2489837.2489846

[147] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. 2018. Partial Escape Analysis and Scalar Replacement for Java. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Orlando, FL, USA) *(CGO '14)*. Association for Computing Machinery, New York, NY, USA, 165–174. https://doi.org/10.1145/2544137.2544157

[148] Kenneth O. Stanley and Risto Miikkulainen. 2002. Evolving Neural Networks through Augmenting Topologies. *Evolutionary Computation* 10, 2 (2002), 99–127. https://doi.org/10.1162/106365602320169811

[149] Mark Stephenson and Saman Amarasinghe. 2005. Predicting unroll factors using supervised classification. In *International Symposium on Code Generation and Optimization*. IEEE Computer Society, 123–134. https://doi.org/10.1109/CGO.2005.29

[150] Mark Stephenson, Saman Amarasinghe, Martin Martin, and Una-May O'Reilly. 2003. Meta Optimization: Improving Compiler Heuristics with Machine Learning. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '03)*. Association for Computing Machinery, New York, NY, USA, 77–90. https://doi.org/10.1145/781131.781141

[151] Michele Tartara and Stefano Crespi Reghizzi. 2013. Continuous Learning of Compiler Heuristics. *ACM Trans. Archit. Code Optim.* 9, 4, Article 46 (Jan. 2013), 25 pages. https://doi.org/10.1145/2400682.2400705

[152] Eclipse Deeplearning4j Development Team. 2016. DL4J: Deep Learning for Java. https://github.com/eclipse/deeplearning4j

[153] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. 2010. Collecting Performance Data with PAPI-C. In *Tools for High Performance Computing 2009*, Matthias S. Müller, Michael M. Resch, Alexander Schulz, and Wolfgang E. Nagel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 157–173.

[154] Mircea Trofin, Yundi Qian, Eugene Brevdo, Zinan Lin, Krzysztof Choromanski, and David Li. 2021. MLGO: a Machine Learning Guided Compiler Optimizations Framework. (2021). https://doi.org/10.48550/ARXIV.2101.04808

[155] Foivos Tsimpourlas, Pavlos Petoumenos, Min Xu, Chris Cummins, Kim Hazelwood, Ajitha Rajan, and Hugh Leather. 2023. BenchPress: A Deep Active Benchmark Generator. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques* (Chicago, Illinois) *(PACT '22)*. Association for Computing Machinery, New York, NY, USA, 505–516. https://doi.org/10.1145/3559009.3569644

[156] Huanting Wang, Zhanyong Tang, Cheng Zhang, Jiaqi Zhao, Chris Cummins, Hugh Leather, and Zheng Wang. 2022. Automating Reinforcement Learning Architecture Design for Code Optimization. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction* (Seoul, South Korea) *(CC 2022)*. Association for Computing Machinery, New York, NY, USA, 129–143. https://doi.org/10.1145/3497776.3517769

[157] Zheng Wang and Michael O'Boyle. 2018. Machine Learning in Compiler Optimization. *Proc. IEEE* 106, 11 (Nov 2018), 1879–1901. https://doi.org/10.1109/JPROC.2018.2817118

[158] Geoffrey S. Watson. 1967. Linear Least Squares Regression. *The Annals of Mathematical Statistics* 38, 6 (1967), 1679 – 1699. https://doi.org/10.1214/aoms/1177698603

[159] Christian Wimmer, Michael Haupt, Michael L. Van De Vanter, Mick Jordan, Laurent Daynès, and Douglas Simon. 2013. Maxine: An Approachable Virtual Machine for, and in, Java. *ACM Trans. Archit. Code Optim.* 9, 4, Article 30 (jan 2013), 24 pages. https://doi.org/10.1145/2400682.2400689

[160] Christian Wimmer and Hanspeter Mössenböck. 2005. Optimized Interval Splitting in a Linear Scan Register Allocator. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments* (Chicago, IL, USA) *(VEE '05)*. ACM, New York, NY, USA, 132–141. https://doi.org/10.1145/1064979.1064998

[161] Christian Wimmer and Hanspeter Mössenböck. 2006. Automatic Object Colocation Based on Read Barriers. In *Proceedings of the 7th Joint Conference on Modular Programming Languages* (Oxford, UK) *(JMLC'06)*. Springer-Verlag, Berlin, Heidelberg, 326–345.

[162] Christian Wimmer and Hanspeter Mössenböck. 2007. Automatic Feedback-Directed Object Inlining in the Java Hotspot™ Virtual Machine. In *Proceedings of the 3rd International Conference on Virtual Execution Environments* (San Diego, California, USA) *(VEE '07)*. Association for Computing Machinery, New York, NY, USA, 12–21. https://doi.org/10.1145/1254810.1254813

[163] Christian Wimmer and Hanspeter Mössenböck. 2008. Automatic Array Inlining in Java Virtual Machines. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Boston, MA, USA) *(CGO '08)*. Association for Computing Machinery, New York, NY, USA, 14–23. https://doi.org/10.1145/1356058.1356061

[164] Christian Wimmer and Hanspeter Mössenbösck. 2010. Automatic Feedback-Directed Object Fusing. *ACM Trans. Archit. Code Optim.* 7, 2, Article 7 (oct 2010), 35 pages. https://doi.org/10.1145/1839667.1839669

[165] Christian Wimmer and Thomas Würthinger. 2012. Truffle: A Self-Optimizing Runtime System. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity* (Tucson, Arizona, USA) *(SPLASH '12)*. Association for Computing Machinery, New York, NY, USA, 13–14. https://doi.org/10.1145/2384716.2384723

[166] Thomas Würthinger, Danilo Ansaloni, Walter Binder, Christian Wimmer, and Hanspeter Mössenböck. 2011. Safe and Atomic Run-Time Code Evolution for Java and Its Application to Dynamic AOP. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Portland, Oregon, USA) *(OOPSLA '11)*. Association for Computing Machinery, New York, NY, USA, 825–844. https://doi.org/10.1145/2048066.2048129

[167] Thomas Würthinger, Walter Binder, Danilo Ansaloni, Philippe Moret, and Hanspeter Mössenböck. 2010. Applications of Enhanced Dynamic Code Evolution for Java in GUI Development and Dynamic Aspect-Oriented Programming. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering* (Eindhoven, The Netherlands) *(GPCE '10)*. Association for Computing Machinery, New York, NY, USA, 123–126. https://doi.org/10.1145/1868294.1868312

[168] Thomas Würthinger, Walter Binder, Danilo Ansaloni, Philippe Moret, and Hanspeter Mössenböck. 2010. Improving Aspect-Oriented Programming with Dynamic Code Evolution in an Enhanced Java Virtual Machine. In *Proceedings of the 7th Workshop on Reflection, AOP and Meta-Data for Software Evolution* (Maribor, Slovenia) *(RAM-SE '10)*. Association for Computing Machinery, New York, NY, USA, Article 5, 5 pages. https://doi.org/10.1145/1890683.1890688

[169] Thomas Würthinger, Christian Wimmer, and Lukas Stadler. 2013. Unrestricted and Safe Dynamic Code Evolution for Java. *Sci. Comput. Program.* 78, 5 (may 2013), 481–498. https://doi.org/10.1016/j.scico.2011.06.005

[170] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to Rule Them All. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (Indianapolis, Indiana, USA) *(Onward! 2013)*. Association for Computing Machinery, New York, NY, USA, 187–204. https://doi.org/10.1145/2509578.2509581

[171] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. 2012. Self-Optimizing AST Interpreters. In *Proceedings of the 8th Symposium on Dynamic Languages* (Tucson, Arizona, USA) *(DLS '12)*. Association for Computing Machinery, New York, NY, USA, 73–82. https://doi.org/10.1145/2384577.2384587

[172] Mingzhou Zhou, Xipeng Shen, Yaoqing Gao, and Graham Yiu. 2014. Space-Efficient Multi-Versioning for Input-Adaptive Feedback-Driven Program Optimizations. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (Portland, Oregon, USA) *(OOPSLA '14)*. Association for Computing Machinery, New York, NY, USA, 763–776. https://doi.org/10.1145/2660193.2660229

Acknowledgements

The slight touch of pride is nothing compared to the humility and gratefulness I am feeling while writing these last lines of my thesis. Pride, as I am about to finish my PhD. Humility, as I could not have finished it on my own. Gratefulness, as there are so many people whose support I could count on during this time.

Most of my gratitude belongs to my supervisor Prof. Hanspeter Mössenböck, whose guidance constantly encouraged me to reflect on my own work, with whom discussions always ended in expanding my own horizon and who always provided invaluable feedback and improvements to my scientific work, even on short notice before paper deadlines.

I also want to thank Prof. Andreas Krall for his readiness to be the second reviewer of my thesis and a member of the defense senate.

My infinite gratefulness belongs to David Leopoldseder, who co-supervised me as part of the Oracle Labs research collaboration. David, thank you for your endless expertise on technical issues, but more importantly: Thank you for your steady encouragements during times of setback. Your words always motivated me to keep on going.

Many thanks belong to Wolfgang Kisling, who I supervised during his Master's and who did a great job when working on our machine learning backend.

I am also thankful to my colleagues from the SSW; for the valuable technical discussions and even more for the less valuable but always amusing office chitchat. Here, I want to especially highlight my former and current fellow PhD students: Markus Weninger, Andreas Schörgenhumer, Florian Latifi, Jacob Kreindl, Sebastian Kloibhofer, Lukas Makor and Christoph Pichler.

Special thanks to Thomas Würthinger and Oracle Labs; being part of a research collaboration with a renowned company has been a great pleasure for me. I also want to thank some

people from Oracle labs who provided frequent advice to my research or collaborated with me when publishing papers: Lukas Stadler, Adam Pocock, Gergö Barany and Milan Cugurovic.

Doing a PhD can blur the line between professional and private life. Thus, I want to express my deepest gratitude to my family and friends who always stood by my side, whether it was to provide moral support or to rejoice with me in times of success. More importantly, you are giving me the feeling that this will not change in any future to come: Thank you!