SPLASH: G: 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 input. 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. 2021. SPLASH: G: Machine Learning to Ease Understanding of Data Driven Compiler Optimizations. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 [2, 10, 12]—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.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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

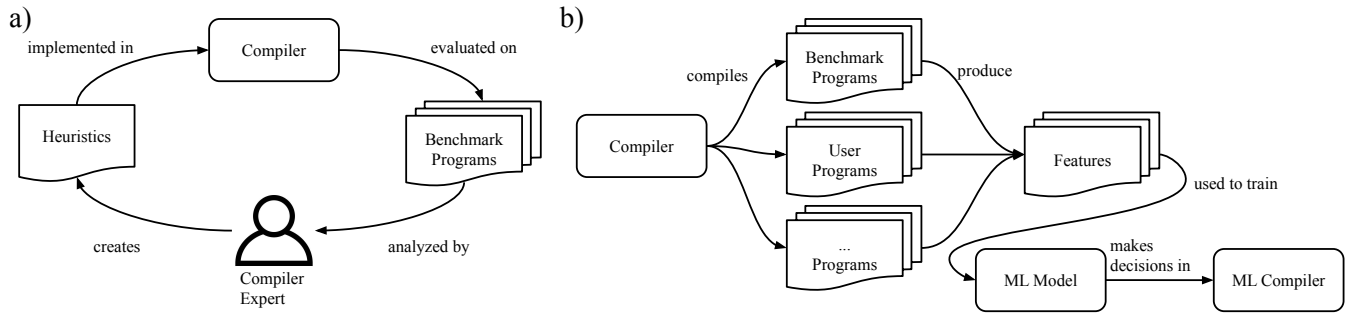

Figure 1. Workflows in existing compilers. a) depicts the iterative process of compiler experts optimizing heuristics and b) illustrates the traditional black-box approach when machine learning is employed in compilers.

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], but none in a dynamic, production level compiler. 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

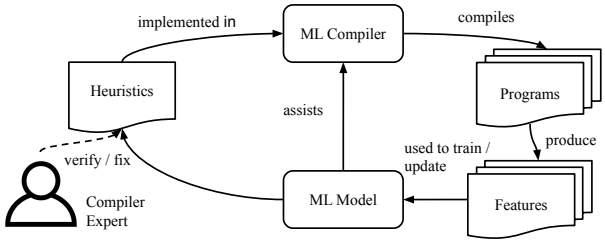

Figure 2. Workflow of assistive machine learning in compilers. Machine learning is used to automatically provide feedback based on observed compilation data.

- *understandability* of compiler internals (data analysis)

More general, a performance improvement is tied to the performance metrics of the underlying optimization, which 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 in the soft skill set 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 present a case study on how to improve an existing compiler optimization with assistance of machine learning. The targeted optimization is *code duplication* [5] shown in Figure 3. Its idea is to copy code at control flow merges (line 3 in Figure 3a) into the predecessors blocks (Figure 3b), which can enable further optimizations (Figure 3c).

When executed prematurely, duplication can cause huge code size bloats. Thus, Leopoldseder et al. [6] present a trade-off heuristic between estimated code growth and estimated number of saved execution 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. However, it resembles a linear regression like model, where estimates for total code size and performance impact

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

```
1 if(x > 0) phi = x
2 else phi = 0
3 return phi + 2
```

(a) Initial code.

```
1 if(x > 0) return x + 2
2 else return 0 + 2
```

(b) Duplication ...

```
1 if(x > 0) return x + 2
2 else return 2
```

(c) ...enables constant folding

Figure 3. Code duplication example, modified from [5]

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

Function	#AddNode	#IfNode	Feature_X	codeSize
$\text{f1}_{\text{benchX}}$	27	8	...	924
$\text{f2}_{\text{benchX}}$	16	1	...	438
$\text{f1}_{\text{benchY}}$	4	0	...	102

of a duplication are calculated as

$$\sum_{n \in G} n * cost(n)$$

In this function, n is one node in the (sub-)graph G to be duplicated and $cost(n)$ is either abstract size or cycles as defined in the node cost model. In reality we would expect a non-linear relationship between the compiler graph after duplication and the final code size. This is because of subsequent compiler phases which manipulate the graph and in further consequence the nodes which are turned into machine code. While a linear model might approximate this non-linear relationship, finding the right coefficients (i.e. abstract code size and abstract cycles for each node type as defined in the node cost model) might be a tedious and error prone task, as subsequent compiler phases have to be taken into account.

In this case study, we want to show how the existing node cost model could be improved, by comparing its estimations to predictions performed by a machine learning model. More accurately, we evaluated the code size impact heuristic of the cost model. Therefore, we trained an ANN for learning the non-linear relationship between the number of IR nodes at the time of duplication and the code size *after subsequent optimization phases*. The input vector is depicted in Table 1. For generating data, we used the benchmark suites from Section 3.1 and gathered over 300.000 data points over several benchmark executions. The Feature vectors, including node counts and code size label are extracted using Graal's

debug phases. Figure 4 shows the accuracy of the resulting neural network, visualized as bar plot . 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 are less than 10% off from the target label. To find misbehavior in

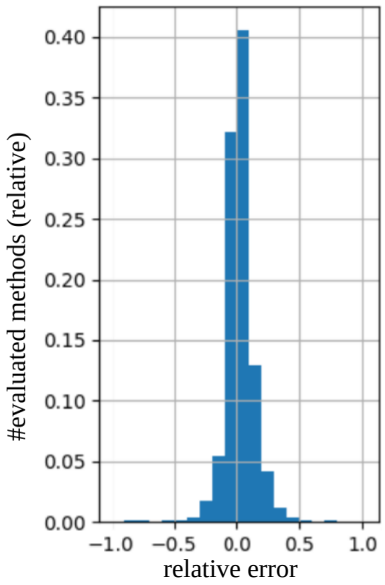

Figure 4. Prediction accuracy of our neural network.

the existing node cost model, we implemented our prototype machine learning predictor in Graal in a way, that both models are executed in parallel. 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.

3.3 Results

Using ML as an assistive technology, both speed-ups and code size reductions could be reported as a result of fixing several misclassifications in the human-crafted model. Figure 5 shows a small subset of benchmarks, where interesting patterns can be seen. Four different configurations are depicted. *Default* is the configuration which is currently used in GraalVM, running on millions of devices worldwide. *Fixed* shows GraalVM after applying fixes to the node cost model, which were motivated by our research using machine learning. *ML* is GraalVM but with our ML model replacing the node cost model when predicting code size impact. Finally, the *NoDup* configuration is the baseline, where duplication in Graal is disabled.

Both, jetstream's *towers* and *containers* benchmarks show severe underestimations of the resulting code size with the default Graal configuration. These were caused by wrong node costs for loop related nodes and *ReturnNodes*. Other benchmarks could be improved by finding and fixing bugs in the node cost model implementation which were unveiled due to vastly different decisions between ML model and node cost model. For some other benchmarks, like *richards* and *typescript*, the non-linear ML model simply outperforms the linear regression like node cost model. *Gameboy* shows a different behavior, where due to the reduction of code size, the performance is degraded. Other benchmarks also had slowdowns or code size growths when executed with our machine learning model. This shows, that it is very hard to create a consistent yet understandable machine learning model for a given task.

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 giving 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, where an existing, production-level heuristic could still be improved by using machine learning. In the future, we plan to transfer our approach also to other domains such as inlining or loop related optimizations. Especially, when it comes to predicting a performance impact, we assume that creating a training dataset with high precision yet little noise is a challenging task.

References

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

[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] 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 âĂŹ06)*. Association for Computing Machinery, New York, NY, USA, 169âĂŞ190. https://doi.org/10.1145/1167473.1167488

[4] 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*.

[5] 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*

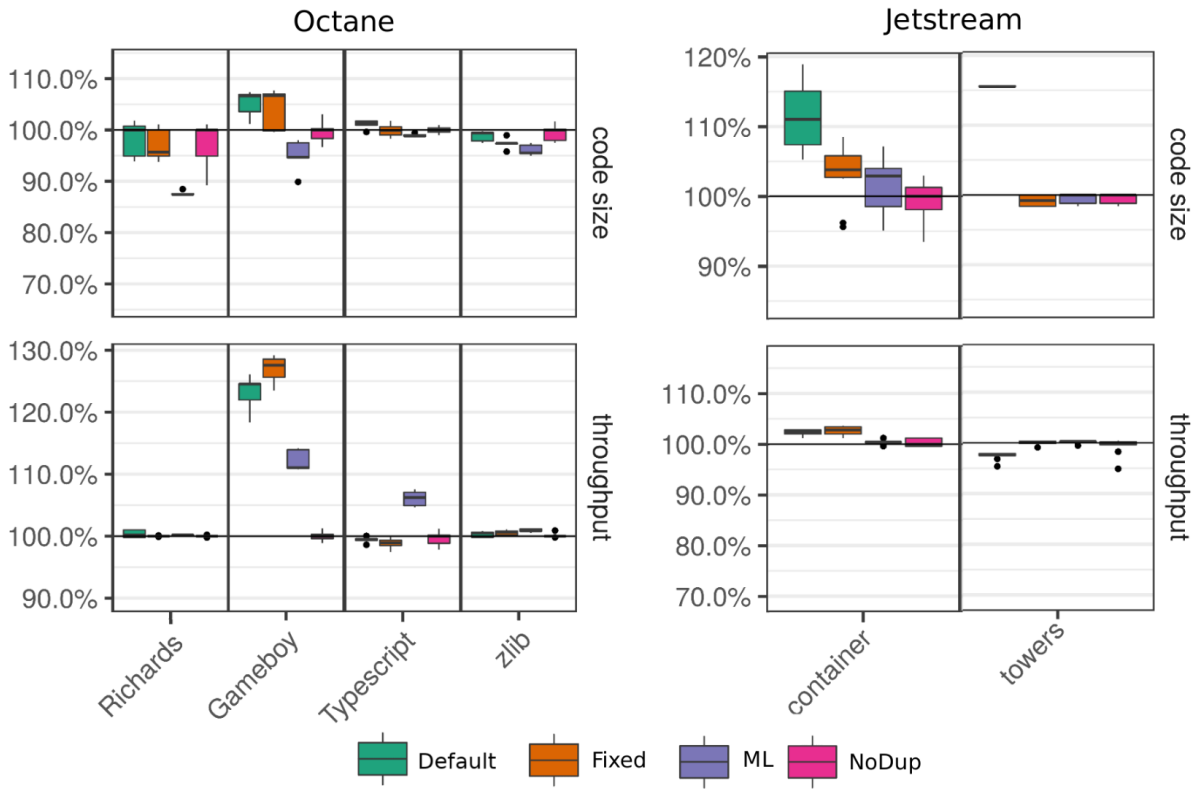

Figure 5. Selection of benchmark results.

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

[6] 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, 126âĂŞ137. https://doi.org/10.1145/3168811

[7] 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

[8] 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, 31âĂŞ47. https://doi.org/10.1145/3314221.3314637

[9] 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 âĂŹ11)*. Association for Computing Machinery, New York, NY, USA, 657âĂŞ676. https://doi.org/10.1145/2048066.2048118

[10] 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) (CGO '13)*. IEEE Computer Society, Washington, DC, USA, 1–12. https://doi.org/10.1109/CGO.2013.6495004

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

[12] 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

[13] 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)*. ACM, New York, NY, USA, 187–204. https://doi.org/10.1145/2509578.2509581