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.


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

ACM Reference Format:

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 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,
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 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.

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 jet-stream’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


\(^2\)https://github.com/chromium/octane

\(^3\)https://browserbench.org/JetStream/