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
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9696-7/22/09.
https://doi.org/10.1145/3546918.3546921

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].

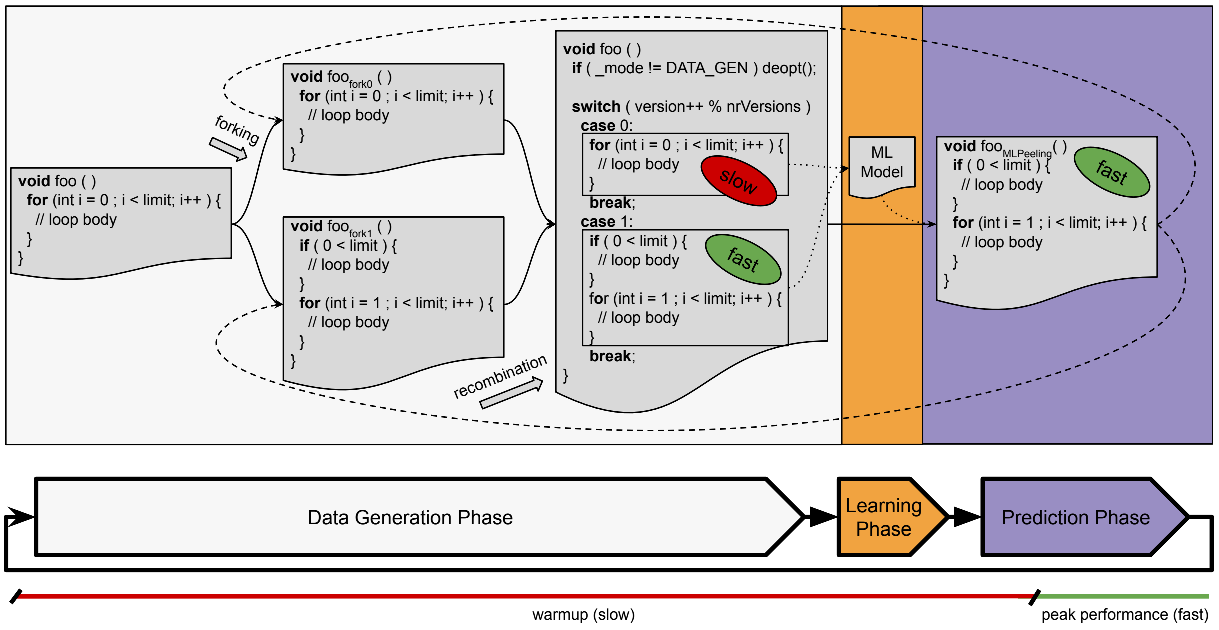
Our approach consists of three phases, two of which correspond directly to how the dynamic compilation is performed. These are the *data generation phase*, the *learning phase* and *prediction phase*. They can be iterated multiple times (see Figure 1), which enables iterative refinement or adjustment 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



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

```
void foo_fork0 ( )
  for (int i = 0 ; i < limit; i++ ) {
    // loop body
  }
}
```

```
void foo_fork1 ( )
  if ( 0 < limit ) {
    // loop body
  }
  for (int i = 1 ; i < limit; i++ ) {
    // loop body
  }
}
```

```
void foo ( )
  if ( _mode != DATA_GEN ) deopt();

  switch ( version++ % nrVersions )
  case 0:
    for (int i = 0 ; i < limit; i++ ) {
      // loop body
    }
  break;
  case 1:
    if ( 0 < limit ) {
      // loop body
    }
    for (int i = 1 ; i < limit; i++ ) {
      // loop body
    }
  break;
}
```

```
void foo_MLPeeling ( )
  if ( 0 < limit ) {
    // loop body
  }
  for (int i = 1 ; i < limit; i++ ) {
    // loop body
  }
}
```

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

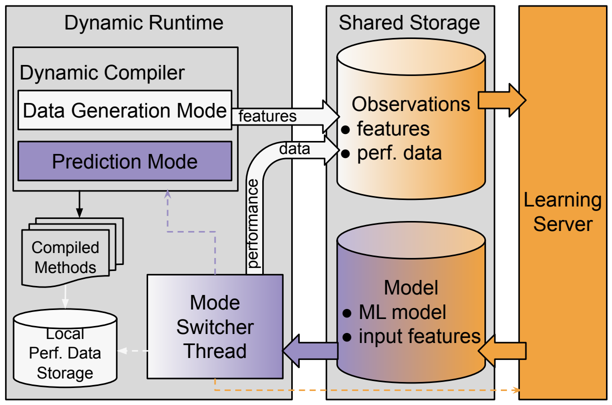


Figure 2: System architecture.

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.

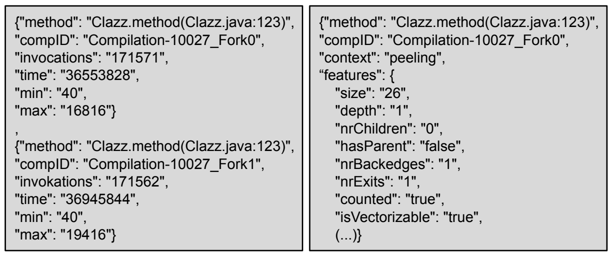
The learning server is invoked via a socket connection by sending either a `learn` or an `update` request. These requests also include the paths to the feature and performance data. The response contains the path to the model in the shared storage or a forwarded error message if training was not successful. 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).

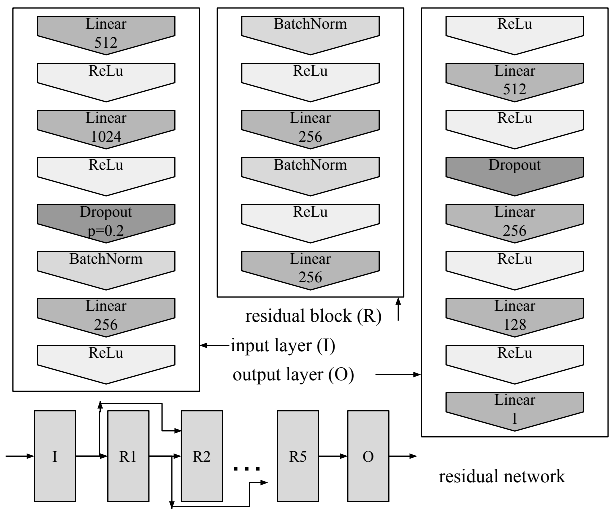


Figure 4: Residual network blocks and network structure.

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%.

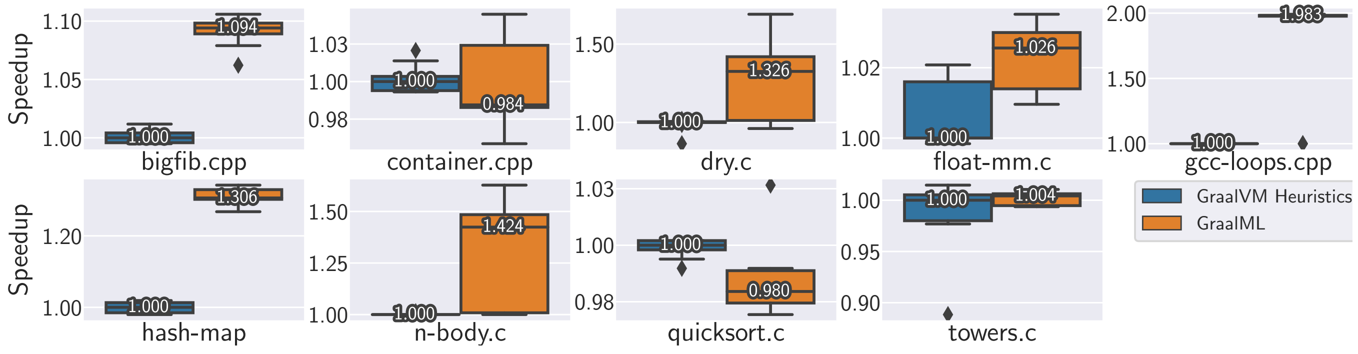


Figure 5: JetStream peak performance. Higher is better.

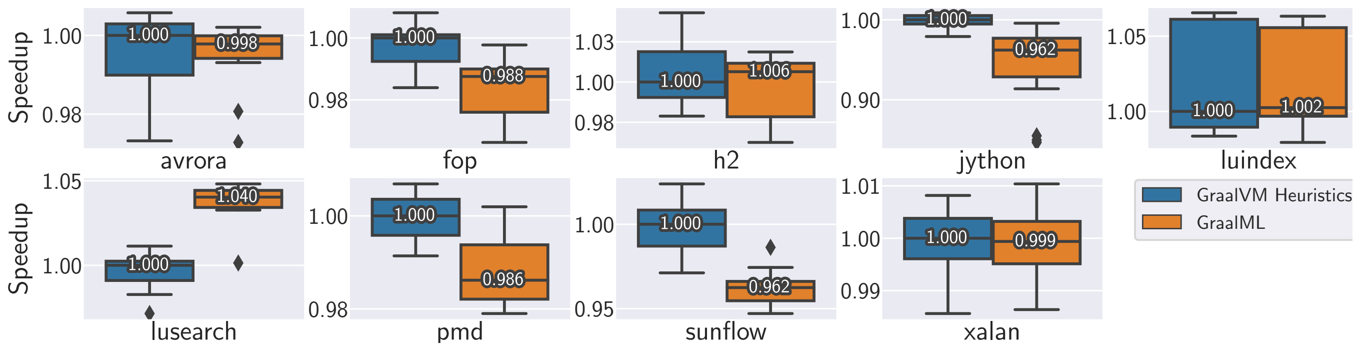


Figure 6: DaCapo peak performance. Higher is better.

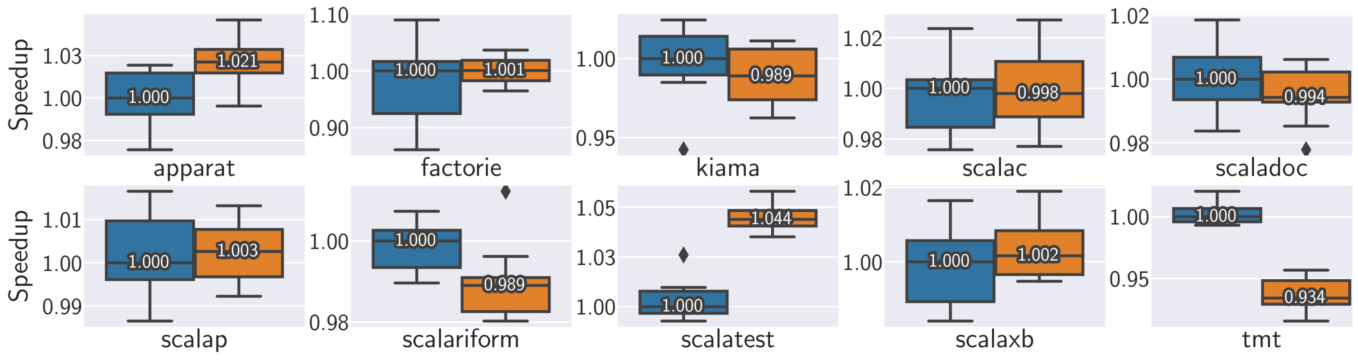


Figure 7: DaCapo Scala peak performance. Higher is better.

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.

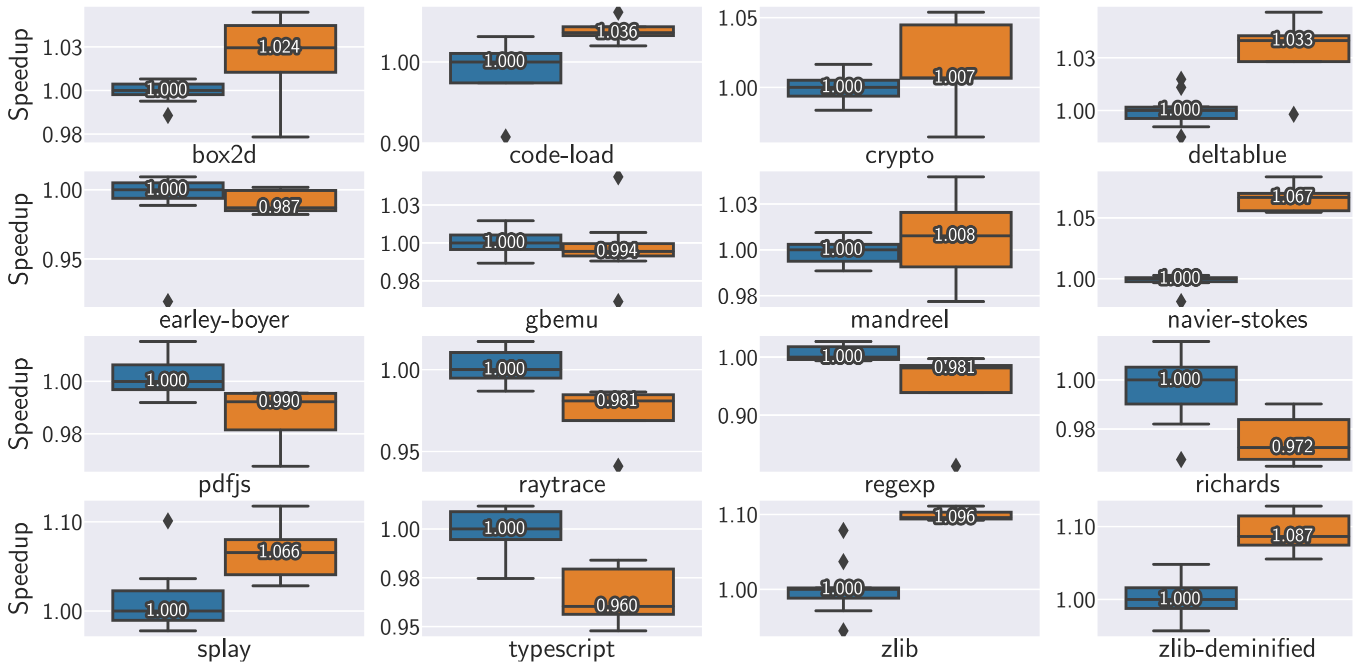


Figure 8: Octane peak performance. Higher is better.

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

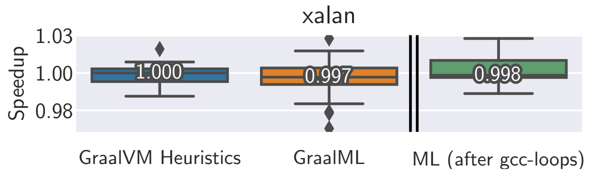


Figure 9: xalan (DaCapo) peak performance. Higher is better.

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

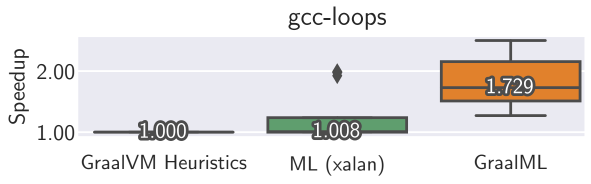


Figure 10: gcc-loops (JetStream) peak performance. Higher is better.

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 hyper-parameter 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